A Test Infrastructure for Layout Algorithms

Annika Pooch

Master's Thesis
2017

Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden
Department of Computer Science
Kiel University

Advised by
Dipl.-Inf. Christoph Daniel Schulze

Selbsständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Abstract

Testing is a necessary and important part of every software development project. There are many different established strategies and many frameworks have been developed to support the testing process. Some of these are universal, but often it is sensible to adjust frameworks to optimize them for a special project.
In this thesis such a specialized framework was developed for the usage in the Eclipse Layout Kernel (ELK) project. The framework aims to make it easy and comfortable to write tests for the *automatic layout algorithms* implemented in the project. Automatic layout algorithms compute a layout for graphs.
The implemented framework is based on JUnit and supports different kinds of tests: *black box tests*, *white box tests*, and *analysis tests*. The additional features of the framework focus on the graphs as test input and the test results. Important properties are the usability in terms of effectiveness, efficiency, satisfaction, and performance.
The evaluation shows that the framework allows to specify test cases effectively and efficiently. In a small case study the subjective satisfaction was evaluated as good, the framework was comfortable to use and it was easy to write tests. Overall the framework was suitable for the usage in the project and only few features were missed.

Acknowledgements

I want to thank my advisor Christoph Daniel Schulze for all the helpful and instructive advice that made me learn a lot. Additionally I want to thank Prof. Dr. Reinhard von Hanxleden for the opportunity to write my thesis at the group for embedded and real-time systems. Further I want to thank Nis Börge Wechselberg and all the others who made it such a nice time.
I want to thank Josefine Wegert for reading and correcting my thesis.
Further I want to thank Janine Preuß for the never ending support and advice. Finally I want to thank my sister, my parents, and my grandparents. You are always there when I need you and your absolute and unconditional support made my studies and all the great experiences around possible.

Contents

Chapter 1

Introduction

The aspect of quality assurance is an important and central part of every software engineering project. In turn a major part of quality assurance is testing the software. Software tests have to be planned and performed many times and in every stage of the development process. That requires much time and effort and therefore it is important to test in an efficient and suitable way. There are many different strategies, methods, and tools available in order to test all the different types of software. Some frameworks are usable for a great variety of software, while others are very specialized. This work focuses on the development of a specialized test framework: a framework to test *automatic layout algorithms* implemented in the ELK project[1].

These *automatic layout algorithms* are used in order to compute a layout of a graph. Graphs can be used in many disciplines to visualize the different types of problems and solutions. One of these disciplines is the computer science and an example use case in this discipline are flow charts used to visualize the execution of a program. An example for the usage of graphs are graph-based music recommendation techniques. These techniques use the information available about users, playlists and other interesting components to derive recommendations for songs that are most likely suitable for a specific user. In this context graphs can become very complex. One approach uses six different types of nodes and 16 types of edges [GL16]. Nodes can represent amongst others a *user* or a *song* while edges represent the relations between the nodes, such as an edge of the type *plays* can connect a user and a song. The visualization of such a graph can use boxes in different colors for the different kinds of nodes and different styles for the kinds of edges. In addition to the high number of different types such graphs can become very big. To manually draw a visualization of such a big graph is very time consuming and laborious [Kla12].

In general graphical visualizations are believed to be easier and faster to understand [Bla96], but this can only be reached with a graphical representation of high quality [Pet95]. Another challenge is to edit such a view [FH10], if for example new nodes are added to the graph. Especially if this happens more often, it can become necessary to rearrange the whole diagram. Additionally the interaction with such visual representations is often uncomfortable and time consuming [FH10].

The ELK project provides the possibility to extend tools with automatic graph layout. The tools, as for example an editor in which a graph is created, need to translate the graph into a format that can be understood by ELK. This graph is enriched with information, such as the layout algorithm that should be used and other options that influence the layout. The chosen layout algorithm will attach the layout information to the graph, such as the positions and sizes of the components. These layout information can then be used by the tool to draw the graph. Automatic layout can save much time and effort: according to Klauske a user spends an estimated 25% of the time on manual layout adjustments [Kla12].

The test framework developed in this work should be used to verify that the output of a layout algorithm conforms to its specification. One example for such a specification is that no nodes overlap with each other. A test that checks this specification has to specify the input for the layout algorithm

