Poznań University of Technology
Faculty of Computing
Institute of Computing Science

Master of Science Thesis

Development of the environment for distributed computing in the Framsticks system

Piotr Śniegowski, 84891

Supervisor
dr hab. inż. Maciej Komosiński

Poznań, 2013.

Contents

Chapter 1

Introduction

1.1 Motivations

The Framsticks system is primarily an environment for simulation of evolution of artificial life, but it can also be employed to perform computations in general [KU96], [KU09]. Using Framsticks, one can define parameters like evolution goal or environment conditions and then observe the evolution of complex life-like structures. It is also possible to express more sophisticated elements, like artificial neuron logic, in a dedicated scripting language: `Framscript`.

The core element of the Framsticks system is a virtual machine that controls the behaviour of the system, including genetics, simulation, and various algorithmic issues.The virtual machine has one important characteristic, which has started to become a major limitation to the possible experimentation scenarios: its process flow cannot be easily distributed. In order to overcome this limitation, a notion of distributed simulation environments or experiments – using multiple Framsticks virtual machines as computational nodes – was devised. The original simulation would be then extended with a controlling entity, exchanging information with the computational nodes over the Framsticks network protocol [FNP13]. This thesis strives to provide such an extension.

1.2 Scope of this thesis

For the case of the thesis a rich set of Java source code has been developed, enabling easy creation of various experiments in the Framsticks environment and providing a Graphical User Interface to those experiments, as well as to already existing Framsticks native simulators. This set will be referred to as Framsticks Java Framework throughout the thesis (FJF in short). Using FJF, a prototype experiments has been conducted in order to prove solution's overall correctness and provide good starting point for future users using FJF to build actual, probably more sophisticated experiments.

1.3 Thesis structure

Chapter 2 will introduce the Framsticks system in more details and present main goals for the Framsticks Java Framework.

Chapter 3 will discuss several aspects of the **Java** programming language that were found especially important for the final form of the FJF.

Chapter 4 will present the development and testing environment by discussing several tools that have proved to be helpful during the development process of FJF.

The following chapters will concentrate on the FJF itself.

Chapter 5, which constitutes the core of this thesis, will describe in details the developed software solution. Each package will be presented, with an emphasis laid down on important design aspects and non-trivial implementation details. All important elements of the framework will be described and their place in the system as a whole will be shown.

Finally, Chapter 6 will present the top-most layer of the FJF, which concentrates strictly on the notion of the distributed experiment definition and rests upon all software elements described in the previous chapter. Features of this layer will be demonstrated on two examples, of which the first one is completely synthetic, and the second one is an extension to the standard Framsticks experiment.

Chapter 7 contains a summary of work the that has been done during the development of the FJF.

Chapter 2

Goals of the Framsticks Java Framework

This chapter will present several main goals of the software solution being developed in the scope of this work, namely the Framsticks Java Framework. Beside the presented goals, the FJF should strive to provide an extensible base for other Framsticks applications.

The FJF should not be designed to be a self-contained solution, but instead to work closely with the Framsticks virtual machine in its server configuration. Because of that fact, the names of Framsticks virtual machine and Framsticks server will be used interchangeably. This Framsticks server, used as a computational node by the FJF, will also be referred to as native Framsticks server, in opposition to the **Java**-based servers hosted by FJF.

2.1 Supporting different kind of experiments

Although the Framsticks system is primarily used for evolution simulation, it is not limited to those applications. One of important project assumptions behind the FJF, is not to limit the experimenter only to develop evolution-based experiments, but to give an extensible and configurable tool, adjustable to virtually any kind of experiment possible in the original Framsticks environment. Because of that, references to the artificial evolution will be rather rare throughout the following document.

2.2 Supporting the netload/netsave communication scheme

The Framsticks server publishes an interface to save and load experiment state, through `netsave` and `netload` procedures available over the network protocol under the `/simulator` path. In order to support experiment state persistence, an experiment definition must implement `onExpLoad` and `onExpSave` functions (in `Framscript` language). The interface does not enforce any particular format of data sent over, but typically experiment state is encoded in Framsticks file format. The approach presented above, which will be referred to as the `netload/netsave` interface, up until now was used only for checkpointing or debugging of experiments involving only a single Framsticks server, and whole experiment logic was expressed in the experiment definition script.

The FJF should support a more sophisticated approach, where the `expdef` script would only include definition of operations to be conducted by a single node, while policies of work distribution and results aggregation should be expressible in possibly short snippets of **Java** code, which would be executed in the FJF experiment environment. The experimenter should be given a possibility of focusing only on the aspects specific to one's experiment, leaving things like communication and experiment infrastructure management to the FJF, and reusing common building blocks from other experiments.

2.3 Infrastructure management

The management module should provide means to connect to existing native Framsticks servers, as well as to start new instances on the as-needed basis, which each such server considered constituting a single computational node. The computational servers should be runnable not only locally, but also on remote hosts.

2.4 Experiment state monitoring and controlling

In order to give experimenter a full insight into the conducted experiment, the FJF should provide means to publish arbitrary properties and functionalities of the defined experiment, which should be made accessible over the same network protocol, as the one used by native Framsticks server. Any existing client implementing the Framsticks network protocol would thus be enabled to access and control not only the native Framsticks servers but also the experiment itself. The FJF should also provide a compatible client in form of full-featured GUI (presented below), as well as a library providing low-level interface embeddable in an arbitrary **Java** application.

2.5 Graphical User Interface

In order to facilitate the experiment management, a specially crafted Graphical User Interface should be developed. The **GUI** front-end should present a browsable tree structure of a remote server, which should be resolved only as needed. Each remote object should be presented as a list of possibly modifiable attributes, callable procedures and events, for which user be able to subscribe. Users should also be given a possibility to connect to multiple servers, both native and FJF ones, from a single Framsticks **GUI**. It should also be possible to host experiment environment directly in the **GUI** (not using network layer between **Java** based entities).

2.6 Clear and layered architecture

One of the important goals of the FJF is to develop a well-established, clean and layered architecture. All packages should be designed with extensibility in mind and to be reusable in any **Java** application existing in the Framsticks system. The lower-layer packages, like those providing parsing functionalities or mirroring the Framsticks object model should have no elements specific to the main goal of distributed experiments.

All upper-layers should represent an asynchronous processing paradigm, which is dictated mostly by network communication with remote servers, but in a minor degree also by GUI interactions.

All main functional entities should be designed to be run inside of the FJF environment, and possibly inside other entities.

Chapter 3

Java

In the following chapter, several aspects of **Java** programming language, which played an important role in presented software solution, will be introduced and briefly described. Following, two programming notions, which may be encountered in several key places throughout the project, will be discussed (immutability and fluent interfaces). Throughout the chapter, some references or minor comparisons with two other languages will be held, namely **C++** and **C#**.

3.1 Java Reflection

The `Reflection API` of **Java** language allows to inspect code, including class inheritance, fields and methods, during run-time. One of important advantages of using reflection is the ability to easily provide extension points to the application that are filled or configured during run-time, for example by using an instance of class which name was read from a configuration file, or to invoke a method specified by name by the end-user. Such capabilities are used extensively throughout the FJF, in almost all its packages.

Using reflection, however, needs some amount of consideration, since it implies performance penalties and security issues, starting with the most trivial situations like modification of object's private fields. The reflection is available since version 1.1 of the **JDK**.

3.2 Java Annotations

Java annotations allows the programmer to mark various language entities (classes, fields, methods) with an additional information, which can be later queried to determine application behaviour. Beside the built-in annotations (like `@Deprecated` or `@Override`), it is possible to specify custom annotation types. Those custom annotations may define attributes of a limited set of **Java** types. Annotations are commonly used by various **Java** frameworks to define behaviour that would otherwise need to be expressed in external resources. One possible example of application heavily depending on the annotations are database-related frameworks, which use annotations to associate given **Java** class with a specific database table or class fields with the table columns. The FJF is another example of such application, where annotations are used to define the relations between **Java** classes and Framsticks types.

Support for annotations in the **Java** language started with version 1.5 of the **JDK**.

3.3 Weak references

Their purpose is to hold a reference to the object (called referent), but not to prevent it from being gathered by garbage collector. The obvious implication of this semantic is that dereferencing operation is not always possible - once the last regular (or strong) reference to the object is lost, the dereferencing operation returns `null`. One of applications of weak references are the event handlers (or listeners), that prove themselves problematic especially in dynamic GUI applications. This situation is one of counter examples to the popular misconception that garbage collectors in managed programming languages (like **Java**, **C#**, etc.) protect against memory leaks. The typical counterparts of **Java** `java.lang.ref.WeakReference` class, are: `std::weak_ptr` in **C++** and System.WeakReference in **C#**.

Weak references are available in **Java** language since version 1.2.

3.4 Generics

Generics are an important feature of **Java** programming language, since they allow to write more type-safe code and to omit unnecessary casts. A popular misconception, partially resulting from similar wording, is that they are similar to **C++** templates. It may be disputed, however, that they have more differences than they have in common. **C++** templates system is much more complex (and even Turing complete). **C++** template arguments can be types, constants, function addresses and even other templates, while in **Java** only types can be the arguments of generic entity (class, interface, method or constructor). In **Java**, generics are a compile-time feature, which is not directly available in run-time due to process called type erasure.

It may be considered interesting, that simple usage of generics leaves much less information regarding type in run-time, that it is the case of **C++**. However, when combined with the `Class<T>` idiom, it gives much more information than **C++** because of reflection subsystem.

Generics were introduced in version 1.5 of the language.

```
public interface WorkPackage<S extends WorkPackage<S>> extends NetFile {
    ...
    S getRemainder(S result);
}
```

```
...

public class PrimePackage implements WorkPackage<PrimePackage> {
    ...
    @Override
    public PrimePackage getRemainder(PrimePackage result) {
        ...
    }
}
```

Listing 3.1: A non-trivial real-world example.

3.5 Anonymous inner classes

The notion critical the Framsticks Java Framework code readability is the anonymous inner class (introduced with version 1.1 of the **Java** language), that is typically used in asynchronous applications. Because of the inherently asynchronous nature of the FJF (expressed in section 2.6), they can be found throughout the framework.

Anonymous inner classes may be seen as an extension of regular inner classes. Instances of both class types are bound with a specific instance of outer class, hence they have transparent access to all fields and methods of the given instance creating them. Anonymous classes are typically used to implement a callback or listener interface and are constructed in specific context, possibly capturing some values from enclosing scope. Because of their limited usage, they do not need to be named by programmer – they are given a name by **Java** compiler.

Listing 3.2 presents a non-trivial usage example that can be found in FJF, where anonymous classes are nested.

```
class NetLoadSaveLogic {
    ...
    protected void issueNetloadIfReady(ListChange change,
            final Simulator simulator) {

        if (!change.hasHint("ready")) {
            return;
        }
        netload(simulator, new Future<NF>(simulator) {
            @Override
            protected void result(final NF net) {
```

```java
                if (net == null) {
                    log.debug("no file for upload provided");
                    return;
                }
                simulator.netload(net, new Future<Object>(this) {
                    @Override
                    protected void result(Object result) {
                        NetLoadSaveLogic.this.messages.info("netload",
                            "done " + net.getShortDescription());
                        log.debug("netload of {} done",
                            net.getShortDescription());
                        simulator.start();
                    }
                });
            }
        });
    }
}
```

Listing 3.2: Example of anonymous classes usage.

Several important aspects may be noticed here:

- the `simulator` variable must me marked as `final` to be accessible from the callback,
- constructors of the abstract class `Future` can be used to create the anonymous subclass,
- the `this` argument passed into the constructor of the nested callback refers to the instance of enclosing callback, not to the instance of enclosing `NetLoadSaveLogic`
- because of above, reference to the `messages` field of `NetLoadSaveLogic` must be referenced using following syntax: `NetLoadSaveLogic.this.messages`

Although anonymous classes may be perceived as hard to read, it has to be noted that the alternative would need explicit classes implemented outside of the only scope in which they are meant to be used, with context arguments (like `simulator` in the example) doubled as fields of such hypothetical class.

3.6 Immutable objects

The notion of immutable objects is used throughout the **Java** programming language, and it is has several application in the Framsticks Java Framework as well. The object is said to be immutable, if it does not change its logical state after construction. The most simple and often occurring example of such class is the `java.lang.String` type. All `String` methods, like `substr` or `concat` never change the object itself, but return new instances of `String` class. This allows to safely pass instances of type `String` throughout the program, including between threads. Extended use of immutable types allows to skip otherwise necessary synchronization and locking.

This approach is used in case of `com.framsticks.structure.Path` class (which will be presented more closely in 5.6), which never changes its state. Operations like appending or removal of path's elements always effectively create new `Path` instance.

Another, not so obvious example of immutable types are boxing classes, like `Integer`, `Double`, etc. In scope of FJF, similar to them is `Param` class with all its descendants, which provides meta information for a single value in FJF, and may also be used as a field in enclosing `FramsClass`. `FramsClass` is also immutable and is semantically a counterpart to the (also immutable) `java.lang.Class` class. (Both `Param` and `FramsClass` are available in package `com.framsticks.params`, which will be presented in 5.2).

Chapter 4

Development environment

In this section several important software elements constituting the development environment will be briefly presented and their impact on the development process and quality of code will be discussed.

4.1 Maven

Maven [MAV13] is a project management tool implementing concept of project object model.

Using `Maven` gives several profits:

- dependencies on external libraries are easily expressible, and are automatically resolved during build time,
- project description is `IDE`-agnostic: instead of binding project to a specific development environment, developer may generate project files adequate for the `IDE` of preference using plugins.
- a variety of plugins exist supporting testing and code analysis, either by themselves or by interfacing external tools.

Listing 4.1 presents some commonly used `Maven` commands.

```
# run find bugs
mvn findbugs:findbugs
# execute the default configuration
mvn exec:exec
# execute all test for the project
mvn test #test
# prepare project files for an IDE
man eclipse:eclipse
```

Listing 4.1: Various Maven commands.

However, beside many unquestionable advantages, some particularly uncomfortable drawbacks were identified, like lack of out-of-the-box support for creation of simple executable files wrapping the `JVM` invocation and designated for distribution to the end-user.

4.2 FindBugs

FindBugs [FIN13] is a static analysis tool for the **Java** programming language that proved itself very useful during development of Framsticks Java Framework. **FindBugs** integrates well with Maven. Below are presented several issues found by **FindBugs** tool in the FJF.

RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE Indicates superfluous checks for `null` as the returned value, if the method is marked with `@Nonnull` annotation.

```
public @Nonnull TreePath convertToTreePath(Path path) {
    // no other return statements
    return new TreePath(accumulator.toArray());
}

final TreePath treeListPath = convertToTreePath(listPath);
if (treeListPath == null) {
    throw new FramsticksException()
        .msg("path was not fully converted")
        .arg("path", listPath);
}
```

Listing 4.2: Redundant null check.

NN_NAKED_NOTIFY Indicates places where `Object.wait()`, `Object.notify()` are done without any state change accompanying those calls (like setting some flag telling that condition is now valid).

HE_EQUALS_USE_HASHCODE Problems arising when objects of class overriding `Object.equals()` and not overriding `Object.hashCode()` method are particularly hard to track down, since they silently break some assumptions taken by other Java entities (like `java.util.HashMap`).