[1] https://www.eclipse.org/elk/documentation.html

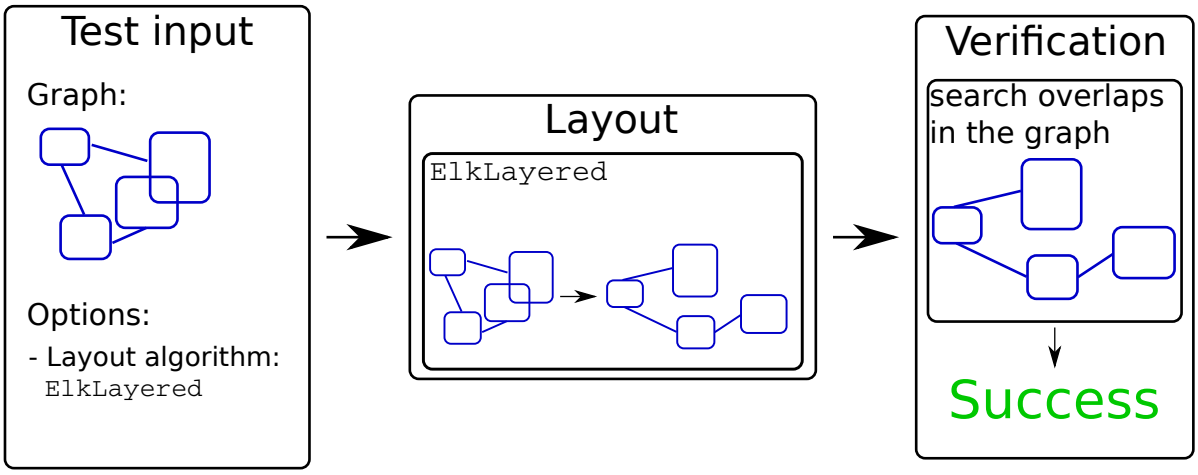


Figure 1.1. An example for the execution of a test

and implemented code to verify its output. The input consists of one or more graphs with the a layout algorithm and other necessary options attached. The framework has to execute the layout algorithm with all the specified graphs and has to use the verification code afterwards to test the output. Figure 1.1 shows an example for the execution of a test that verifies that no nodes overlap with other nodes. A small graph is specified as input and the layout algorithm that should be tested is set as an option on the graph. The framework executes the layout algorithm with the graph and then uses the verification code with the laid out graph as input. The verification code checks whether there are overlaps between nodes in the graph. This is not the case for the output graph and therefore the test will succeed.

1.1 Problem Statement

The goal can be summarized as the implementation of a test framework that makes it easy to write tests for the verification of layout algorithms implemented in the ELK project.

The framework to be developed in this work has to provide an efficient and comfortable way to create and execute tests. The developer that creates a test should be able to focus on the content of the test, not on the technicalities required to execute it. It is the framework's responsibility to execute the tests and to present the results.

The development and execution of tests should be well suited for the project and fit well into the workflow and the used tools. Like the source code of the project the tests should be developed with Eclipse and be written in Java. One of the demands for the framework was that it has to be based on JUnit because it is comfortably integrated in Eclipse, known by most developers, and supported by many tools.

The features that have to be provided by the framework are the following:

Import graphs In order to test layout algorithms graphs are required as input. It requires many graphs with different sizes and properties in order to gain an adequate test coverage. One feature to be implemented in the framework is the ability to import such graphs from files. This allows using the same graphs for several tests and using real world examples exported out of projects using ELK.

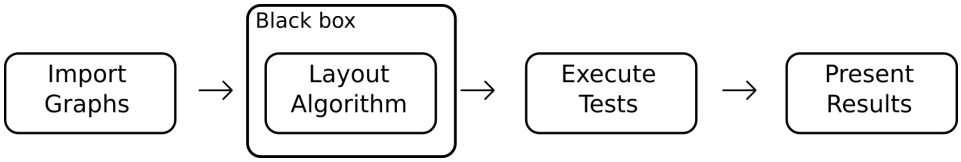


Figure 1.2. The execution of a black box test

Black box tests A *black box test* is used in order to verify the output of a whole layout run, as shown in Figure 1.2. The layout algorithm can be viewed as a *black box* and no knowledge is needed about its implementation, only about its specification. This kind of test should be possible for every layout algorithm as soon as it is integrated in ELK.

White box tests The purpose of a *white box test* is the verification of interior results of a layout algorithm. This requires more knowledge about the structure of the algorithm needed in addition to its specification. This kind of test cannot be available for every layout algorithm without adjustments. Therefore the execution of white box tests is not demanded for every layout algorithms, but only for the `ElkLayered` algorithm.

Present results One important part of the test framework is the presentation the test results in a structured and readable way. They have to include all necessary information to identify a test case. Features provided by JUnit should be used in order to achieve compatibility with the Eclipse Integrated Development Environment (IDE) and other tools to present the results.

Some additional features are nice to have besides the features defined above:

Random graph generator In addition to the import of graphs it is useful to generate graphs randomly. The predecessor project implemented a random graph generator that can be migrated to the ELK project. These graphs can be used in addition to the imported graphs. This feature leads to a fast and comfortable way to create new graphs.

Quality measurement Besides the correctness of the results, their quality is important as well. This quality can be measured with analyses that are part of Graph Analysis (GrAna). GrAna was developed for the predecessor project of ELK and the analyses can be migrated to ELK. Their results can be compared to old results in order to evaluate the quality. This comparison can be done in an *analysis test* and if the quality of the results decreased too much a failure should be reported by the framework.

Performance measurement In addition to the quality of the results the execution time of the layout algorithms is relevant. The execution time can also be compared to the last test runs and a failure should occur if the execution time increases beyond an acceptable level.

Automatic execution Besides of the manual execution of tests in the Eclipse IDE it is sensible to execute the tests automatically triggered by events or in regular intervals. Such an event can be a commit or a build on the build server.

Debug mode If the layout algorithm supports the execution in a debug mode it can be useful to automatically execute failed tests again with the algorithm in this mode in order to find the fault.

Test severity It can be sensible to assign a severity to the tests. Examples for such severities are *fatal*, *non fatal* and *disabled*.

Some tests are very central and it is no longer sensible to execute any other tests on the same input or even on the same layout algorithm if such a central test failed. Such a test is a *fatal* test. If a *non fatal* test fails the other tests are executed in the usual way. A test can be tagged as *disabled* if the tested feature is temporarily not supported. This should be done with a message that describes why the test is disabled and a warning would be sensible if such tests are encountered.

Prioritization of test cases If there are many tests in a project and the execution of these tests takes much time, it can be sensible to assign a priority to each test case. This priority determines the order in which the tests are executed. This is in general only done for sets of tests that are executed many times and take a long time to execute. The prioritization is done with the aim to maximize an objective function. One example is that subsystems should be tested in the order of their historical propensity. Often it is sensible to improve the time needed to detect faults within the testing process. If a test that leads to a fault is executed earlier the developer has an earlier feedback and can start debugging earlier [RUC+99].

1.2 Outline

This section gives a short overview about the chapters and their content.

Chapter 2 provides the information necessary for the following chapters. First of all methodologies of software testing and after that the tools and technologies used in this work are explained. These are the necessary foundations of the testing of software. Then it presents information about the ELK project and about automatic graph layout.

Chapter 3 provides an overview about related work. It introduces established test frameworks and analyzes how testing is done in similar projects. In addition to that it presents standards and publications about the testing process and the creation of tests.

Chapter 4 discusses how layout algorithms can be tested, focusing on the ELK project. First the current situation and the predecessors of the developed test framework are described. After that a testing strategy for the project is proposed together with the kinds of tests sensible for the project.

Chapter 5 gives an overview about the testing process from the perspective of a test developer. The features of the framework are explained together with their advantages, and example test cases are designed that show how the features can be used.

Chapter 6 describes the framework from the framework designer's perspective, explaining how the framework is structured and how these features are implemented.

Chapter 7 presents an evaluation of the framework. The framework was used by two of the three main developers of the project and their feedback is analyzed. This evaluation is based on a publication that proposes a framework for the evaluation of test methodologies and frameworks. In addition to that the performance of the framework is analyzed.

Chapter 8 provides a summary of the work and an overview of some features proposed as future work.

Chapter 2

Preliminaries

Testing is an important, time-consuming and complex part of every software development project. Therefore there are many books and papers about the testing process that give guidance on how to test, introduce process models, and report experiences. Much work has also been spent on tools supporting the testing process. There are tools written especially for the purpose of testing software and for many other software engineering tools the integrated support of testing is an important part. Some relevant methods and techniques are introduced in this chapter.

Since this work focuses on the testing of layout algorithms and not on the testing process in general, techniques and methods from the field of automatic graph drawing will be introduced as well.

2.1 Methodologies

2.1.1 Types of Tests

Different types of tests take place in different stages of the development process [TLM+10; Key03; Tam13].

Unit test Testing a single unit of the designed software is the purpose of a unit test. The surrounding of the tested unit can be simulated by *mock objects* that imitate the behavior of the components the unit interacts with. One advantage of unit tests is that they can be written very early because it is not necessary to have a completed software prototype to execute them. These tests are normally written and executed by the programmer in parallel to the development work. There are several frameworks for unit testing, such as JUnit[1].

Integration test Integration tests are used to verify that the interfaces of the components work properly. Different components are executed together and the interactions between them are tested. Therefore these tests can be executed as soon as parts of the software are ready to be integrated. These tests verify the interaction between components. In order to execute integration tests there is a testing environment constructed in which all the required dependencies are available and mock objects are used for parts of the project that should not be tested, such as external websites.

Scenario test During a scenario test the tester goes through a whole use case and verifies that the system works as expected. These tests can be as well executes automatically.

System test System tests, sometimes called functional tests, verify the developed system as a whole, using its public interface. Depending on the kind of the interface these tests can be very different. In case there is a GUI to interact with the software, the parts of the GUI are tested and it has to be verified that all features can be accessed and the results are right. A system test should be performed in an environment that is as similar to the environment the software will be used in, if possible.

[1] http://junit.org

Stress test and performance tests During a stress test a large number of requests is sent to the system and the response time is measured. There are special tools available for this purpose. Normally just the response time and not the correctness of the results is tested. With the help of performance tests bottlenecks can be found.

Acceptance test Acceptance testing is done to verify that the software meets the customer's needs. During these tests all sorts of other tests can be performed, such as functional tests and performance tests. Criteria for acceptance tests can be quite subjective, such as *easy-to-use*. These tests are executed by the customer or by a company working for the customer.

Regression tests These tests are executed after parts of the already completed software changed. This normally happens many times and should therefore be automated.

The tests listed above are all functional tests, except for the stress and performance tests, which are only functional if a certain performance is claimed by the customer or the environment. There are also different types of non-functional test that verify that the software is stable, efficient, secure, fault-tolerant, user-friendly and well-looking. Other commonly used categories for tests are black box and white box tests.

Black box test A black box test can be written without further knowledge of the system's innards. All that has to be known is the systems functional specification and the tests verify its input-output-behavior. These tests can be written by everyone, such as a customer, and can be developed as soon as the specifications are known.

White box test A white box test is written with knowledge about the implementation of the system. The components of the system and the interactions between them can be tested by white box tests.