NP_LOAD_OF_KNOWN_NULL_VALUE Listing 4.3 presents an issue found with **FindBugs**. The `javaClass` input parameter was used an iterator in `while` loop (finishing that loop with `null` value), but is used again a key to the `setsCache` map, clearly in the original meaning (as it was passed to the function). In the presented example the issue did not manifest itself in any obvious way, since `null` was used as a key in `Map` serving as a cache, rendering that cache unusable. It is worth to mention a good practice of specifying input parameters to methods as `final` values, i.e. not changing values during the method execution, since it is a typical situation, and it would prevent the presented mistake.

```
public static Set getAllCandidates(Class<?> javaClass)
        throws ConstructionException {
    Set result = setsCache.get(javaClass);
    if (result != null) {
        return result;
    }
    List<Class<?>> javaClasses = new LinkedList<>();
    while (javaClass != null) {
        javaClasses.add(0, javaClass);
        javaClass = javaClass.getSuperclass();
    }

    result = new Set(...);
    /* the main method logic filling up the result instance */
    setsCache.put(javaClass, result);
    return result;
}
```

Listing 4.3: Load of null value.

4.3 TestNG

```
@Test
public void buildModel() {
    ...
    assertThat(model.getParts().get(2).getPosition()
            .sub(new Point3d(2.27236, -0.0792596, -0.958924)).length()
        ).describedAs("position error").isLessThan(0.0001);
}
```

Listing 4.4: Fluent assertions in TestNG.

TestNG is a testing framework that was chosen for the Framsticks Java Framework. Initially, **JUnit** was used; migration to a different framework was dictated by several issues:

- **TestNG** supports data-driven testing [TES13] (an example from FJF is presented in listing 4.5),
- **TestNG** allows to express dependencies between tests,
- **TestNG** provides expressive fluent interface for assertions (presented in 4.4).

```java
@Test
public class RequestTest extends TestConfiguration {

    @Test(dataProvider = "requests")
    public void parsingAndPrintingRequests(
            Class<? extends Request> requestClass,
            String line) {
        Pair<CharSequence, CharSequence> pair = Request.takeIdentifier(line);
        Request request = Request.parse(pair.first, pair.second);
        assertThat(request).isInstanceOf(requestClass);
        assertThat(request.stringRepresentation()).isEqualTo(line);
    }

    @DataProvider
    public Object[][] requests() {
        return new Object[][] {
            { CallRequest.class,
                "call /object function first second \"thi rd\""},
            { GetRequest.class, "get /test"},
            { GetRequest.class, "get /test one_field"},
            { GetRequest.class, "get /test first_field,second_field"},
            ...
        };
    }
}
```

Listing 4.5: Data-driven testing in TestNG.

The simplicity of the approach to data-driven testing found in **TestNG** encourages developers to avoid the common testing anti-pattern of creating multiple testing methods with only some parameters changing.

4.3.1 TestNG drawbacks

Although **TestNG** proved to be a good choice, several drawbacks were identified. One of them is that the order of calling methods annotated with `@BeforeClass`, `@AfterClass`, `@BeforeMethod`, `@AfterMethod` is unspecified, which renders usage of those methods unstable, thus unusable, in situations involving test classes constituting some inheritance hierarchy.

4.3.2 Testing multi-threaded application

Although very useful, **TestNG** needed to be customized for the Framsticks Java Framework specifics of inherently multi-threaded environment. Several tests, like communication and hosting tests, included situations where spawning user threads was needed. If any exception was thrown in those threads, in particular `AssertionError` raised by failed **TestNG** assertion, it was propagated up to the enclosing `java.lang.Thread`, where it was handled by the default thread exception handling routine, resulting in mere printing stack trace and finishing thread, hence no failing to instrumented test.

The **JDK** class `java.lang.Thread` exposes a facility to handle such situations: it allows to register an instance of class implementing `Thread.UncaughtExceptionHandler` interface. If an exception is not caught earlier, it is passed to that object. Although supported out-of-the-box by **JDK**, this approach proved to be not suitable for the testing purposes of FJF, because it is being executed completely outside of the FJF stack, which prevents proper threads joining.

For this reason, a special routine was added to the `TestConfiguration` class (a utility base class for all test classes in FJF). Method `failOnException()` constructed on the fly a special `ExceptionHandlerInstance`, that remembers handled exception (possibly encloses in `AssertionError`) and pushes it to the queue of test assertions. That queue is checked after each `@Test` annotated method returns, which happens always in the main thread monitored by the **TestNG** framework, and if any `AssertionError` is found, it is rethrown and then caught by the **TestNG**, thus failing the test.

4.4 GUI testing – FEST

Thorough unit testing of various Framsticks Java Framework elements allowed more effective development cycle, since various regressions were identified immediately after introduction. Using **TestNG** it was relatively easy – keeping in mind problem described in the previous paragraph – to integrate tests of the solution as a whole (for example tests including connecting to an experiment hosted in the framework itself).

Still, one important part of the solution remained untested, namely the Graphical User Interface. GUIs are especially time consuming in case of manual tests; at the same time such tests may be very useful as integration tests, since any problem in a lower layer will probably manifest itself somehow in the final layer (abstracting from where the problem lies). Previously mentioned unit tests may only detect regressions in relatively small parts of the system, whereas GUI tests may detect regressions resulting from broken contracts between solution's subsystems interfaces.

FJF uses **Swing** as a GUI framework. Research regarding testing Swing applications revealed several possible solutions, including **Jemmy**, **UISpec4j** and **FEST** [FES13]. From those **FEST** was chosen because of its convenient fluent interface and supported integration with **TestNG**. **FEST** allowed for instrumenting GUI with operations like:

- click cursor on the specified button,
- choose specified item from the tree component,
- enter predefined text into the specified text box.

After bringing GUI to the wanted state, assertions may be tested against:

- proper values in specified components,
- their visibility, etc.

Using **FEST** freed programmer from the cumbersome and tedious repeating the same GUI operations over and over again. Still, during the test run, programmer was unable to perform tasks more productive than watching the test run, since obviously GUI testing needed to use screen, instrument mouse and keyboard. This drawback was dealt with using the **Xvfb** application [XVF13], that provides a virtual frame buffer (in the **X** windowing system). Using **Xvfb** it is possible to run GUI tests in background as any other tests. It is worth noting that **Xvfb** may also be used to run such tests on a remote server lacking graphic card.

```
xvfb-run -s '-screen 0 1920x1020x24' maven test
```

Listing 4.6: Command to run all tests in virtual frame buffer of HD resolution.

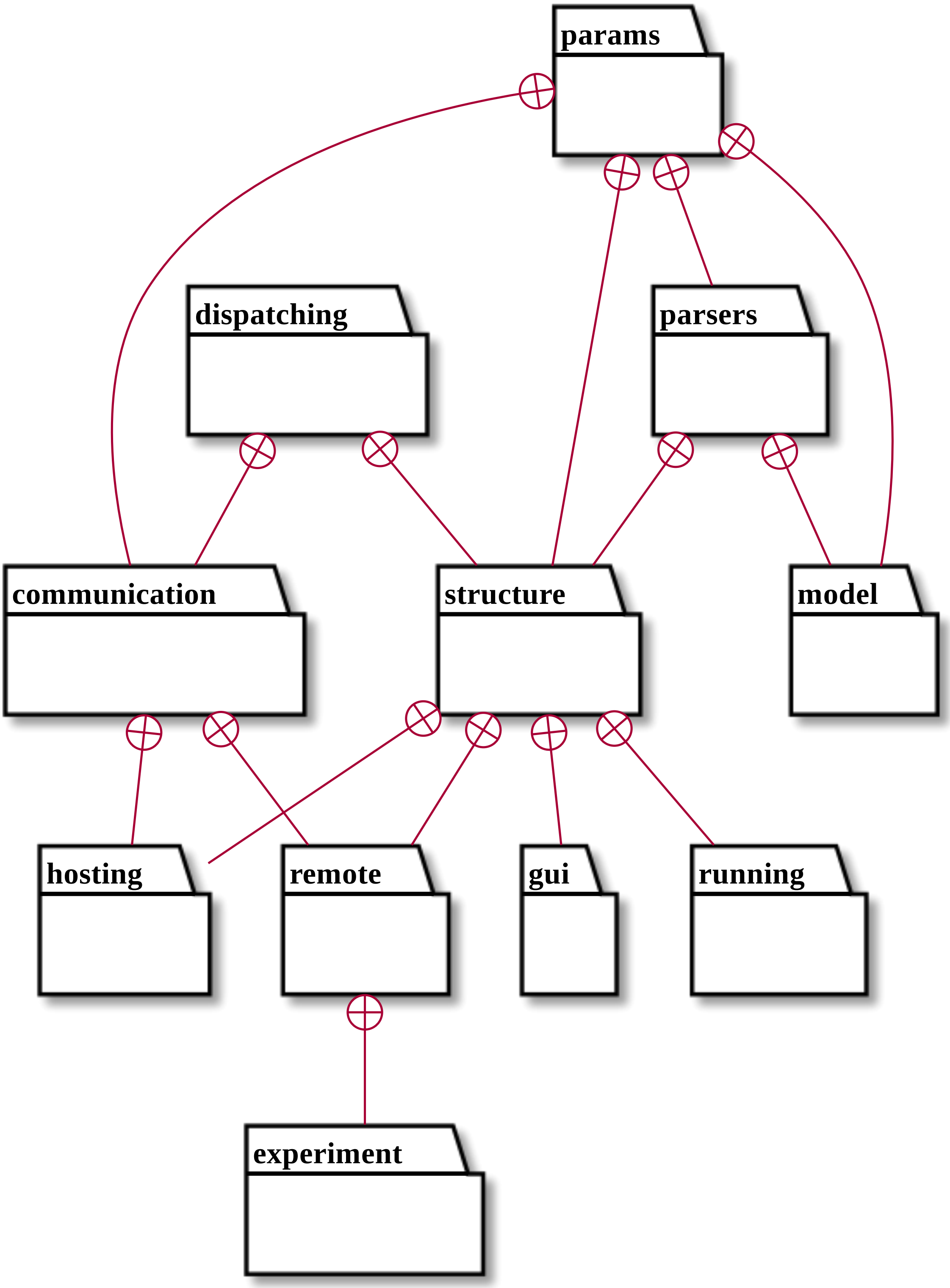

Figure 4.1: Packages relations.

Chapter 5

Framsticks Java Framework

5.1 Packages

The Framsticks Java Framework is organised into over a dozen packages of different character. All of those subpackages are direct or indirect subpackages of `com.framsticks`. Figure 4.1 presents dependencies between packages, while table 5.1 most important classes for each package. Special care was taken so that dependency graph was a in fact a tree, i.e. there are no cyclic dependencies. Although **Java** language does not impose such a constraint, it makes the packages and classes layout in the project more intuitive and easily understandable.

It is clearly visible that the `params` package constitutes a root package in the dependency tree, while the packages like `gui` or `experiment` are the leaf packages, finally aggregating functionalities spread throughout the FJF.

Main packages, containing the most specific functionalities, are: `params`, `communication`, `structure`, `hosting`, `remote`, `gui` and `experiment`. They will be presented in more details in the following sections.

Package `util` is not presented in 4.1, it contains various utilities not specific to the FJF (it can be seen as a extension to `java.util` standard package).

5.2 Params

It may be seen in figure 4.1, that the package `com.framsticks.params` constitutes a foundation for all other Framsticks Java Framework elements. The following section will confirm its importance to the solution as a whole. Moreover, by referring to the upper layer packages, many implementation decisions, that would otherwise remain unclear, will be justified.

This package's importance comes mainly from the fact that it implements the interface with the Framsticks server, i.e. the parameters entity description. That description is used throughout the Framsticks system to encode virtually all entities, including genotypes, creatures, experiment definitions and settings.

Package	Major classes
`params`	`Param`, `FramsClass`, `Access`
`communication`	`Request`, `Connection`
`experiment`	`Simulator`, `Experiment`
`gui`	`Browser`, `Frame`, `Panel`, `Control`
`hosting`	`Server`, `Cli`
`model`	`Genotype`, `Creature`
`parsers`	`MultiParamLoader`, `XmlLoader`
`remote`	`RemoteTree`
`running`	`FramsServer`
`structure`	`Tree`, `Path`

Table 5.1: Major classes.

Package `com.framsticks.params` contains following important elements:

- `FramsClass` class representing Framsticks type (a counterpart to `java.lang.Class`),
- `Param` class and its extensions describing various Framsticks type members,
- `Access` class and its extensions providing unified and simple access to object instances,
- annotations used to mark **Java** classes meant to be used as direct storage for data downloaded from Framsticks server (like `com.framsticks.model.Part`).

5.2.1 Annotations

Annotations (from package `com.framsticks.params.annotations`) are also used to describe Java classes that do not have their counterparts in the Framsticks server, but are used to build distributed evolution experiments. Those annotations allow to automatically read their configuration and set experiment up or to prepare GUI (this kind of usage will be discussed in section 5.3 regarding `parsers`).

There are two main annotations: `FramsClassAnnotation` and `ParamAnnotation`

ParamAnnotation This annotation is used to annotate **Java** class members: both fields and methods. The process of inferring Framsticks type for most members is straightforward. Fields can be used directly (possibly circumventing the **Java** non-`public`) access descriptors) or through access methods (getters and setters), which are considered as such if are following **Java** naming conventions. They can also be marked explicitly as the interface to a field, by using `ParamAnnotation`'s `paramType` attribute. Methods are also converted automatically, including their formal parameters types and the return value type.

One of Framsticks parameter types, namely the `event` type (represented in FJF by `EventParam`), is not directly expressible in the **Java** programming language. In this case the conversion proceeds in a way similar to the one of access methods discovery – it assumes two things for these methods:

- methods naming convention (`addEventListener` and `removeEventListener`),
- their argument type: `EventListener<Argument>` interface.

Of course methods can also be explicitly marked as the ones providing an interface to the event.

FramsClassAnnotation While previous annotation was used to annotate class members, `FramsClassAnnotation` is used to annotate an arbitrary **Java** class as FJF-compatible; only classes marked as such are scanned for members annotated with `ParamAnnotation`. Furthermore, only classes marked with `FramsClassAnnotation` can be used as storage for `ReflectionAccess` (presented closely in the following part).

The declaration of `FramsClassAnnotation` is presented in listing 5.1 together with a simple usage example.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FramsClassAnnotation {
    String name() default "";
    String id() default "";

    String[] order() default {};
    Class<?>[] register() default {};
    String[] registerFromInfo() default {};
}

...