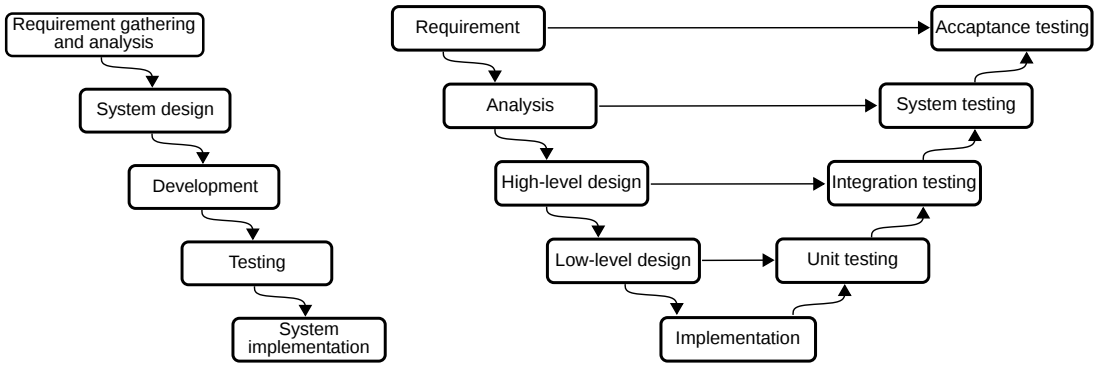
For many projects it is reasonable to prepare a test plan that includes the system description, testing strategy, testing resources, testing metrics, testing artifacts, and testing schedule [Key03].

2.1.2 Development Workflow

Before a software development project can be started it has to be planned how the development process should be managed. While planning the workflow it is important to explicitly integrate the testing activities at the right points. The different types of tests can be executed during different phases of the project. There are some tests that can be executed as soon as there is source code, some are executed as soon as there is a first version of the whole system, and some have to be executed repeatedly after parts of the system have been changed. The management of a project can become quite complex and therefore it has proven to be effective to organize the workflow of a project along the lines of a model [Mat07; DS16].

One of the earliest process models is the *waterfall model* that does not seem to be used that much any more [DS16]. It is illustrated in Figure 2.1a. This model consists of a sequence of phases: *requirement gathering and analysis*, *system design*, *development*, *testing*, and *system implementation*. Each of the phases only starts after the previous phase is finished. There is a testing phase in this model, but some testing activities can already be done in other phases. The unit tests are implemented and executed during the *development phase* and the acceptance tests are done in the *system implementation phase*.

A more recent model is the *V-model* that is based on the waterfall model [DS16]. It is illustrated in Figure 2.1a. This model assigns testing activities to each development phase. The model has the shape of a "V" with the waterfall model on the left side and validation phases on the right. The validation phases are processed in parallel to the development phases. In this model the testing process begins



(a) The Waterfall Model **(b)** The V-Model

Figure 2.1. Process models used in software development.

early and therefore there is a good chance to find an error early, hopefully leading to lower costs. This model shows as well that tests are highly dependent of other tasks. In order to specify the demands in detail it is is necessary to derive test cases out of them.

To develop high quality software it is sensible to not just test the software, but to have code reviews as well. Tests are necessary to find bugs in software in a repeatable way, but reviews can lead not only to correct software, but to software with a higher overall quality [KS07].

The models focus on the development of software from the costumer's order to the delivery of the software. Tests that are executed many times on a software after a release are not handled by the models, but happen afterwards.

Another development workflow is the *test-driven development*. There the test cases are implemented prior to the source code. The tests are used to specify the requirements of the project.

2.2 Technologies

2.2.1 JUnit

One of the demands for this thesis was the usage of *JUnit*. JUnit has been used in the ELK[2] project before and therefore the developers are already experienced in its usage. The old tests have to be migrated to be used with the new framework, but the parts using JUnit statements can generally stay as they are.

JUnit is a test framework that provides features to test Java classes and methods. Kent Beck and Erich Gamma started to develop JUnit in late 1995. Since then JUnit became very common and therefore it is integrated in many IDEs and continuous integration tools. It is also integrated in Eclipse[3] and Maven[4] used in the ELK project [TLM+10]. Before JUnit has been introduced the developers had to test their Java classes manually. That was quite tedious and prone to failures, because often only the changed functionalities had been tested and failures caused in other parts had not been noticed. Therefore the invention of JUnit meant a big improvement for many Java projects [Tam13].

The implementation of a JUnit test generally starts with the creation of a *test class*. This class is required to be public and to contain a zero-argument constructor. In general there is one test class for

[2] https://www.eclipse.org/elk

[3] www.eclipse.org/

[4] maven.apache.org/

each production class. In the test class the *test methods* are implemented. A JUnit test method has to be annotated with `@Test`, be public, take no arguments, and not return anything. One feature of JUnit in order to avoid side effects and execute the tests independently of each other is the creation of a new instance of the test class for each execution of a test method. In the test methods the *assertion methods* provided by JUnit are used to check whether conditions are met. An example for an assertion is the following:

```
assertTrue("failure - should be true", value)
```

The first argument is a failure message that is returned if the second argument is `false`. There is a great variety of assert methods and the possibility to use a *matcher* to compare the actual and expected value.

The tests are executed by a test *runner*. There are some runners implemented by JUnit and it is possible to implement custom runners in order to execute tests in a different way. This is done by extending JUnit's `Runner` class.The default runner is the `BlockJUnit4ClassRunner` that executes the test methods of a single test class. The `Suite` runner provides the possibility to execute several test classes and the `Parameterized` runner executes the tests with different parameters. The constructor of the test class is invoked with the parameters and the created instance is used to execute one test method. For each test method a new instance of the test class is created. There are some more runners provided by JUnit and many others developed by third parties. Which runner should be used to execute the tests in a test class is specified by its `@RunWith` annotation.

Beside the `@Test` annotation there are several other annotations available. Methods on the test class annotated with `@Before` (or `@After`) are executed before (or after) each test method. Methods annotated with `@BeforeClass` (or `@AfterClass`) have to be static methods and are executed once before (or after) all test methods of a test class. Another annotation is the `@Ignore` annotation that causes the test runner to ignore the test method [TLM+10]. The advantage of the usage of the `@Ignore` annotation compared to the omission of the `@Test` annotation is that the number of ignored tests can be reported by a test runner. Additionally it is possible to specify a message containing a reason for the `@Ignore` annotation. The annotation is also usable as class annotation. The `@Test` annotation has values that provide the possibility to specify a timeout or expected exceptions.

Figure 2.2 shows an example for a simple JUnit test in Eclipse. The class that should be tested is `Elk` (left) and the test class is `ElkTest` (right). JUnit is integrated in the default installation of the Eclipse IDE for Java Developers and a JUnit test can be created and executed very easily. The results of the tests are displayed in a readable way, as shown in Figure 2.2 at the left side. There is a bar that is green if all tests had been executed succeessfully or red if at least one test failed or an error occurred. This bar was introduced very early in JUnit GUIs and is now part of every integration of JUnit in IDEs. Developers even call a build a "green build" or a "red build" because of this bar [Tam13].

In the Eclipse IDE a test can be simply started by selecting *Run as... JUnit Test*. In order to execute the test `JUnitCore` is invoked with the test class and JUnit looks for the `@RunWith` annotation, constructs an instance of the runner with the test class, and calls its `run(RunNotifier)` method that executes the test. The notifier that is the parameter of the `run` method has to be notified by the test runner about the following events: a test or a test run starts or finishes, a failure or an assumption failure occurs, or a test is ignored. The notifier has a list of `RunListeners` that are notified in case of the events, making it possible to collect the results of the tests and display them afterwards.

2.2.2 Build Servers

In the context of software development, *continuous integration* [Pau07; Man11] describes the strategy to continuously build and test the system under development. The term has first been used in this context

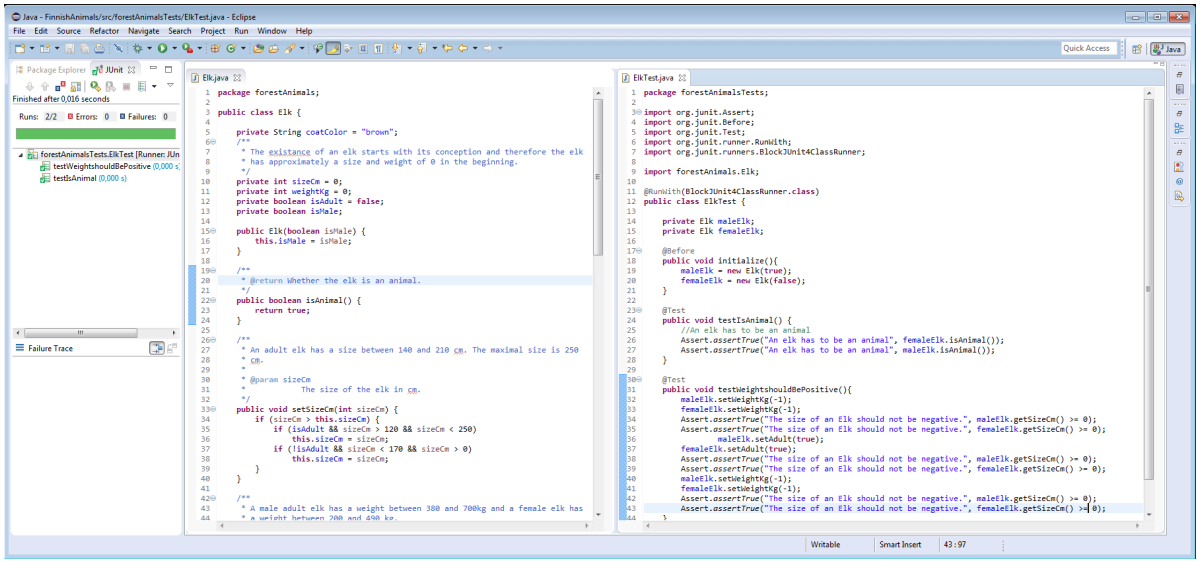


Figure 2.2. An example for a JUnit test in Eclipse.

first by Martin Fowler and Kent Beck in 1999. These days it is a common practice and supported by many tools.

More developers working on different parts of the same software project increase the need to regularly integrate the different components into the whole system. Every time this is done integration and system tests should be executed. If the integration and related testing is planned for the end of the project, the risk of an unplanned delay and quality problems becomes very high. Already some time before the release a special integration phase has to start. The whole development comes to a stop so that a single release manager can build the system on a single build machine. Before that the developers run the unit tests on their own code and the whole process of integration and system testing can solely be done in the end of the development process.