@FramsClassAnnotation(id = "p")
public class Part {
    ...
}
```

Listing 5.1: FramsClassAnnotation.

It is worth to note that the presented annotation is also annotated with two important pieces of information: first defining that the annotation should be available in run-time (through Reflection API), and the second specifying that the FramsClassAnnotation can only be used to annotate **Java** types.

Beside the obvious attributes like id and name, FramsClassAnnotation also provides attribute order, which needs to be used, if the ordering of parameters in the FramsClass has to be deterministic, since the **Java** reflection layer does ensure any particular ordering of the members found in java.lang.Class. In practice that ordering is important because of only one aspect: presentation in user interface, where a layout stable accross FJF invocations is desirable. That ordering is mainly useful with classes designed to viewed in GUI.

5.2.2 Param hierarchy

Param class is a root class for all params available in the Framsticks system. Param objects are used to represent fields in classes as well as collections' elements. They are lightweight entities and are not bound to any particular instances of FramsClass or Access classes. For example, ListAccesses create such Param objects on the fly, based on the underlying collection being accessed.

Figure 5.1 presents a complete hierarchy of all its descendants. All classes extending Param are immutable, which allows to safely pass them in multi-threaded environment. The labels above inheritance arrows designates the type that used as the parameter to the generic superclass (for example: BooleanParam extends PrimitiveParam<Boolean>).

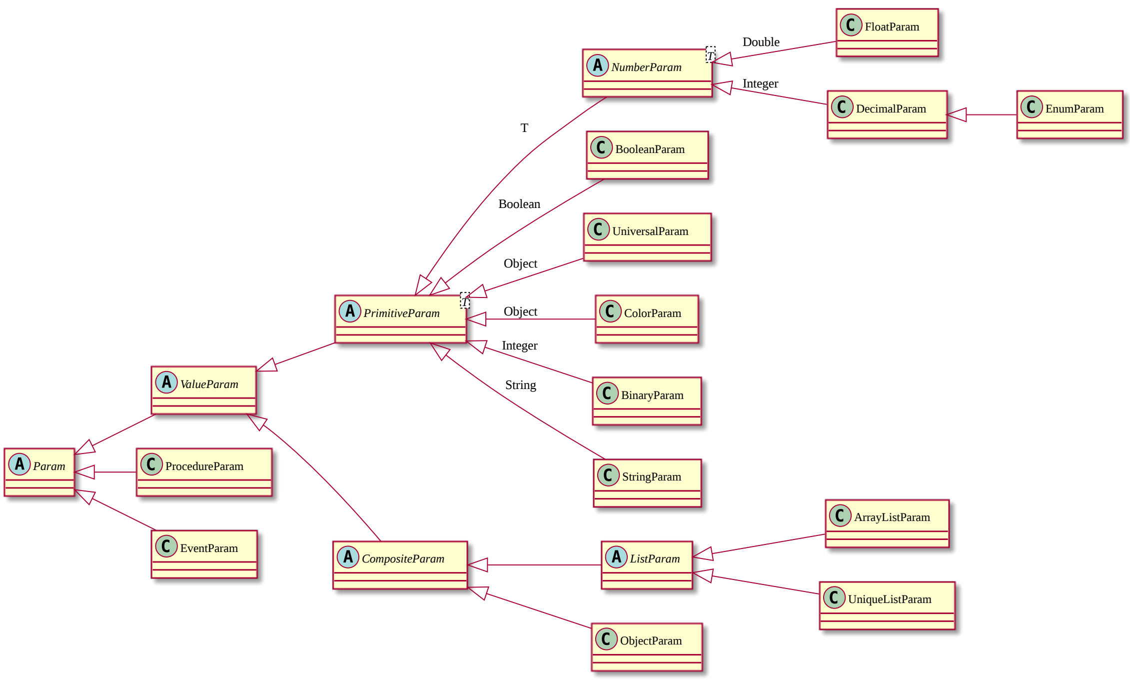

Figure 5.1: Param class hierarchy.

Presented hierarchy is very important and useful, since it allows to narrow down the accepted argument type from Param to its subclass in many places throughout the Framsticks

Java Framework:

- `com.framsticks.gui.controls.ValueControl` holds only params extending `PrimitiveParam`,
- `com.framsticks.gui.tree.TreeNode` holds only params extending `CompositeParam`,
- only params extending `PrimitiveParam<?>` are printed/parsed directly by typical serialization algorithms,

It also allows to easily filter out `Param`s based on their type, which is presented in listing 5.2.

```
Access access = bindAccess(path);
for (EventParam eventParam
    : Containers.filterInstanceof(access.getParams(), EventParam.class)) {
    /* register for event */
}
```

Listing 5.2: instanceof-based filtering.

Table 5.2 presents mapping between two types representations: Framsticks string representation and FJF class representation. Conversion from textual representation to a `Param` instance is enclosed in `ParamBuilder` (described below), while the conversion in opposite direction is done by the `Params` types themselves.

Framsticks type	FJF type
`d`	`DecimalParam`
`d 0 1`	`BooleanParam`
`d 0 2 ~Add~Remove~Modify`	`EnumParam`
`f`	`FloatParam`
`s`	`StringParam`
`p`	`ProcedureParam`
`e`	`EventParam`
`x`	`UniversalParam`
`o Creature`	`ObjectParam`
`l Joint`	`ArrayListParam`
`l Genotype uid`	`UniqueListParam`

Table 5.2: Param types

5.2.3 ParamBuilder

Instance of `Param` subclasses are not constructed directly, instead a special class `com.framsticks.params.ParamBuilder` is used. The relation between `Param` and `ParamBuilder` is exactly the same as between `String` and `StringBuilder`; similarly to `StringBuilder` it allows `Param` type to be an immutable type. `ParamBuilder` is also used during parsing `prop:` sections of Framsticks classes descriptions, where the type of `Param` is known only after reading field `type:`; `ParamBuilder` allows to store values of all other fields and at the end construct actual `Param` instance of appropriate type deduced as it shown in table 5.2. Another advantage of `ParamBuilder` is its convenient fluent interface, which allows to build up `Param` instances in concise manner (shown in listing 5.3) by referring only to those `Param` fields that are actually needed. Once all parameters that are different from the default values are given, the call to method `finish()` is issued, which finally constructs the new instance of adequate `Param`'s subclass. Single `ParamBuilder` can be reused to build another `Param` instances, possibly changing only a subset of parameters; this ability is used as an optimization by `ListAccess` types to setup parameters common to all list's elements beforehand.

```
Param param = Param.build().id("simi")
    .group(1)
    .flags(READONLY | DONTSAVE)
    .name("Similarity")
    .type("f")
    .finish();
```

Listing 5.3: Building Param instance.

5.2.4 FramsClass

It was stated before that `com.framsticks.params.FramsClass` constitutes an analogy to `java.lang.Class`; it is also immutable. It is used as named container for `Params`, which can be grouped into named groups; it exposes an interface to access stored `Params` by number or by their identifier. `FramsClass` can be build in tree ways presented below.

Building manually Similarly to `Param` and `ParamBuilder` relation, a special builder class for `FramsClass` exists, namely `FramsClassBuilder`. It also provides a fluent interface, allowing to add new `Params`, set up groups and other attributes in a concise manner. `FramsClassBuilder` is also used internally by the following methods of `FramsClass` building.

Building from textual representation `FramsClass` instance can be built automatically from textual representation returned by the Framsticks server.

Both `FramsClassBuilder` and `ParamBuilder` are annotated with FJF annotations, thanks to which no routines specific for the loading of `FramsClass` exist: `FramsClasses` can be loaded using generic `MultiParamLoader` (described in section 5.3) with default configuration. The use case of loading object of type `FramsClass` is presented as an example in section 5.3 regarding parsers.

Building from Annotations The most sophisticated way of building a FramsClass is an automated conversion from `java.lang.Class`. The object of type `Class`, representing a specific **Java** class annotated with `FramsClassAnnotation` is scanned for members (fields, methods and events), annotated with `ParamAnnotation`. In most situations a parameterless annotation in sufficient, parameters have to be given explicitly only if intended values are to be different from the ones automatically derived based on adopted convention.

First approach to the problem of building `FramsClass` out of **Java** classes used static methods accepting a special builder, which was filled with references to methods and fields. The adoption of annotations-based solution allowed for a more concise, readable and standard way of expressing this semantics; it also has the advantage of clearly decoupling the data (here annotations) from the algorithm (here process of building `FramsClass`).

Thanks to `FramsClass` immutability and clearly functional algorithm of conversion (it has no dependency on program state), instance reflecting a given **Java** class, once created, can be cached by the `FramsClassBuilder` and reused for all following conversion requests.

A typical use case for this approach is the hosting scenario (presented in 5.8), in which data structure expressed in native **Java** classes (like the `Experiment` and `Simulator`), is reflected into the `FramsClass` representation. It is then transmitted to the remote client in response to `info` requests, where it is interpreted using previously described method, and allows building a shadowing tree structure on the client side.

Another typical use case occurs at the client side, where a regular **Java** class can be used to store data received from server instead of the default approach using generic `PropertiesObject` – this approach is described more closely in section 5.10.

5.2.5 Accessing values

In FJF data can be stored in several composite types:

- `PropertiesObject` for objects of type not known beforehand,
- `java.util.List` for simple lists (e.g. `l Joint`),
- `java.util.Map` for uniquely identified lists (e.g. `l Creature uid`),
- any regular **Java** class annotated with `FramsClassAnnotation`.

To allow a uniform access to members of those composite types, a special layer of access objects was devised. The root of access classes hierarchy (shown in the 5.2) is the `com.framsticks.params.Access` interface, which is most commonly referred to throughout the FJF. `Access` interface declares means of getting and setting fields in a type-safe manner, calling methods and registering to events. Interfaces `ObjectAccess` and `ListAccess` are extensions to the `Access` interface and are used throughout the FJF as a type-based distinction between accesses to simple objects and accesses to lists, whilst `SimpleAbstractAccess` and `SimpleListAccess` are considered implementation specific, and should not be referred to outside the `params` package.

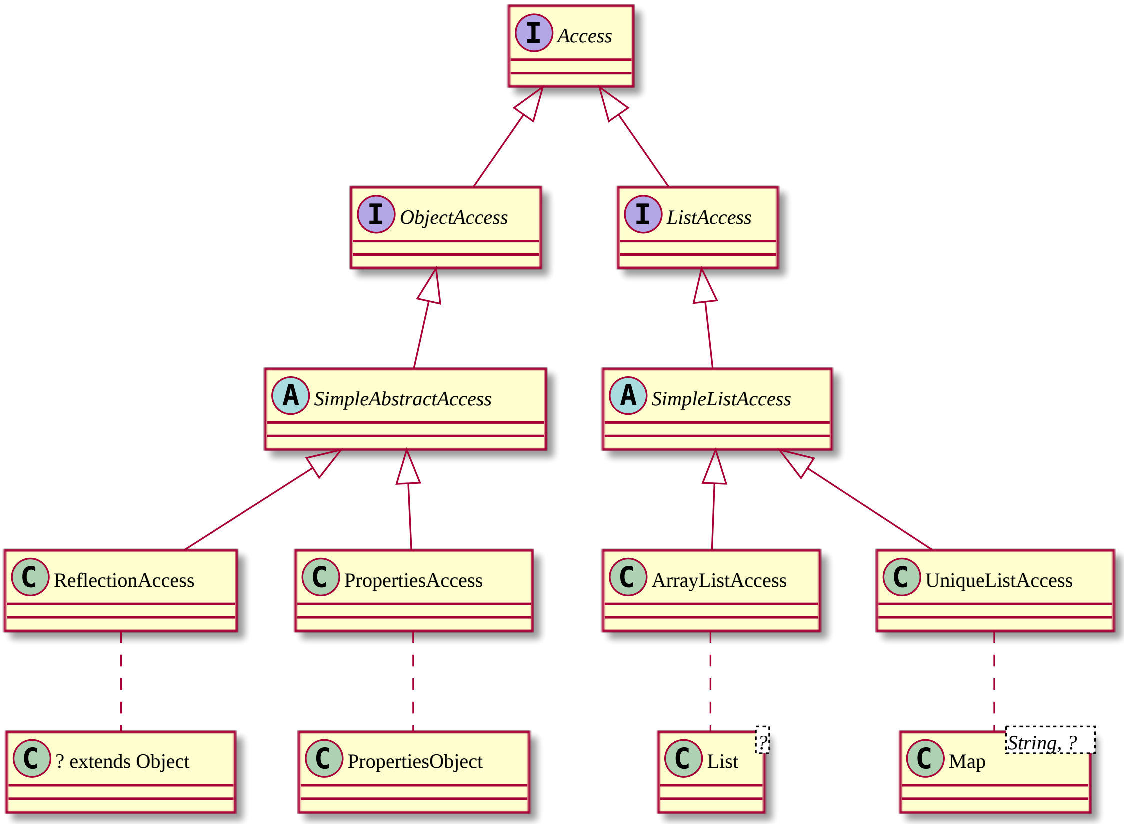

Figure 5.2: Accesses class hierarchy.

ObjectAccess and ListAccess There are two important behavioural differences between these access methods:

- the former contains heterogeneous elements, both primitives (`PrimitiveParam`) and composites (`CompositeParam`), while objects wrapped with the latter contain a set of objects of homogeneous type;
- the number and identifiers of elements in `ObjectAccess` are constant for a given `FramsClass`, while in the case of `ListAccess` it varies throughout the lifetime of the accessed list object.

Because of the presented differences between `ObjectAccess` and `ListAccess`, while the former uses `Params` directly from the stored `FramsClass`, `ListAccess` synthesises `Param` instances only on demand and stores a prefilled `ParamBuilder` internally.

ReflectionAccess This way of accessing object is especially important in hosting scenarios, where it allows an arbitrary complex tree structure, expressed with regular **Java** classes, to be accessed directly and transparently by the FJF, a notable example of which is the `Experiment` class (described in section 6). The `ReflectionAccess` class itself is a lightweight class – all logics needed to bind members of `FramsClass` to members of `java.lang.Class` (using `ParamAnnotations`) are hidden in the `Backend` class. Since member binding involve searching through `Class` members and checking whether fields' types or function arguments are matching their counterparts in `FramsClass`, performing all those operations each time `ReflectionAccess` is created would easily outweigh its actual functionality (`Access` instances are rather short-lived objects). Because of immutability of both `FramsClass` and `java.lang.Class` it is possible to execute those operations only once for each pair of bound classes, and cache them for future use – the `Backend` itself is also an immutable type.

Another specific aspect of `ReflectionAccess` is that it is currently the only `Access` that actually supports the events registration. Methods of regular **Java** class are automatically considered as the events registration interface, if they accept a single argument of type `EventListener<A>` (where `A` designates the type of event argument) and their name starts with „add" or „remove" prefixes. Entry points not obeying that convention can be explicitly marked as the events interface.

5.2.6 Registry

The `Registry` class is an important part of the `params` package, providing functionality similar to the one of `java.lang.ClassLoader`. `Registry` maintains a set of known `FramsClass`es as well as their optional associations to **Java** `Class`es. It is used as a utility entity in several key places, one them being the `Tree` instances (presented closely in section 5.6).

5.3 Parsing

This section will present approaches to parsing two different file formats used by the FJF: Framsticks file format and `XML`.

5.3.1 Framsticks file format

The Framsticks the main data serialisation format, that is used throughout the Framsticks system for following purposes:

- experiment definitions,
- experiment states,
- object serialization,

- class description.

Content described by this format can be found in regular files (typically with extensions `*.expdef` for experiment definitions or `*.epxt` for experiment state), as well as in data sent over network protocol by the native Framsticks server in response to `get` and `info` requests (where it is delimited by the `file` and `eof` keywords). An example of the data encoded is Framsticks file format is included in listing 5.4, presenting part of response for the `info /simulator` request.