With continuous integration the whole process of testing, deploying, and integration is done regularly, usually in response to every commit. In case of failures everyone is notified and therefore bugs will hopefully be found shortly after they are introduced. This rapid feedback makes it much easier to fix bugs and improves the quality of the software.

Martin Fowler names the following points in his popular article "Continuous integration" [Fow17]

▻ developers integrate their work frequently, usually each person integrates at least once a day

▻ each integration is verified by an automated build and tests are executed

He points out that this approach helps many teams to reduce integration problems and to develop cohesive software more rapidly.

For the practical development work these points can be interpreted in the following way [Pau07]: the developers run private builds and unit tests in order to make sure not to break the integration build. As soon as this is successful the code can be committed. Integration builds are triggered several times a day on a separate build machine. For the builds all the tests have to pass and in case of a failure it has a high priority to fix the build.

There are several tools supporting the process in an automated way and some of them are introduced in the next sections.

Maven and Surefire

Maven [5][Pau07; TLM+10] is an automated build tool with support to manage Java-based projects and to gain high quality. To facilitate quality assurance Maven provides quality project information and guidelines for best practices development. The original purpose of Maven was to simplify the build processes in the Jakarta Turbine project and it is still one of Maven's main objectives to simplify the build process. It should be easy to start a new project so that the developers are able to start as soon as possible with the implementation of the software, without having to spend much time and work on the setup and configuration of the build system beforehand. One principle of Maven to achieve this is *convention over configuration*: there are conventions the developer can follow in order to avoid the work to configure the build. If the developer does not want to follow the conventions it is possible to configure Maven accordingly. This is in general more complex than it would be for example with Ant. Another drawback is that not everything is supported by Maven, but for example by Ant[6]. Apache Ant is a Java library and command line tool that is often used to build Java applications. It can be used to pilot a process that can be described in terms of targets and tasks. To configure Maven it is necessary to create a `pom.xml` file, shortened from Project Object Model. Maven can be used, to build, test, or generate the projects documentation.

There are many plugins available for Maven which implement a wide range of features and it is even possible to add own plugins. One available plugin is *Surefire*[7]. The purpose of Surefire is the execution of unit tests implemented for an application. Surefire is designed in a way that best practices were used as guidelines and one of them is the usage of test case naming conventions to locate and execute tests. There are default parameters for these naming conventions, which can be adjusted. After the tests are executed a report is generated in the file formats `.txt` and `.xml`.

Hudson

Hudson[8] [Man11] is a continuous integration server developed as an Eclipse project. The main purposes of Hudson are to build and test software projects continuously and to monitor executions of externally-run jobs. Besides an easy installation and easy configuration, two of Hudson's features are JUnit test reporting and plugin support. For example different build tools, such as Maven and Ant, different unit testing frameworks, and different code analysis tools are supported. Support for Git is available in default Hudson installations and allows to use different configurations. It can for example be specified which branch should be built, and only changes committed on this branch will trigger a new build. Apart from builds and tests, Hudson also supports software releases, documentation, and monitoring amongst other features.

Travis CI

Travis CI[9] [Cry15; PW15] is another widely used continuous integration solution that can be used to build and test different projects in different programming languages. It is easy to use, especially in combination with a Git repository on GitHub[10]. GitHub provides the possibility to define a post commit hook that triggers Travis CI to build and test the project. The configuration is done in a `.yml` file. The testing done by Travis CI is mainly focused on unit tests, but there are some features useful for

[5] `https://maven.apache.org/`
[6] `http://ant.apache.org/`
[7] `http://maven.apache.org/surefire/maven-surefire-plugin/`
[8] `http://hudson-ci.org/`
[9] `https://travis-ci.org/`
[10] `github.com`

integration tests as well, such as the deployment of common database management systems. Another feature of Travis CI is the possibility of continuous deployment, which makes it possible to deploy a new version of the software automatically.

2.2.3 Eclipse

The *Eclipse Foundation*[11], founded in 2004, provides an environment for open source projects. One of the projects is the *Eclipse project*[12] which is concerned with the Eclipse SDK and can be regarded as the "Eclipse top-level project". Sub-projects of this project are amongst others Java development tool and the Plug-in Development Environment. The entirety of projects focuses on the development of an open development platform. Eclipse is well known for its Java IDE, but they provide IDEs for other languages as well. The IDEs can be easily extended by many available tools. Eclipse applications, such as an IDE, are based on a dynamic plug-in model and can be built out of a set of plugins. The minimal set needed to build such an application is known as the Rich Client Platform. There are many other plugins available to adjust the application to the needs of the user. Such non-minimal applications are called Rich Client Application[Ecl17].

2.2.4 Eclipse Layout Kernel

ELK is an Eclipse project that provides automatic layout algorithms and the necessary infrastructure to connect editors or viewers with them. ELK arose from the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[13] project. In the context of the KIELER project, research on the enhancement of graphical model-based design of complex systems is conducted. The project's parts are the semantics and the pragmatics part. The semantics part is concerned with the semantics of model-based languages. They contribute a tool chain for the synchronous language Sequentially Constructive Statecharts (SCCharts)[14]. The pragmatics part is where the ELK project originates.

The pragmatics part of the project arose from the perception that users require new and better mechanisms to interact with graphical models. The assumption is that graphical models are often better to present facts more readably and comprehensibly. On the other hand manual placing and routing of nodes is very time-consuming and the editing of models is often more complex. Motivated by these facts the KIELER project conducts research in the field of models pragmatics [FH10]. The subject of pragmatics, in this context, encompasses all interactions of a user with the model during its design, such as editing and browsing the model. To synthesize views automatically, leading to less time exposure and effort necessary to manually layout a model. As part of this a layout of the graph is computed automatically and features like filtering of a view and label management enhance the readability. The KIELER Lightweight Diagrams framework aims to reduce the time spend in creating a view of a model, increase efficiency, and provide interactivity. The efficiency is important to be able to browse rendered diagrams without table delays and interactivity can be implemented by highlighting ans semantic zooming that is especially sensible for displaying simulations [SSH13].

The ELK project is implemented in Java and composed of Eclipse plugin-projects. The basic parts of ELK are the *kernel* and the *layout algorithms* that can be used by editors to compute a layout of a graph. To use these layout algorithms the graph that should be laid out has to be converted into a special format, a *ElkGraph*. This is done by layout connectors or client code of the editor. To this *ElkGraph* options can be added and the layout algorithm is invoked. This layout algorithm attaches

[11] http://www.eclipse.org

[12] http://wiki.eclipse.org/Eclipse_Project

[13] https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Home

[14] http://www.rtsys.informatik.uni-kiel.de/en/research/kieler

layout information to the graph. With these information the editor can draw a view of the graph. It is possible to parameterize the chosen algorithm to obtain the required results.

2.3 Automatic Graph Layout

In many fields in computer science graphs are used to model and solve practical problems. In its easiest form a graph consists of a set of nodes and a set of edges between nodes. The field of graph theory addresses the properties of graphs and theoretical problems formalized on graphs and their solutions. Often these problems are of practical importance. One basic problem is for example the shortest path problem that can be used to find the shortest connection between two nodes in a network.

In a view on a graph the nodes are normally represented by circles or boxes and the edges by lines drawn between the nodes. These views on graphs are often used in computer science to present information in a more readable and understandable way. There are many forms of diagrams established in computer science like flow charts and class diagrams.

Like already pointed out in Section 2.2.4 it is very time consuming and tedious to manually place and route the nodes and edges. Unfortunately this has to be done in a proper way, because the quality of the layout has a great influence on the readability [SSH14].

Time can be saved by using automatic graph layout algorithms. There are different approaches for layout algorithms, like for example the force-directed [FR91] or the layer-based [STT81]. These approaches lead to quite different results and are therefore suitable for different problems.

2.3.1 Layered Graph Layout

Im 1981 K. Sugiyama, S. Tagawa and M. Toda presented the approach of *layer-based layout* [STT81]. In this approach all edges are arranged in the same direction and the nodes are arranged in layers. It has already proved to be a good choice for different applications [SSH14].

The layer-based approach focuses on the following aspects to gain a high readability of the graph:

Hierarchical layout of nodes It is easier to understand a diagram if the nodes are in some regular form. For example a clustered layout helps to grasp the structure of a diagram.

Few edge-crossings To trace a path is much more difficult if lines cross [Pur97].

Straightness of lines It is easier to trace straight lines [WPC+02]. This is important for edges between nodes in adjacent layers and for *long span edges* that connect nodes in more distant layers.

Close layout Nodes connected to each other should be laid out close to each other. That leads to short paths.

Balanced layout Incoming and outgoing edges of a node should be drawn in a balanced way. That results in a better readability of branching and joining paths.

The layout algorithm Sugiyama et al. developed is structured in four steps of which each can be implemented in different ways.

Step 1 In this step the nodes are assigned to a layer. This is done in a way that the successors of a node in layer n are located in a layer $m > n$ with a minimal m. If there are edges between nodes with $m > n + 1$, i.e. long spanning edges, dummy nodes are inserted. In case of a cycle in the input graph, the graph is condensed.

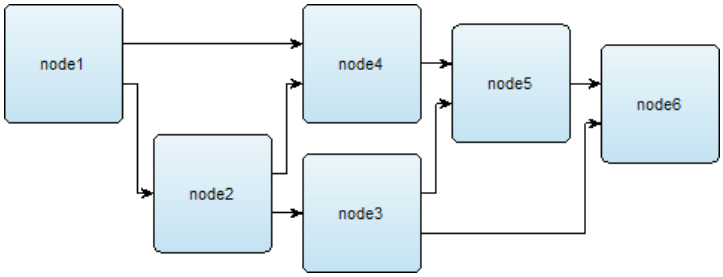


Figure 2.3. A simple graph laid out by ELK Layered.

Step 2 The nodes in each layer are ordered in a way that crossings between edges are reduced.

Step 3 Horizontal positions are assigned to the nodes in each layer. This is done in a way that long spanning edges can be drawn straight, connected nodes are near each other, and the incoming and outgoing edges are drawn balanced.

Step 4 The picture of the graph is drawn. Therefore the dummy nodes are deleted and the long span edges are restored. In actual algorithms the drawing is not considered to be part of the algorithm. In the implementation by Schulze et al. for example this phase is the edge routing step [SSH14].