```
class:
id:Simulator

prop:
id:print
name:print information message
type:p(s text)
flags:32
help:One argument: message to be printed

prop:
id:message
name:print message
type:p(s text,d level)
flags:32
help:~
The second argument can be:
 -1 = debugging message
 0 = information
 1 = warning
 2 = error
 3 = critical error~

...
```

Listing 5.4: Example response for info request.

Most of the key aspects of the Framsticks file format are visible in the presented example:

- data is divided into object sections, each starting with the identifier of the object's type (here `class` and `prop`);
- each section consists of multiple key-value pairs, delimited with a colon;

- value spans until end of line, unless it is enclosed in ~ characters, which delimits multiline values;
- object section ends with an empty line.

It is important to notice, that the file format itself does not specify the relations between objects found in a given file. In the previous example, `prop` objects are in fact representing properties of `class Simulator`, but that relation is not expressible in the Framsticks file format.

MultiParamLoader For reading data encoded in the format presented above, FJF provides a single yet extensible parser, namely `MultiParamLoader` class. It allows to specify actions, which should be taken upon specific events (like encountering unknown class type), in terms of callbacks or processing breaks. It is designed to be a low-level reader used by other entities implementing more specific reading schemes.

5.3.2 XML configuration

The configuration of FJF entities is expressed through an `XML` file, however, **Java** classes from FJF do not need to explicitly read configuration given in this format. Instead, special `com.framsticks.parsing.XmlLoader` class was devised, constituting a bridge between class descriptions expressed through **Java** annotations (presented in section 5.2.1) and the `XML` document object model.

To allow such automatic conversion from `XML` document into **Java** hierarchic class structure, only few new elements needed to be added beside those implementing the core functionality common to FJF and Framsticks native server:

- `com.framsticks.params.annotations.AutoAppendAnnotation` used to marks methods which will be used to associate instances resulting from reading the enclosed `XML` node to the enclosing one (for example attach `RemoteTree` instance to the `Browser` instance in the listing 5.5),
- `com.framsticks.params.Builder` interface used to mark classes which should not be directly embedded in the enclosing scope, but instead an object of different type is emitted by the configured instance (an example is `FramsClassBuilder`).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Framsticks>
    <import class="com.framsticks.gui.Browser" />
    <import class="com.framsticks.remote.RemoteTree" />
    <import class="com.framsticks.model.ModelPackage" />
    <import class="com.framsticks.gui.table.ColumnsConfig" />
    <Browser>
```

```
        <RemoteTree name="localhost:9009" address="localhost:9009">
            <ModelPackage />
        </RemoteTree>
        <include resource="/common-columns.xml" />
    </Browser>
</Framsticks>
```

Listing 5.5: XML configuration example.

Listing 5.5 presents all major steps in XML configuration processing scheme, they are listed in order below:

- first, all classes specified in `<import/>` elements are searched for using standard **Java** Classpath searching mechanisms,
- an instance of `Browser` is created
- an instance of `RemoteTree` is created with name and remote server address read from attributes,
- an instance of `ModelPackage` is created, which contains list of classes to be used for direct storage,
- the `ModelPackage` instance is added to the enclosing `RemoteTree` which causes registration of classes like `Genotype` or `MechJoint` in the `RemoteTree`'s class registry (described in 5.2.6),
- the `XML` document referred to from the `<include/>` statement is read and processed as it was embedded directly (in this example it contains a common configuration of columns to be displayed in the table views in **GUI**).

`XmlLoader` is also used to read the `f0` scheme – that process is described in section 5.10.1 regarding the `model` package.

5.4 Multithreading

Presence of elements such as network communication and GUI in Framsticks Java Framework forces it to be a heavily multi-threaded solution. Obviously, the first issue to be resolved in such an environment is to protect against possible race conditions. Since the tree is the central data structure in the FJF, a solution to that problem based on explicit locking would be very tedious and susceptible to deadlocks.

Furthermore, an inherent asynchronous aspect of mentioned elements requires from the FJF infrastructure a simple way of dispatching computations to be executed in the future, may it be after other computation or communication is done, after some specified time or as soon as possible.

Package `dispatching` is a response to those problems – it implements a threading model with support for asynchronous task dispatching.

5.4.1 Dispatcher

Because of conditions stated above, a solution based on task dispatchers was adopted. Mentioned dispatchers are represented by interface `Dispatcher` presented in 5.14:

```
package com.framsticks.util.dispatching;

public interface Dispatcher<C> extends Joinable {
    public boolean isActive();
    public void dispatch(RunAt<? extends C> runnable);
}
```

Listing 5.6: Dispatcher.

Most entities implementing the `Dispatcher` interface are direct proxies to underlying dispatchers (with an exception of `RemoteTree`, which is described in 5.6), with only three classes actually managing assigned tasks:

- `com.framsticks.util.dispatching.Thread` which enriches `java.lang.Thread` with a task queue,
- `com.framsticks.gui.SwingDispatcher` which encloses `javax.swing.SwingUtilities`,
- a trivial case of `com.framsticks.util.dispatching.AtOnceDispatcher` that executes assigned task immediately, and is considered always active.

Most operations regarding `Tree` instances are only allowed to run in that instance's context; also almost all GUI operations are executable only from the specified GUI context (being in fact the **Swing** dispatcher thread).

Static context confinement As it can be seen in listing 5.14, `Dispatcher` interface is a generic interface parametrised with a single argument `C`, which stands for „context". Because of the reference to that parameter in the `dispatch` method signature, it effectively limits the set of acceptable runnables only to those explicitly marked to be executed in that context, which is presented in listing 5.7, where also the choosing of name for `RunAt` interface becomes clear, since it can be read in the call-place as: „dispatch new task to be run at browser". This way, the proper execution context has to be always clearly stated at the dispatchment place.

```
    final Path p = Path.to(tree, "/");
    log.debug("adding path: {}", p);
```

```java
dispatch(new RunAt<Browser>(this) {
    @Override
    protected void runAt() {
        mainFrame.addRootPath(p);
    }
});
```

Listing 5.7: Example of dispatching operation.

That limitation is a completely compile-time solution – compiler marks as errors passing of non-matching `RunAt`s to `Dispatcher`s. Only after implementation of presented approach, several issues were found regarding passing of runnables to be executed in a wrong context, that was opening possibilities for race conditions.

Dynamic context confinement Execution of code in proper context can also be controlled in run-time using mechanism complementary to the one presented above.

`Dispatcher` interface provides the method `isActive()`, which is typically used in **Java** assertions at the beginning of methods (an example presented in listing 5.8).

```java
public final class TreeOperations {
    ...
    public static @Nonnull FramsClass processFetchedInfo(Tree tree,
            File file) {
        assert tree.isActive();
        ...
    }
}
```

Listing 5.8: Activity assertion.

Those assertions, placed throughout the FJF in all methods of classes accessible from different threads (like `Tree`, `Browser` or `Connection`, but not `Access` or `Param`), enable the developer to find context confinement issues fast, not waiting for actual failures that can would be otherwise hard to understand and may even never happen in the development or testing stage. Typically, in production environment **Java** virtual machine is running with disabled assertions, hence that solution incurs no running time penalty.

5.5 Communication

The `communication` package implements the Framsticks network protocol, as described in [FNP13]. An important design aspect of this package is an abstraction from the actual local representation of the remote server tree – the FJF provides such an implementation

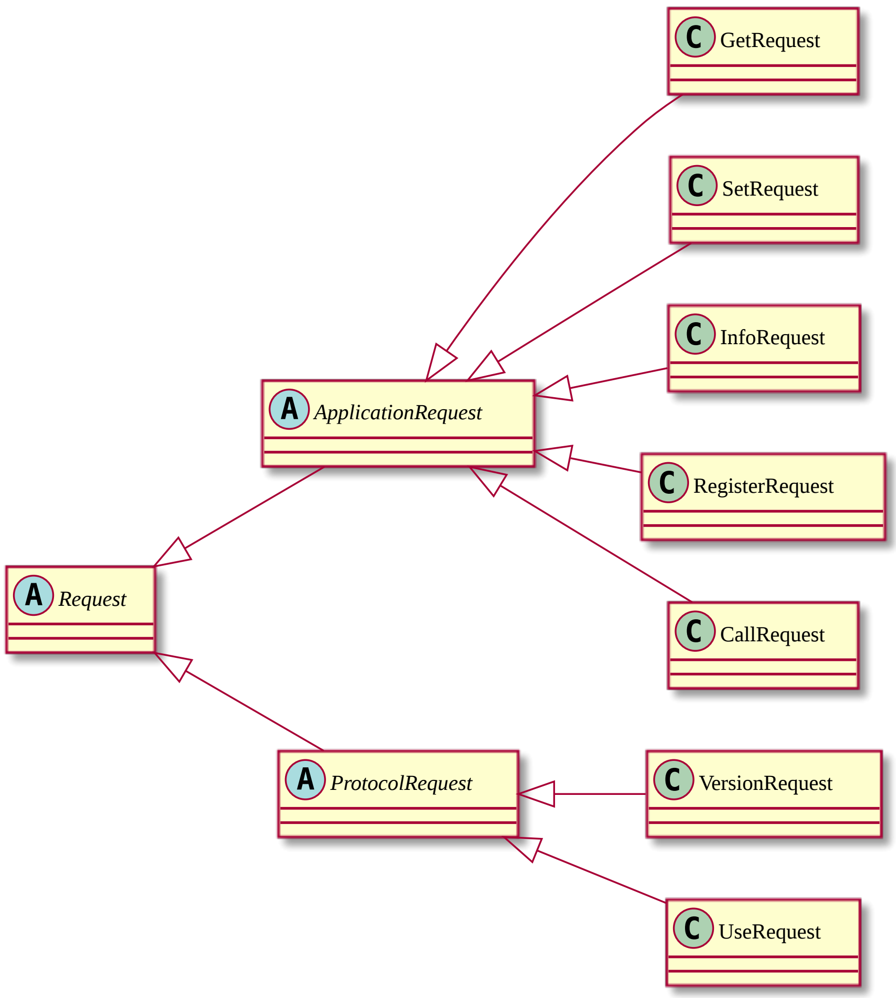

Figure 5.3: Requests class hierarchy.

(the `structure` package presented in following section), but `communication` has no dependency on it (it can thus be used with applications interested only in a specific element in the remote server object tree).

It is worth noting that `communication` package provides not only the client-side implementation, but also a full-featured server-side implementation. Thanks to that, all FJF-compatible entities can be accessed with any generic client designed to connect to native Framsticks servers.

This package may be perceived as root cause of the overall asynchronicity of the FJF, which entails the heavy use of anonymous inner classes as a mean to implement future callbacks. In this place it is worth to note the existence of programming languages targeting specifically communication systems, like `Erlang` [ERL13] or to some degree `Scala` [SCA13].

5.5.1 Connections

Communication package implements Framsticks network protocol in both directions; it allows not only connecting to Framsticks server, but also supports running a Framsticks-compatible server. Each `Connection` instance maintains two separate threads for sending and receiving data through `java.net.Socket`. All operations on both sides are fully asynchronous. At the client side, when issuing a request, a callback is given (typically expressed as an anonymous class), and control is returned immediately. Passed callback is stored by the `Connection`, and will be run upon receiving response from server, with potential result passed in as argument; this callback always extends `ExceptionHandler`, so if a request fails from any reason, information about that failure is also passed to the request issuer. At the server side, entity handling incoming requests may defer sending response as long as needed without blocking communication channel from other requests.

5.5.2 Requests

Classes extending `com.framsticks.communication.Request` represent all requests issued by client. Framsticks Java Framework provide both printing and parsing of those requests, since these classes are used on the server side as well (in the `com.framsticks.hosting` package). Requests are basically divided into two categories: `ProtocolRequests` and `ApplicationRequests` (the complete class hierarchy is presented on figure 5.3). The former is used internally by the `ManagedConnections`, while only the latter can be issued by the connection's owner.

`ClientSideManagedConnection`, used internally by `RemoteTree`, presents a single entry point for request issuing (presented in listing 5.9).

```
class ClientSideManagedConnection {
    public <C> void send(
        final ApplicationRequest request,
        final Dispatcher<C> dispatcher,
        final ClientSideResponseFuture callback
    ) {
        ...
    }
}
```

Listing 5.9: ClientSideManagedConnection

Holding path (for example `/simulator/genepools`) as an ordinary `String`, instead of `Path` object, allows potential usage of `com.framsticks.communication` in isolation from `com.framsticks.structure`, which constitutes an upper layer with respect to the former package. It may also be clearly seen in listing 5.11, which presents a bridge method between the `Communication` package elements (which sees the network part) and the `Structure` package, which understands the tree hierarchy, with the `Files` as an information conveyor understood by both. Mentioned listing also shows implementation of auto-removing the path as a result of exceptional situation's occurrence during the `GetRequest` processing.

```
public final class RemoteTree ... {
    ...
    public void get(final Path path, final FutureHandler<Path> future) {
        final ExceptionHandler remover = pathRemoveHandler(path, future);
        ...
        final Access access = registry.prepareAccess(path.
            getTop().getParam());
        connection.send(
            new GetRequest().path(path.getTextual()),
            AtOnceDispatcher.getInstance(),
            new ClientSideResponseFuture(remover) {
                @Override
                protected void processOk(Response response) {
                    TreeOperations.processFetchedValues(
                        path,
                        response.getFiles(),
                        access,
                        future
                    );
                }
            }
        );
    }
}
```

Listing 5.10: RemoteTree.get() method.

`ApplicationRequest` provides a fluent interface to easily and clearly construct such a request, which can be seen in listing 5.11.

```
public final class RemoteTree ... {
    ...
    @Override
    public void set(final Path path, final PrimitiveParam<?> param,
            final Object value, final FutureHandler<Integer> future) {
        assert isActive();
        final Integer flag = bindAccess(path).set(param, value);

        log.trace("storing value {} for {}", param, path);

        connection.send(
```

```java
            new SetRequest()
                .value(value.toString())
                .field(param.getId())
                .path(path.getTextual()),
            this,
            new ClientSideResponseFuture(future) {
                @Override
                protected void processOk(Response response) {
                    future.pass(flag);
                }
            }
        );
    }
}
```

Listing 5.11: ApplicationRequest fluent interface.

5.6 Structure package

Package `com.framsticks.structure` constitutes a central package of the FJF upper layers, as it contains the `com.framsticks.structure.Tree` interface, which together with `com.framsticks.structure.Path` class provides access to the Framsticks server structure. Currently, there are two implementations of the `Tree` interface, namely `com.framsticks.structure.LocalTree` and `com.framsticks.remote.RemoteTree`. The former provides access to the actual data structure, and is used mainly in `com.framsticks.hosting` package. The latter, although closely related to other entities in `com.framsticks.structure` package, is placed in separate package to stress out, that it is the only entity using functionalities provided by `com.framsticks.communication` package.

The `Tree` interface is designed to be a minimal interface providing all necessary operations. All common operations, that may be seen as external to the `Tree`, are grouped in `TreeOperations`, which is another example of the approach mentioned earlier: separation of data and algorithms.

The whole package strives not to double the underlying data structure. `com.framsticks.structure.Node` is not meant to be used as a building block of a tree structure shadowing the actual data structure, but merely as part of `com.framsticks.structure.Path` class, which may be considered as a snapshot of the state of specific tree path. `Path` also serves as an optimisation in the area of `Tree` operations, since the actual data structure does not need to be traversed each time when given operation is performed. However, this class is not meant to be stored in some permanent way during the FJF execution, which would effectively create a shadowing tree structure, but only to exist for the time needed to perform some specific operation. An important aspect of Path class design is its immutability, all modification operations do not change the object in question, but instead return a modified copy of it. Owing to this feature, Path objects can be safely passed between threads. The process of Path construction will be referred to as „path resolution". Path resolution process starts from the Tree root or from the other Path instance, which holds reference to the Tree internally – this is crucial for the resolution since it is the Tree that contains type information about objects: Registry of Framsclass objects. Because of access to this information, it is also possible during path resolution to construct objects along the way – this ability is used mainly in the RemoteTree, where the structure is incrementally created in response to, for example, user exploring new nodes in the GUI.

5.6.1 Side notes

As stated above, one of the important design aspects behind com.framsticks.structure package is not to create a shadowing structure, but to traverse the actual data structure directly and regardless, whether it is a LocalTree or a structure build as a replica of remote server's structure. Moreover, the Tree structure is built out of objects, like Joint, Genotype for reflected types, PropertiesObject for unknown types or Maps and Lists for list params, none of which have any relation to the enclosing Tree. Presented solution has several advantages, including clear design and lack of dependency enforcement, still it has one important drawback of inability to store meta data concerning tree nodes.

Example of such meta data include:

- flags marking whether object was fully fetched from the remote peer (get request with no fields specified),
- uncommitted changes in GUI,
- history of occurred events.

This issue is addressed with introduction of special data stash, which is maintained alongside the actual tree structure. This pieces of information will be referred to as side notes, and are accessed with means of special keys of generic type com.framsticks.structure.SideNoteKey<T>, which is parametrised by the type of value associated with that key.

Side notes are also a good example of data and algorithm separation (minimal interface notion) and an interesting use case for generics – because of those aspects it will be discussed more thoroughly.

Tree interface defines 3 methods for side node manipulation:

```
public interface Tree extends ... {
    ...
    public <T> void putSideNote(Object object, SideNoteKey<T> key, T value);
```

```
    public <T> T getSideNote(Object object, SideNoteKey<T> key);
    public boolean removeSideNote(Object object, SideNoteKey<?> key);
    ...
}
```

Listing 5.12: Side notes interface in Tree.

First argument for all these methods is an object that is a part of the tree structure. As it is clearly noticeable, no requirements for the type of node are given.

The `AbstractTree` implementation maintains a two-level map, with identity and weak reference key semantics on both levels, with first level being the `object` and the second the `key`; values stored in that structure are automatically removed when either `object` or `key` become unreachable.

The parametrisation of `SideNoteKey<T>` gives important functionalities:

- it makes the interface type-safe in compilation time,
- reduces verbosity of the code (no casting is needed)
- allows to automatically create the side note value, if it is missing.

The former is possible because of parametrisation of the interface (presented in listing 5.12), the latter is possible because `SideNoteKey<T>` instance stores `Class<T>` class during construction time.

5.7 Remote package

The main functionality provided by the `remote` package is the ability to build and maintain a representation of the remote Framsticks or FJF server tree structure. This package builds upon primarily two FJF packages: `structure` and `communication`.

Mentioned functionality is enclosed in the `RemoteTree` class implementing the `Tree` interface.

There are few subtle issues handled by the `RemoteTree` implementation which presented below.

5.7.1 Handshake

Figure 5.4 presents the sequence diagram of connecting to the remote Framsticks server. It can be seen that all user requests (e.g. experiment specific requests) can be issued right after creation of the `RemoteTree` representation. They are buffered until the network protocol settings are established and the most basic information about the remote site is transmitted back to the client. This way the whole process is completely transparent to the user.

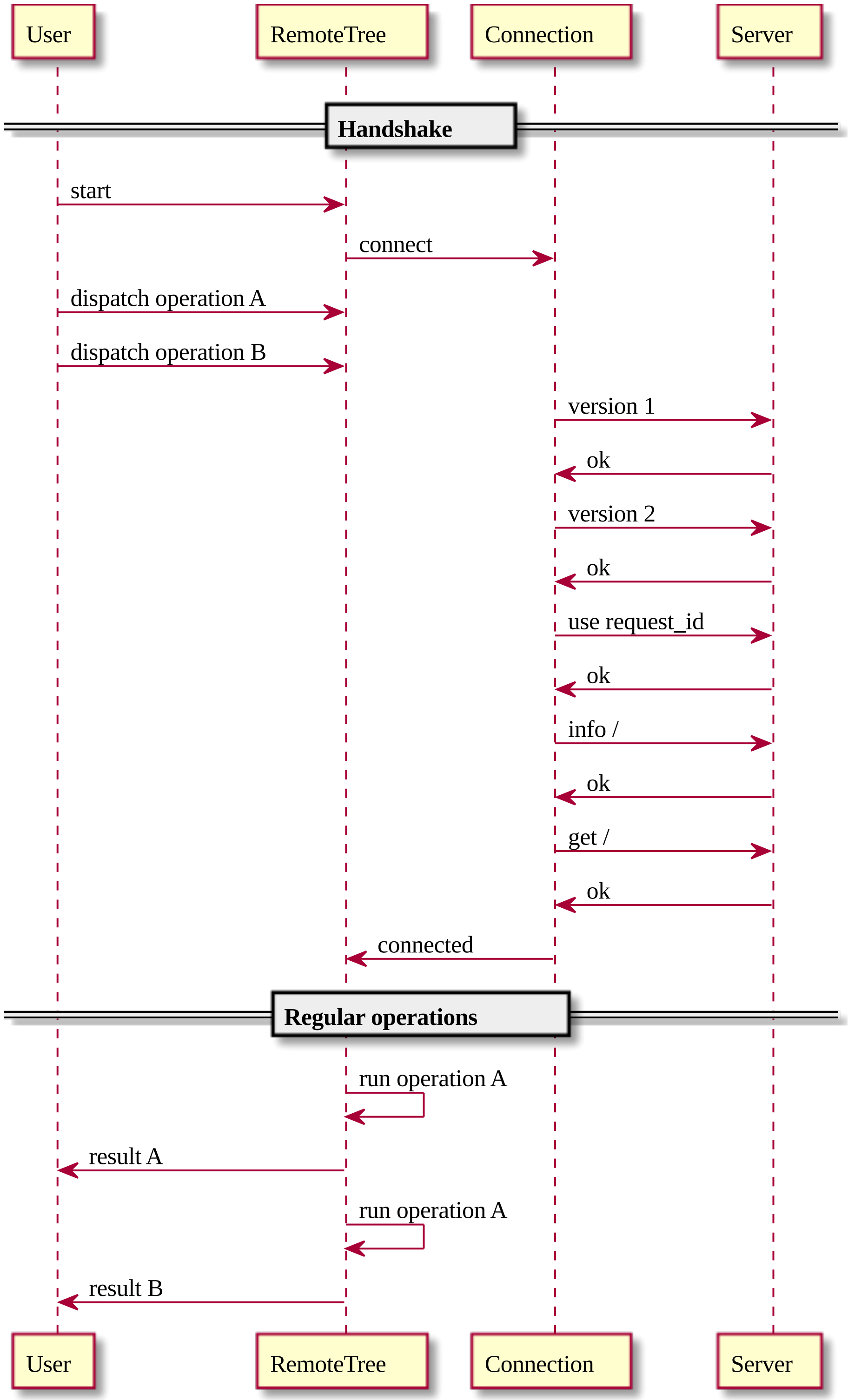

Figure 5.4: Handshake sequence.

5.7.2 Response processing

Figure 5.5 presents the processing scheme of the network protocol response.

First, user calls the method `get`, defined in the `Tree` interface. It is important to notice, that the calling site does not need no know the actual implementation of the `Tree` being used.

Next, the `RemoteTree` encodes that request into the `GetRequest` object, which is passed to the `Connection` to be sent (which is is being executed in a separate thread). That `Request` object is then buffered, and sent after all previously dispatched requests are also sent.

Subsequently, the remote server (be it the Framsticks or FJF server) sends the response encoded in the Framsticks file format. It is parsed to the intermediary representation of the `PropertiesObject`, even if the final target object uses different storage representation. That representation is passed to the `RemoteTree`, where all its attributes are rewritten to the actual representation of the remote object, which is created placed in the maintained tree structure.

Finally, the callback passed originally with the `get` method invocation is being executed.

The presented intermediary step of `PropertiesObject` is necessary, because the parsing of the `file` content takes place in the `Connections` receiver thread, from where it is not possible to safely access the target object.

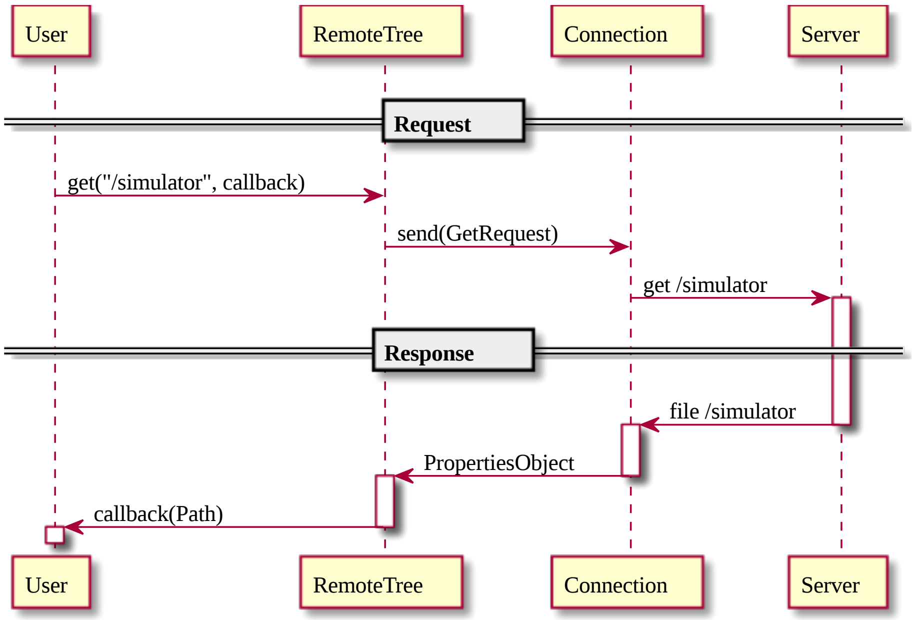

Figure 5.5: Get request processing sequence.

5.8 Hosting package

In the first versions of Framsticks Java Framework, part responsible for communication provided only the ability to connect as a client to the native Framsticks server. This allowed to develop a GUI client (described in the 5.9), and have made possible to propose the solution of an entity that would be able to conduct a distributed evolution experiment by means of controlling multiple Framsticks servers, for which it would be visible as a client. Approach presented above, albeit providing basic experimentation facilities, would not provide any feasible way of tracking experiment progress; user conducting the experiment could only connect to the separate Framsticks servers and track their individual progresses, still it would not provide a holistic view of the experiment. Because of the requirement justified above, the support was introduced for hosting Framsticks Java Framework entities in a Framsticks-compatible server. In fact, this server-side support constitutes a one-to-one complementary to the client-side solution and consists of two main parts:

- a server-side connection (`ServerSideManagedConnection`), sharing much of its implementation with `ClientSideManagedConnection`,
- a `LocalTree` being a counterpart to `RemoteTree`, extensively using `ReflectionAccess` and related utilities to provide the client with full information about hosted structure.

The only requirement for **Java** class to be available through over Framsticks network protocol, is to be annotated with `FramsClassAnnotation` and to have members with `ParamAnnotations`. In typical situation, the majority of classes comprising the experiment structure is already annotated for the sake of configuration parsing; it is another example of the strength of data and algorithm separation notion.

The hosting infrastructure (`com.framsticks.hosting.Server` and supporting classes) is completely external in regard to the hosted entity, which is intuitively reflected in the FJF configuration presented in listing 5.13. Hosting functionality, providing network connectivity to the experiment instances, can be thus seen as an application container very similar to the web application containers, like `Apache Tomcat` [APT13].

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Framsticks>
    <import class="com.framsticks.hosting.Server" />
    <import class="com.framsticks.test.PrimeExperiment" />
    <import class="com.framsticks.experiment.SimulatorConnector" />
    <import class="com.framsticks.structure.LocalTree" />
    <Server port="9007">
        <LocalTree name="prime-local-tree">
            <PrimeExperiment maxNumber="3000">
                <SimulatorConnector address="localhost:9100" />
                <SimulatorConnector address="localhost:9101" />
                <SimulatorConnector address="localhost:9102" />
            </PrimeExperiment>
        </LocalTree>
    </Server>
</Framsticks>
```

Listing 5.13: Hosting.

The `Server` class accesses the hosted structure through means of `LocalTree`. Once started, `Server` listens on a configured port for connections over Framsticks network protocol. For every accepted connection an instance of `com.framsticks.hosting.ClientAtServer` class is created, which encapsulates an instance of `com.framsticks.hosting.Cli` separate for each client, thus closely mimicking the original Framsticks behaviour.

Thanks to solution presented above, there is in fact no difference between the original Framsticks server and the FJF server, as long as `RemoteTree` and GUI is interested. The difference may be noticeable by the user mainly because of differences in naming conventions between Framsticks and FJF, the latter following standard **Java** naming conventions. Described solution presents the user with a GUI access to both running experiment controller and to the working Framsticks servers in a unified manner, thus allowing to monitor the experiment run and to track possible problems in all system elements.

5.8.1 Listeners wrapping

Notion of events constitute an important part of original Framsticks solution. The Framsticks Java Framework presents an interface typical in the **Java** programming language, namely `EventListener` interface. In the FJF events are visible as `EventParam` attributes of the `FramsClass`; as it may be seen in figure 5.1 they are siblings to `ValueParam`) and `ProcedureParam` types.

An entity wishing to register on all events published by a Framsticks object may simply filter out all `EventParam` instances, and use routines presented in 5.14.

```
public interface Tree {
    ...
    public <A> void addListener(Path path, EventParam param,
        EventListener<A> listener, Class<A> argumentType,
        FutureHandler<Void> future);

    public void removeListener(Path path, EventParam param,
        EventListener<?> listener,
        FutureHandler<Void> future);
}
```

Listing 5.14: Dispatcher.

It is worth noting that presented interface is strongly typed and supports automatic conversion of events' arguments, where requested type of the argument is denoted as `A` and passed in runtime through the `Class<A>` argument. The actual conversion is performed by the `convert()` method (5.15), which executes the following algorithm:

- if both arguments are `File`, do nothing;

- otherwise:
  - if `from` argument is a `File`:
    - if `toJavaClass` is `Object.class`, then try read using registry;
    - otherwise: try to use `loadComposites`;
  - if `to` argument is a `File`: use `Registry` to `saveAll`
  - fail otherwise.

```
public final class AccessOperations {
    ...
    public static <T, F> T convert(Class<T> toJavaClass,
        F from, Registry registry) {
        ...
    }
}
```

Listing 5.15: AccessOperations.convert.

Conversely, for the sake of full hosting support, events published in **Java** types are discovered by **Java** type to Framsticks type conversion utilities, thus making them visible and accessible through the Framsticks network protocol.

In conclusion, the events consumer is completely separated from the events' producer, which can be one of the following:

- a **Java** object, accessible through the `LocalTree`),
- a native Framsticks server, accessible through the `RemoteTree`),
- a remote **Java** object accessible through the `RemoteTree` on the client side and through `LocalTree` hosted in `Server` on the server side.

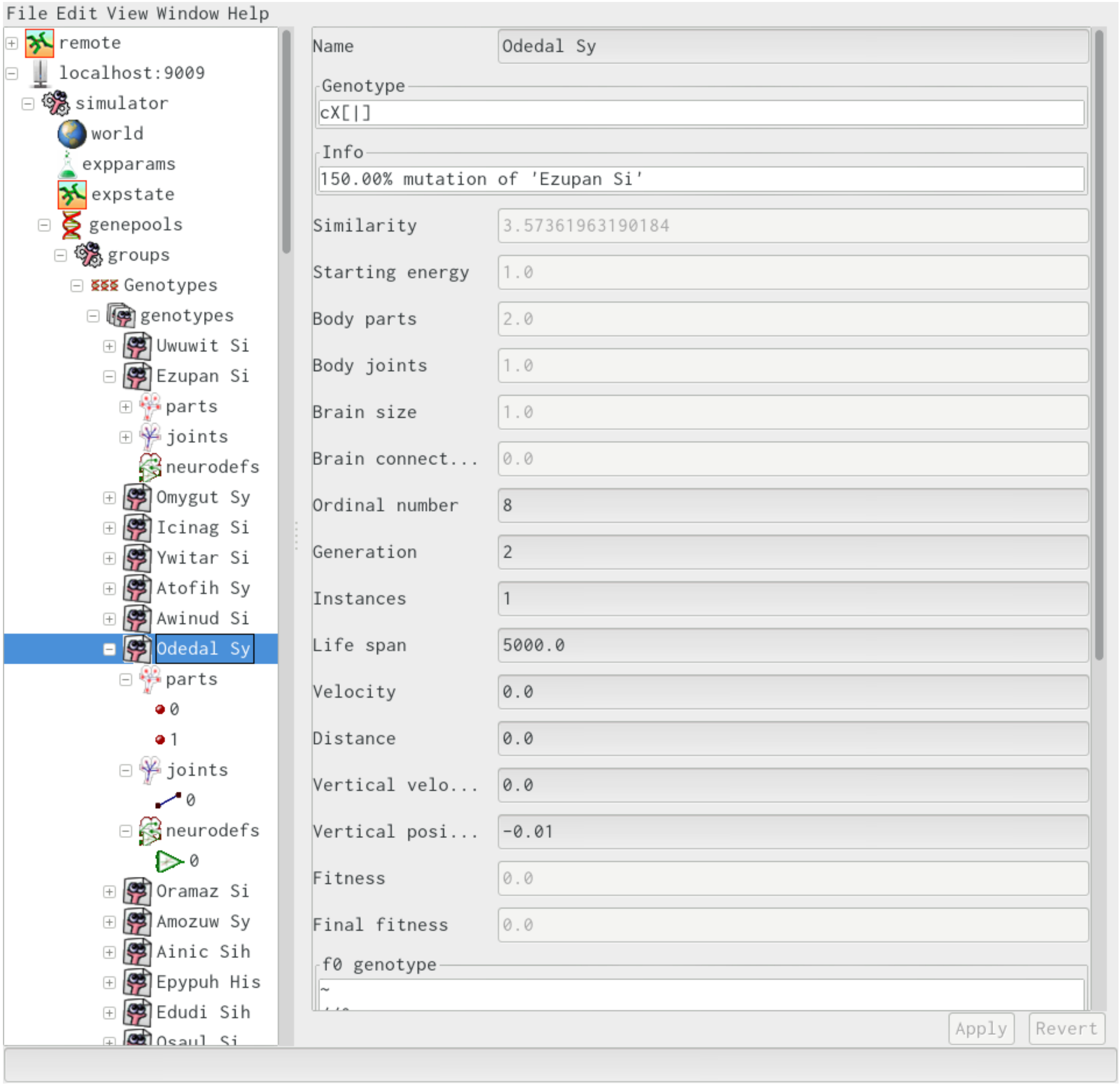

Figure 5.6: A genotype instance presented in GUI.

5.9 Graphical User Interface

Graphical User Interface developed for Framsticks Java Framework constitutes of potentially several `com.framsticks.gui.Frame` instances, each allowing to access potentially several `Tree` instances, both local and remote. Each `Frame` presents user with a tree GUI component, which presents all attached `Tree` instances. The nodes can be dynamically resolved, which in case of `RemoteTree` results in network requests being issued. Currently chosen tree node is viewable on the right side of the `Frame`, in a special `Panel` instance prepared for this specific object's Framsticks type. Figure 5.6 presents a typical view of the **GUI** provided in FJF.

5.9.1 Panels

Each `Frame` maintains a cache of `Panel` instances for each `com.framsticks.core.Tree` instance. The cache is accessed with string value representing Framsticks type, like `o Simulator` or `l Genotype uid`. Usage of such a cache is crucial for the overall responsiveness of the user interface, since it makes the most time-consuming construction logics a one-time effort, leaving the operation of filling up control values the only operation to be done each time, when the currently viewed object is changed.

The process of creating panel for a specific Framsticks type is prepared to be a customisation point of **GUI** solution. `Browser` holds a set of `PanelProvider` instances, which may be attached in configuration or programmatically.

Framsticks Java Framework includes two standard `PanelProvider` classes:
`ObjectPanelProvider` and `ListPanelProvider`.

If for a given Framsticks type more than one `PanelProvider` decides to provide a `Panel` instance, all those `Panel`s are automatically wrapped in a `MultiPanel`, where they are accessible through tabs.

5.9.2 Tree

`JTree` component constitutes the central element of the browser `Frame`. Architecture of **Swing** allowed not to double the `Tree` structure by maintaining a separate tree structure for means of **GUI**. **Swing** does not impose any constraints on the type, that will serve to represent `JTree` nodes – it is just an `Object`. All logics of the tree has to be provided by a class implementing `javax.swing.tree.TreeModel` interface, which is presented in its entirety in listing 5.16.

```
package javax.swing.tree;

public interface TreeModel {
    Object getRoot();
    Object getChild(Object parent, int number);
```

```java
    int getChildCount(Object parent);
    boolean isLeaf(Object node);
    void valueForPathChanged(TreePath treePath, Object value);
    int getIndexOfChild(Object parent, Object child);
    void addTreeModelListener(TreeModelListener listener);
    void removeTreeModelListener(TreeModelListener listener);
}
```

Listing 5.16: TreeModel interface.

Documentation of `TreeModel` states that all arguments of type `Object` passed to the `TreeModel` by `JTree` are always those previously returned by `getRoot` or `getChild`, with regard to `equals()` and `hashCode()`. It might seem that objects like `com.framsticks.model.Joint` or `com.framsticks.params.UniqueList` building up `Tree` could be used here directly. This approach is not valid because of the following constraints and goals:

- the `TreeModel` needs to provide interface to several `Tree` instances and this information in general is not available in these objects;
- `JTree` uses `equals` logics for nodes distinction, which might be already used by these classes to implement semantics not compatible with `JTree` requirements;
- foreseen support for user favourites, which would result in the need of presenting the same object under several nodes in the same `TreeModel`, which would corrupt the `JTree`.

Because of the issues presented above, a special thin-wrapper around actual `Tree` objects was introduced, namely the `com.framsticks.gui.treee.TreeNode`. `TreeNode` holds a `WeakReference` to the actual `Tree` object and implements `equal()` taking into account not only the held object itself but also the mount point (that would be different in user favourites sub-trees). `TreeNode` also holds `CompositeParam` describing Framsticks type of the object being hold and caches reference to appropriate `Panel`. It is important to note that `TreeNode` does not store `com.framsticks.structure.Path` instance (for the reasons described in 5.6), but only its textual representation.

Tabular view Very similar to the `TreeModel` is the `TableModel` interface successfully utilized for `ListPanel` implementation, which is used to present in **GUI** objects of type `ListParam`. Because of lack of the presented above non-trivial tree structure aspect, implementation of `TableModel` interface was much more straightforward than the implementation of `TreeModel`. Figure 5.7 presents an example of a tabular view.

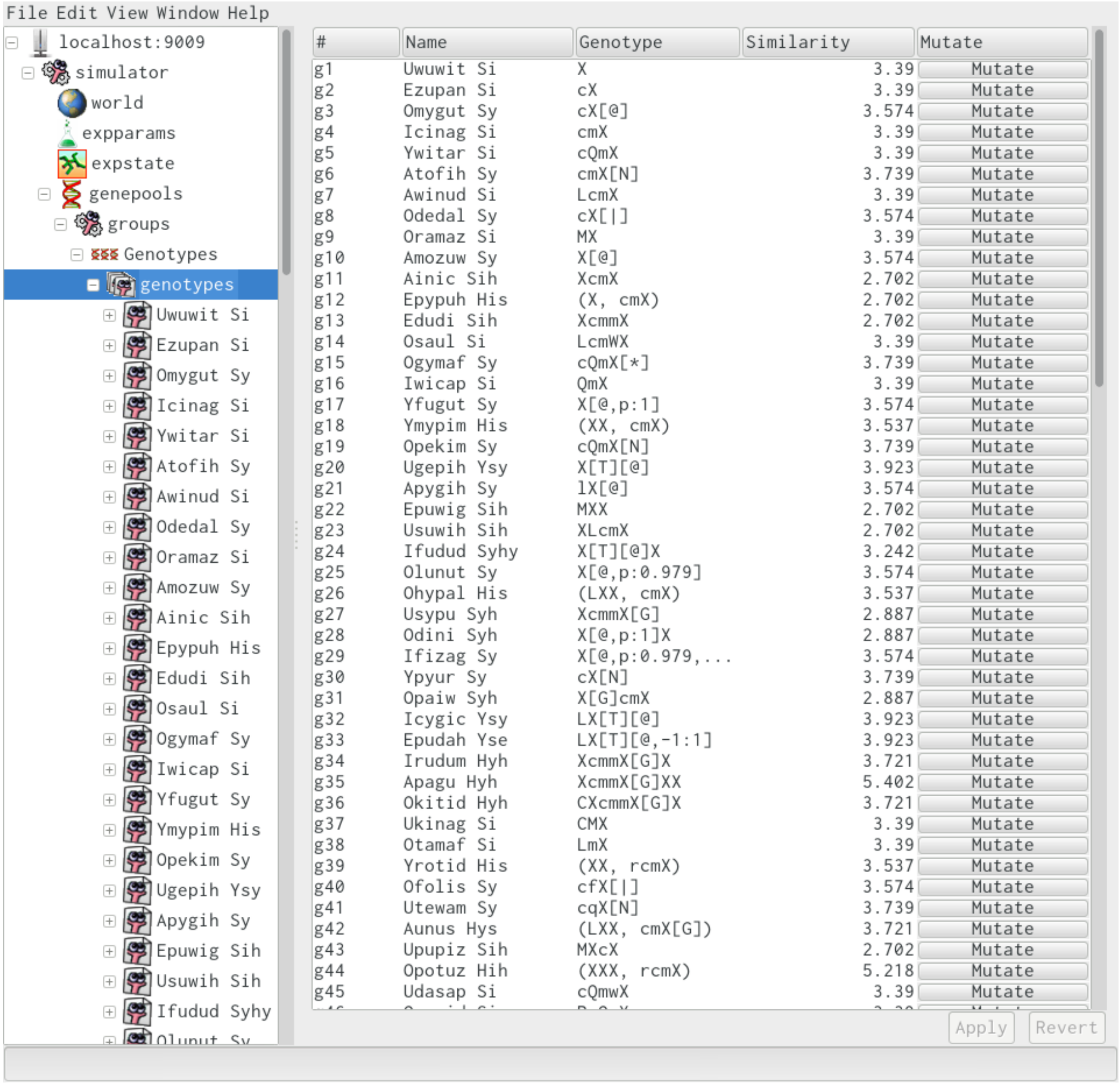

Figure 5.7: A list of genotypes visible in the GUI.

5.9.3 Framsticks server event utilization

Framsticks server events are utilised in Framsticks Java Framework doubly. First, user interface uses adheres to the convention of `Param` naming, i.e. for each `param` of type `ValueParam` identified as `some_param` it automatically registers for event identified as `some_param_changed`. If the `Param` is of type `ListParam`, it is expected that event will contain argument of type `ListChange`, which is then interpreted accordingly by adding/removing/updating object in `Tree` and notifying **Swing** about changes occurred in underlying tree structure, that need to be rendered in the **GUI** component. In case of `PrimitiveParam`, it is expected that event will contain `ValueChange` argument, which will be used to update the maintained `Tree` (otherwise this `param` will be explicitly reloaded) and refill **GUI** control, if the object is the currently viewed one.

For every `EventParam` (if it is not marked with `USER_HIDDEN` flag) `ObjectPanelProvider` adds to the constructed `ObjectPanel` an `EventControl` component, which presents user with possibility of manually registering for an event. Received events are available in the list, that is a part of `EventControl`.

5.9.4 Consoles

User is also given a possibility of accessing remote Framsticks servers (both native and FJF ones) through means of a `Console` frame. There are three types of consoles available:

- `TrackingConsole`,
- `RawConsole`,
- `ManagedConsole`.

The main part of all console frames is made of a single text area containing communication messages. First one is meant for debugging or educational purposes, as it merely shows communication between **GUI** and the remote server (user may filter directions of communications). Through `TrackingConsole` user may check what requests FJF **GUI** issues in response to various user interactions.

Both remaining consoles extend `InteractiveConsole`, which allows to manually write requests to the remote server and supports history of sent commands and tab-completion. In case of `RawConsole` tab-completion is based on the history, whereas in `ManagedConsole` it is based on the actual tree structure of the remote server (during completion it automatically issues `info` and `get` requests).

5.9.5 Hosting

It is worth to note that **GUI** infrastructure is also able to directly access a `LocalTree` instance, which allows to run experiment directly in the **GUI**. This approach constitutes an alternative to the much more sophisticated scenario presented in the 5.8 section (of which **GUI** is also a part). The difference between those configurations is similar in the nature to the difference between running program directly in the terminal emulator and running in it inside of `screen` utility.

5.10 Model

Package `com.framsticks.model` contains various classes reflecting those present in native Framsticks simulator. They are primarily used to build `RemoteTree` representing native Framsticks simulators, where they are used through `ReflectionAccess`. They could also be used in completely standalone fashion, since this package is considered a lower-layer package and has no dependencies on `Structure` package (as it can be seen in 4.1).

Usage of specially crafted classes being accessed through `ReflectionAccess`, instead of generic `PropertiesObject` used in conjunction with `PropertiesAccess`, is dictated by one of the project goals, i.e. to provide an easy to use, compile-time checked environment for interpretation and manipulation of genotypes and creatures, in order to ease experiment creation. Another important aspect is to simplify future visualization solutions, which would not have to use `Access` facilities.

Presented package makes an extensive use of annotations: `@FramsClassAnnotation` and `@ParamAnnotation` (described in section 5.2.1). They are used to associate specific **Java** classes and their fields with Framsticks classes. In many situations `@ParamAnnotation`'s `id` attribute is used to clearly associate Framsticks to **Java** naming conventions; such an approach is far superior to the one using comments due to it's robustness for future changes (it is presented in listing 5.17).

Several classes found in the `com.framsticks.model` provide also view to their fields in a more object-oriented friendly way, which is exemplified in listing 5.17 with `getRotation` and `setRotation` routines.

```
@FramsClassAnnotation(id = "p")
public class Part extends BasePart implements ModelComponent {

    @ParamAnnotation(id = "rx")
    public double rotationX;
    @ParamAnnotation(id = "ry")
    public double rotationY;
    @ParamAnnotation(id = "rz")
    public double rotationZ;

    public Point3d getRotation() {
        return new Point3d(rotationX, rotationY, rotationZ);
    }
    public void setRotation(Point3d r) {
        rotationX = r.x;
        rotationY = r.y;
        rotationZ = r.z;
    }

    @ParamAnnotation(id = "dn")
    public double density;
```

```
    @ParamAnnotation(id = "ing")
    public double ingestion;
    @ParamAnnotation(id = "as")
    public double assimilation;
    @ParamAnnotation(id = "i")
    public String info;
    @ParamAnnotation(id = "Vstyle")
    public String visualizationStyle;
    @ParamAnnotation(id = "vs")
    public double visualThickness;
    ...
}
```

Listing 5.17: Model Part.

5.10.1 F0 scheme

In the Framsticks system, the scheme of f0 representation is expressed in an `xml` file. Thanks to the generic character of `XmlLoader` described in previous section, it was possible to use the `XmlLoader` to read not only the FJF configuration, but also the f0 representation scheme. The `SchemaBuilder` class only imports needed classes (like `FramsClassBuilder`) into the `XmlLoader`, after which just runs the loader, not interfering with its internal logics (the approach is very similar to the one of `MultiParamLoader`).

5.11 Exceptions

In previous sections describing individual Framsticks Java Framework packages, it was made apparent, that FJF as a whole is an inherently asynchronous environment (with main causes presented in section 2). Although usage of anonymous classes (described in 3.5) proved to be a clear and robust way to express a processing path distributed along both time and space (being called from different threads) in a compact piece of code, one important issue remained unsolved, namely the exception handling.

Introduction of exception notion in modern programming languages proved to be a very important improvement. In a typical situation, thrown exception is passed to the first `catch` statement, with the stack above that statement being unwound. In the case of an exception being thrown from body of an asynchronous callback (possibly expressed using anonymous class, but it is not required in this context), if it is not caught explicitly using `try/catch` inside that body, it will propagate to the calling environment, which typically is not the one appropriate to handle arisen exceptional situation represented by the exception.

5.11.1 FramsticksException

A standard exception class found in **JDK** provides some short description and information regarding stack state at the moment exception was thrown. Again, in most situations, this information only seemingly allows the developer or end user to understand the state of environment when the exceptional situation occurred. What is missing is the context, i.e. values of some variables crucial to understand when the situation actually occurs – the difference being between exception saying about read file failure, and the exception conveying also file's name.

If the exception arguments are not to be utilised by the application code itself (for example to dispose some problematic resource), it is not actually needed for those arguments to be stored in a structured way, i.e. by exception class' fields.

Issues presented above are standing behind the shape of `FramsticksException` class. `FramsticksException` serves as a root in the FJF hierarchy of exception classes. It presents a fluent interface, enabling an easy construction enclosing message header, optional cause, and an arbitrary number of context arguments. A typical use case is presented in listing 5.18.

```
try {
    ...
} catch (ClassCastException e) {
    throw new FramsticksException().msg("failed to cast").cause(e)
        .arg("param", param)
        .arg("actual", value.getClass())
        .arg("requested", type);
}
```

Listing 5.18: Throwing FramsticksException.

The second argument to the `FramsticksException.arg()` method can be a value of any **Java** type, which provides a concise `toString()` implementation. All stored arguments are converted to string when the construction of human-readable message is requested from the `FramsticksException` object. This typically happens during logging, but is also used to fill status line in GUI or to provide a comment to the `error` response in Framsticks network protocol (what is will be presented more closely at the end of this section). Adopted approach has one more advantage, namely a trivial implementation of exception classes extending `FramticksException`: just the `extends` clause with an empty class body.

5.11.2 ExceptionHandler

Drawing from the exception notion itself, Framsticks Java Framework introduces a notion of `ExceptionHandler` that is passed behind the scenes, much like the exceptions processing path is expressed behind the main application logic.

The regular **Java** exception handling scheme together with introduced `ExceptionHandler` will be referred to as asynchronous exception path.

5.11.3 Future and FutureHandler

`ExceptionHandler` is used extensively with `Future` and `FutureHandler` generic abstract classes. The main idea behind those classes is to provide a concise way of expressing future result value processing as well as to allow hiding of exception processing path. `FutureHandler` defines an interface allowing passing result of computation (network response in particular) through `pass` method, which is internally handled by overloaded `result()` method. If any exception is thrown during result processing, it is handled by that class itself – it implements `ExceptionHandler` interface, but leaves the actual implementation of `handle()` method to the user. Argument of type FutureHandler<T> is typically passed as the last argument of asynchronous methods that would, if synchronous, return value of type T.

`Future` class extends `FutureHandler` by proxyfing the exception processing path to other `ExceptionHandler`, which is passed at construction time.

```
public abstract class FutureHandler<T> implements ExceptionHandler {

    protected abstract void result(T result);

    public final void pass(T result) {
        try {
            result(result);
        } catch (FramsticksException e) {
            handle(e);
        }
    }
}
```

Listing 5.19: FutureHandler.

```
public abstract class Future<T> extends FutureHandler<T> {

    protected final ExceptionHandler handler;

    public Future(ExceptionHandler handler) {
        assert handler != null;
        this.handler = handler;
```

```
    }

    @Override
    public final void handle(FramsticksException exception) {
        handler.handle(exception);
    }
}
```

Listing 5.20: Future.

Listing 5.22 presents a real-world yet clear example of those classes usage (with comparison of hypothetical synchronous case in 5.21). Here, only the logging operation is executed before returning result to the outer `FutureHandler`, but in other situations more complex operations could be performed here, possibly changing the type of result being passed along. The synchronous example shows a situation when no exceptions are to be specially handled by the `netsave()` routine, so no `try/catch` construct is needed here. Presented approach also allows not to include it explicitly in the asynchronous case, since the `try/catch` block is already provided in `FutureHandler` class. Hence the initial goal of hiding the exception processing path and not cluttering the main processing path, as it is easily achievable in synchronous case, is also possible in asynchronous case.

```
public <N> N netsave(Class<N> netJavaClass) {
    N net = call(simulatorPath,
        getParam(simulatorPath, "netsave", ProcedureParam.class),
        arguments(), netJavaClass
    );
    log.debug("netsave of {} done", net);
    return net;
}
```

Listing 5.21: Synchronous exception handling.

```
public <N> void netsave(Class<N> netJavaClass,
        final FutureHandler<N> futureNet) {
    call(simulatorPath,
        getParam(simulatorPath, "netsave", ProcedureParam.class),
        arguments(), netJavaClass,
        new Future<N>(futureNet) {

            @Override
            protected void result(N net) {
                log.debug("netsave of {} done", net);
```

```
                futureNet.pass(net);
            }
        }
    );
}
```

Listing 5.22: Asynchronous exception handling in FJF.

5.11.4 Framsticks network protocol integration

It is worth mentioning that also `error` responses to Framsticks network protocol requests are being passed along the presented asynchronous exception, on both sides of communication channel. In the FJF server implementation, if an exception is thrown during the request processing, it is caught and the its description is sent to the client as a comment to the `error` response. At the client side, if a request results in an `error` response (irrespectively from whether it is the native Framsticks server or FJF server on the other side of communication channel), an exception object is constructed using that error comment, and it is then passed to the request callback; the implementation is distinctively compact and is presented in listing 5.23). This way, to some degree, FJF supports passing exceptions not only between threads in an asynchronous environment, but also between processes running on distinct machines.

```
public abstract class ClientSideResponseFuture extends Future<Response> {
    ...
    protected abstract void processOk(Response response);

    @Override
    protected final void result(Response response) {
        if (response.getOk()) {
            processOk(response);
        } else {
            handle(new FramsticksException()
                .msg("invalid response")
                .arg("comment", response.getComment())
                .arg("request", request));
        }
    }
}
```

Listing 5.23: Processing request's failure.

5.12 Problems

5.12.1 The difference between Void.TYPE and Void.class

`java.lang.Void` is an important part of **Java** type system. In section 5.11 class `Future<T>` was shown, that is used to wrap computations to be performed on result available asynchronously. In situations when there is no appropriate return value to be passed, and the `Future` is used just to express the ordering of operations, `Void` type is used as the generic argument of `Future` (listing 5.24 shows such an example). Existence of `Void` allows not to prepare special `Future`-like class allowing to pass result of a function not returning any value (being of type `void`), that would force to double also other associated classes, like `FutureHandler`.

```
addListener(path,
    getParam(),
    newListener,
    Object.class,
    new Future<Void>(owner.getFrame()) {

        @Override
        protected void result(Void result) {
            putSideNote(path, listenerKey, newListener);
            refreshState();
        }
    }
);
```

Listing 5.24: Listener registration in EventControl.

All primitive types in **Java**, like `int` or `float`, have accompanying boxing types: `Integer`, `Float`. In the reflection layer of **Java** boxing type `Integer` is represented by singleton `Integer.class`, whereas primitive type `int` is represented by singleton `Integer.TYPE` – only boxing types have a static field `TYPE`. Also `Void` type has both `class` and `TYPE` fields, but the difference between them is more subtle due to specificity of `void`. Value of type `Void.class` has only one possible value: `null`, whereas `Void.TYPE` has no valid values at all. The former is found in reflection to represent formal argument type of methods like the one presented in listing 5.24, while the latter represents type of value returned by methods of type `void`. That difference, although quite intuitive once known, may be very confusing and is not clearly stated in the documentation of `Void` type found in official **Java** documentation [JAV13].

Chapter 6

Computational experiments

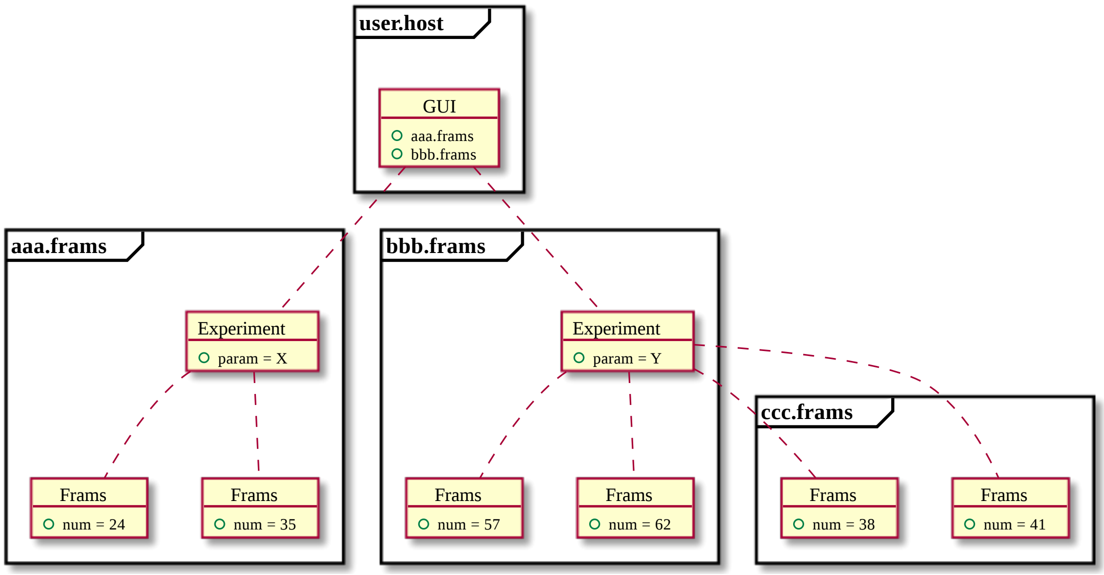

Figure 6.1: Example of distributed experiment configuration

This chapter will present the developed infrastructure for distributed experiments on artificial evolution in the Framsticks system. The experiment is divided logically into two parts: computational part – executed in Framsticks servers and expressed in `Framscript` – and controlling part, hosted in FJF, with experiments' logics expressed in **Java**.

The experimentation framework provides possibility of using multiple Framsticks servers as computational nodes running on remote hosts. The controlling node can be directly embedded in the **GUI** instance or be executed remotely, in background. The former case can be used for short-running experiments or during development of new experiment, while the latter case is designed for long-running experiments. In this scenario, user can access the controlling server by attaching a **GUI** to the remotely running server, and after checking its state or manipulating the experiment flow, user can again detach from the server. In all cases, user can attach **GUI** to the running computational nodes directly – this possibility might be used for debugging purposes. Controlling server is able to use instances of Framsticks working servers that are already running or to spawn new instances. FJF also allows user to connect to several running experiments from a single **GUI**– an example of such a scenario is presented in the 6.1, where user connects with experiment instances configured with different parameters and uses different number of Framsticks computational servers.

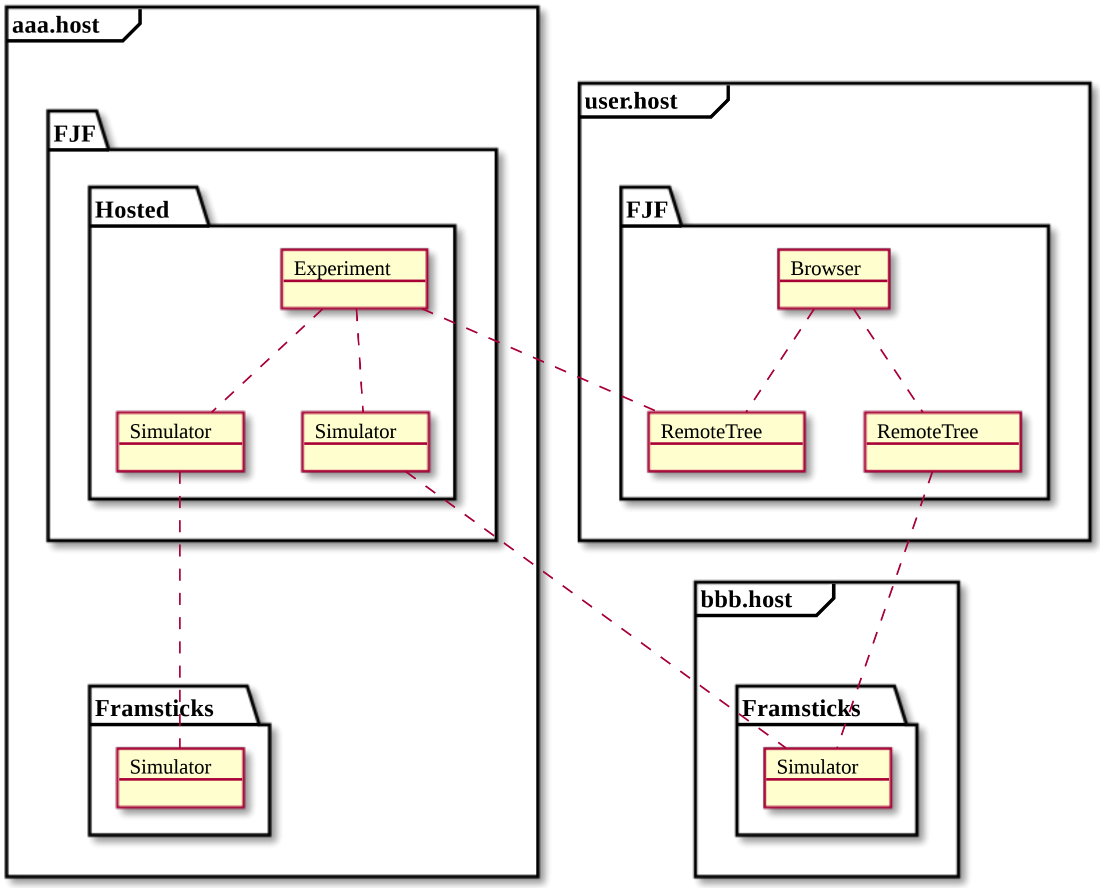

Figure 6.2: Example of experiment internal hierarchy.

6.1 Running infrastructure

In this section a generic infrastructure for defining and running experiments will be presented (specific examples will be discussed in the following section). All major classes providing functionalities described in this section are placed in `com.framsticks.experiment` package.

The experiment in FJF is expressed through a hierarchic structure of entities possibly distributed among several hosts (shown on figure 6.2). Instance of `Experiment` class represents the root of this tree, enclosing all experiment logics through various instances of classes extending `ExperimentLogic`, which are bound together to express the experiment run (they will be discussed in the next section). The computational nodes are represented as instances of `Simulator` class; each of them internally holds a single instance of

RemoteTree class representing the actual Framsticks server. Figure 6.2 also presents the possibility of direct attachment of **GUI** to the computational node.

Infrastructure presented up to this point leaves out one important issue, namely starting and maintaining the computational nodes – for this purpose several classes were devised. They are briefly described below with their hierarchy presented in figure 6.3.

SimulatorProvider is an interface allowing to request for a new `Simulator` instance in asynchronous way.

SimulatorConnector is the simplest provider, which allows to connect to a single already running Framsticks server instance which should be available under preconfigured address. After successful connection to the remote server, but before returning new `Simulator` instance to the requesting user (an `Experiment` instance), `SimulatorConnector` automatically tries to resolve the `/simulator` path and checks whether proper `expdef` is loaded.

SimulatorRunner is similar to the `SimulatorConnector`, but it starts the Framsticks server on its own, possibly on a remote host (using `SSH`). The user is responsible for setting up `SSH` keys on both communication ends.

SimulatorRange is a composite provider which is able to provide multiple simulators running on multiple hosts, using internally `SimulatorConnector` or `SimulatorRunner` (depending on configuration).

SimulatorGroup is a composite provider using internally any other `SimulatorProviders`.

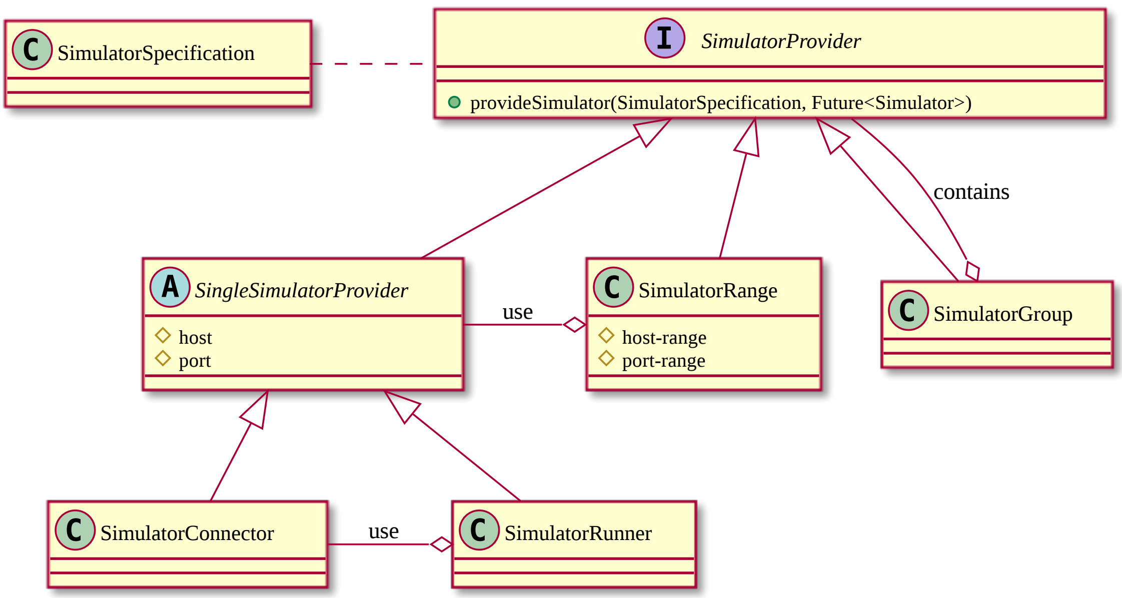

Figure 6.3: Hierarchy of simulator providers.

Typically, an `Experiment` instance is configured to use a composition of presented `SimulatorProvider`s, which are then called to provide new `Simulator` instances as needed, i.e. up to a configured amount or as a response for a manual user request (available from **GUI**). It is important to note that `SimulatorProvider`s hierarchy is an open class hierarchy (in opposition to for example `Param`s hierarchy), which means that custom `SimulatorProvider`s implementing different policies can be easily added and utilised.

6.2 Experiment definition

The final element of the experimentation infrastructure is the `ExperimentLogic`s subsystem. An important design aspect of the FJF is to facilitate definition of new experiments with the least effort possible, however with an assumption that experimenter has a basic **Java** understanding. In order to fulfil those goals a notion of `ExperimentLogic` has been devised and implemented. The `ExperimentLogic` is an abstract block of logic, that is designed to work in the environment provided by the `Experiment` and provides only a single, specific functionality. A typical `ExperimentLogic` can be described with following properties:

- provides extensions points (in the shape of callback), to which other logics can attach their own functionality;
- automatically registers on specific events of the `Experiment`, attached `Simulator`s or other `ExperimentLogic`s;
- can maintain an internal state: global or local to the specific `Simulator` instance.

The most basic example of an `ExperimentLogic` is the `NetLoadSaveLogic`, which encapsulates `/simulator/netload` and `/simulator/netsave` procedures and presents a pure **Java** interface to that functionality. Although fundamental to most experiments, it needs to be stressed out that `NetLoadSaveLogic` is not obligatory to be used at all, if only given experiment takes a different approach to the communication with controlling server.

Other concrete implementations of `ExperimentLogic` notion will be discussed in following sections presenting proof-of-concept experiment scenarios.

6.2.1 Prime experiment

Prime experiment is a work-case example, in which a task of finding all prime numbers in given range is exercised. The computational server side implementation is deliberately non-optimal. It was considered valuable to include discussion of this trivial experiment in this chapter, since it presents the same advantage which was exploited during experimentation framework development – it allows to concentrate solely on infrastructure, management and communication issues. This section may be used as a guide to creation of new experiments of arbitrary character.

State representation

In terms of Framsticks file format, the state experiment is expressed in a way presented in listing 6.1, namely using two objects of types `ExpParams` and `ExpState`, where the former holds the task question, while the latter holds the result.

```
ExpParams:
from_number:150
to_number:200

ExpState:
current_number:201
result:@Serialized:[151,157,163,167,173,179,181,191,193,197,199]
```

Listing 6.1: Prime experiment state.

For interoperability with FJF, two short dedicated classes were prepared, annotated with FJF annotations (presented in section 5.2.1), with names matching the ones coming from the computational server (although they could be different, if only properly annotated). They are both enclosed in `PrimePackage` class (presented in listing 6.2)

```
@FramsClassAnnotation(order = {"params", "state"})
public class PrimePackage implements WorkPackage<PrimePackage> {

    @ParamAnnotation
    public final ExpParams params = new ExpParams();

    @ParamAnnotation
    public final ExpState state = new ExpState();

    ...
}
```

Listing 6.2: Prime experiment state.

Annotations used in all three mentioned **Java** classes – together with several FJF generic algorithms working on those annotations – allow the experiment creator not to write any code responsible for experiment state serialisation/deserialisation.

Work package model

The main `ExperimentLogic` used by the `PrimeExperiment`, is the `WorkPackageLogic`, which assumes that the problem domain can be decomposed into a number of independent tasks. In terms of FJF it is expressed in the fact, that the presented `PrimePackage` class implements `WorkPackage` generic interface, which is used by the generic `WorkPackageLogic` (which is parametrised with type of the work package).

An instance of `WorkPackageLogic<PrimePackage>` specialisation is included as a part of `PrimeExperiment` class defining the experiment.

`WorkPackageLogic` internally uses the `NetLoadSaveLogic` to send computation requests and receive results. One of facilities provided by that logic is the ability to track the work packages being sent and resend them, if necessary. If the class implementing `WorkPackage` interface provides an appropriate implementation, it also possible to resend only subdomain of the original package instance, if partial results were received.

6.2.2 Standard experiment

The standard experiment definition is the canonical example of Framsticks experiment. It provides the means to express an arbitrary fitness criterion and simulate evolution of individuals using a single genotype pool.

In the scope of FJF, it is represented by the `StandardExperiment` class, which allows to use multiple native Framsticks servers running standard experiment, and migrate genotypes between them. The `StandardExperiment` does not provide any scientific value on its own, it rather presents a FJF-based approach to the problem.

The state of a single Framsticks server instance is represented by the `StandardState` class, which implements the `NetFile` interface, thus making it compatible with the `NetLoadSaveLogic`. It is presented in its entirety in listing 6.3.

```
@FramsClassAnnotation(
    register = {Genotype.class, Creature.class},
    registerFromInfo = {"Population", "GenePool"}
)
public class StandardState implements NetFile {

    @ParamAnnotation(stringType = "o sim_params")
    public Object simParams;

    @ParamAnnotation(stringType = "l GenePool")
    public final List<Object> genepools = new ArrayList<>();

    @ParamAnnotation(stringType = "l Population")
    public final List<Object> populations = new ArrayList<>();

    @Override
    public String getShortDescription() {
        return ...;
```

```
    }
}
```

Listing 6.3: StandardState.

The `simParams` field will be assigned an instance of `FreeObject` type, containing all experiment settings, and each `GenePool` will contain a list of `Genotype` instances. The choose of `FreeObject` is dictated by the fact that the definition of the object is not available beforehand through the regular `info`. It is worth pointing out that the code presented in the mentioned listing provides all information needed by the FJF to properly serialize and deserialise that structure (using utilities provided by `AccessOperations`). Beside the typical description of the class (`@ParamAnnotations`) used by various FJF automated mechanisms, the annotation of this class contains hints telling the registry of `FramsClass`es to also register `Genotype` and `Creature` types (based on their **Java** counterparts), and to load descriptions of `Population` and `GenePool` from Framsticks files.

Chapter 7

Summary

This work has briefly yet thoroughly presented all major elements and aspects of the developed software solution: the Framsticks Java Framework.

During the development phase several useful tools were used, including static code analysis and automated **GUI** testing.

The FJF utilized several non-trivial aspects of **Java** programming language, like reflection and annotations.

The type model defined by the Framsticks system has been implemented in the **Java** language, providing a solid base for any **Java** application related to the Framsticks system.

The network protocol has also been implemented in a low-level manner, thus enabling the creation of various applications communicating with native Framsticks servers. Beside the client-side implementation, the FJF also provides full server-side implementation, allowing to expose arbitrary **Java** data structures through mentioned protocol.

Specially designed **GUI** module has been prepared, allowing the user to communicate with the Framsticks servers in a convenient fashion.

The FJF allows to use native Framsticks servers as computation nodes in a distributed experiment.

Together with the network protocol implementation, **GUI** also provides the user with an insight to the distributed experiment being controlled in the FJF.

The adopted approach has been validated by the preparation of two distributed experiments, one being a trivial prime number searching, and the second being a distributed version of a standard Framsticks experiment.

The Framsticks Java Framework has been designed to be an extensible and open framework. Many features provided by the FJF were not used in their full capabilities during the validation phase, but they were designed specifically for future extensions.

Bibliography

[APT13] *Apache Tomcat*, http://tomcat.apache.org/, 2013.

[ERL13] *Erlang*, http://www.erlang.org/, 2013.

[FES13] *FEST – Fixtures for Easy Software Testing*, http://fest.easytesting.org/, 2013.

[FIN13] *FindBugs*, http://findbugs.sourceforge.net/, 2013.

[FNP13] *Framsticks Network Client/Server Specification*, http://www.framsticks.com/common/server.html, 2013.

[JAV13] *Java Platform, Standard Edition 7 API Specification*, http://docs.oracle.com/javase/7/docs/api/, 2013.

[KU96] Komosinski, M., Ulatowski, S., *Framsticks Web Site*, 1996, http://www.framsticks.com.

[KU09] Komosinski, M., Ulatowski, S., *Framsticks: Creating and Understanding Complexity of Life*, chap. 5, 107–148, Springer, New York, 2009, second ed., URL http://www.springer.com/978-1-84882-284-9.

[MAV13] *Maven*, http://maven.apache.org/, 2013.

[SCA13] *The Scala Programming Language*, http://www.scala-lang.org/, 2013.

[TES13] *TestNG*, http://testng.org/doc/index.html, 2013.

[XVF13] *Xvfb*, http://unixhelp.ed.ac.uk/CGI/man-cgi?Xvfb+1, 2013.

Streszczenie

Tematem pracy jest rozwój środowiska do prowadzenia rozproszonych obliczeń w systemie Framsticks. W jej ramach został opracowany projekt oraz wykonana implementacja tego w środowiska w języku **Java**, w związku z czym otrzymało ono nazwę Framsticks Java Framework.

Początkowe rozdziały pracy przedstawiają narzędzia wspomagające proces rozwojowy (takie jak statyczna analiza kodu, automatyczne testowanie interfejsu użytkownika) oraz elementy języka **Java** szczególnie istotne z punktu widzenia FJF (refleksja, anonimowe klasy).

Dalsze rozdziały prezentują szczegółowo wszystkie najważniejsze elementy FJF, wraz z nakreśleniem ich roli ich na tle całości rozwiązania, począwszy od implementacji modelu typów zdefiniowanego z systemie Framsticks oraz implementacji odczytu i zapisu danych kompatybilnego z systemem Framsticks, poprzez komunikację sieciową i reprezentację struktury zdalnego serwera, kończąc na graficznym interfejsie użytkownika.

W rozdziałach tych zawarto także przykłady praktycznych zastosowań kilku idei programistycznych, takich jak na przykład ścisłe rozdzielenie struktur danych od algorytmów na nich operujących, czy obsługę wyjątków w środowisku asynchronicznym i rozproszonym.

Przedostatni rozdział zawiera opis przykładowych eksperymentów przygotowanych w oparciu o FJF, które stanowią weryfikację poprawności przyjętego podejścia: trywialnego poszukiwania liczb pierwszych oraz zrównoleglonej wersji standardowego eksperymentu dostarczanego wraz z system Framsticks.

Ostatni rozdział zawiera podsumowanie prezentowanego rozwiązania.