Another possible solution is to implement the layer-based approach in five phases [SSH14]: elimination of cycles, layer assignment, crossing minimization, node placement and edge routing. In this approach, Step 1 of the originally design is split into cycle breaking and layer assignment. The cycle breaking phase does not just condense the graph, but reverses edges, and the original graph is restored afterwards. The edge routing phase assigns the positions of the nodes and edges like Step 4 and draws the long spanning edges. Another difference is the possibility to use different types of edges and not just straight lines.

2.3.2 Structuring Layout Algorithms into Processors

If a layout algorithm can be structured into phases, such as the implementation of the layer-based algorithm by Schulze et al. [SSH14], it is useful to extend this structure.

The phases of the algorithm can be implemented in different ways and can therefore be adjusted to different problems. They are the skeleton of the algorithm and have to be present in every configuration. For example there are different implementations of the edge routing phase that enable the usage of different types of edges: orthogonal, polyline, or spline edges.

Schulze proposed to introduce *intermediate processors* before, between and after the phases [Sch11]. This approach has two main advantages:

The first advantage is the avoidance of code duplication. Often the different implementations of the phases have to perform similar tasks at the beginning and the end. These can be done by intermediate processors that run before or after each implementation of the phase.

The second advantage is that the intermediate processors make it easy to provide layout options by adding pre- or postprocessors. By doing this the different layout problems given to the phases are reduced to the same basic simpler problem and there have to be less lengthy conditional statements in the phase implementation. One example for this kind of option is the layout direction.

The drawbacks are that the dependencies between processors in the same slot have to be handled and that there are more iterations over the graph. Experience shows that the added iterations are

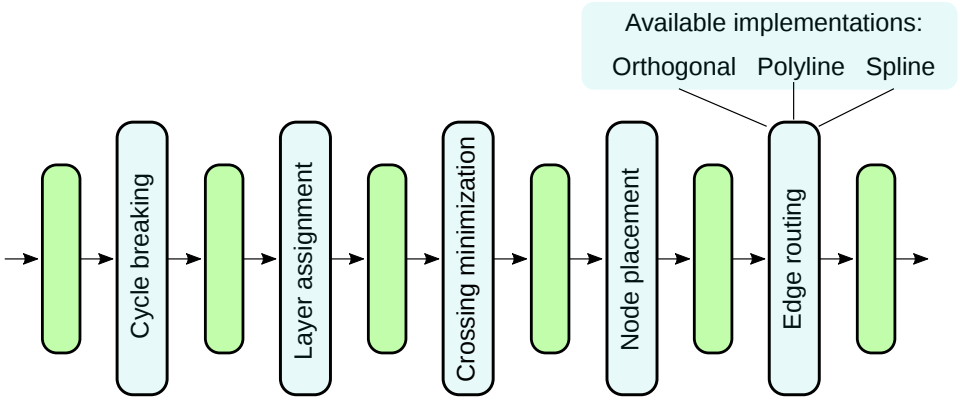


Figure 2.4. The phases of ELK-Layered with intermediate processing slots

no problem and that it is neither a big problem to handle the dependencies in case they are set in a proper way for the processors. The conclusion of the developers was that the advantages, i.e. easier implementation and well-structured code, far outweigh the performance impact [SSH14]. In addition this solution makes it easy to add new features.

ELK already provides the infrastructure required to implement layout algorithms in this way. Therefore not just the layered approach, but as many algorithms as possible (and reasonable) should be implemented in this way.

Sometimes it can happen, that phases are executed several times. This is especially the case, if the graph contains several levels of hierarchy and the phases have to be executed on every level separately.

2.3.3 Aesthetic criteria

How easy and fast a graph can be understood depends on the layout of the graph. There are aesthetic criteria [WPC+02; Pur97; Pur02] that are believed to improve the readability of graphs. Often layout algorithms are designed to optimize one or more these criteria. Some common criteria are:

- ▻ minimizing edge crossings
- ▻ minimizing edge bends
- ▻ maximizing symmetry
- ▻ maximizing the minimum angle between edges leaving a node
- ▻ maximizing edge orthogonality
- ▻ maximizing node orthogonality
- ▻ maximizing consistent flow direction (directed graphs only)

To find aesthetic criteria, discoveries made by other sciences are used, such as the principle of good continuation, invented by Gestalt psychologists [DET+99]. This implies that a path is much easier to perceive if the nodes are laid out along a smooth continuous line instead of in a zigzag pattern. Discoveries made by neurophysiologists reveal that crossing between two lines are much easier to handle if they cross in an angle close to 90 degree.

It is not just important to find the criteria, but to know how strong their influence is. This can be done by carrying out an experiment to measure how well and fast special problems on a graph can be solved or a graph can be understood. The graphs on which the experiments are performed vary according to the tested aesthetic criteria and the performance of the testers is measured.

If there are metrics defined according to the criteria they can be used to analyze a layout or they can be used by layout algorithms to define cost functions.

Chapter 3

Related Work

This chapter gives an overview about related work. This thesis is very practical and therefore there are not only scientific publications relevant as related work. First of all other frameworks are presented and after that the testing process in similar projects is analyzed. In addition standards are introduced and strategies for test case creation are presented.

3.1 Test Frameworks

This section presents some popular and universal test frameworks that are useful as orientation for this work. The most important framework is JUnit. Because JUnit is a used tool it was already presented in Section 2.2.1.

3.1.1 Google Test

Google Test[1] is Google's C++ test framework. It is organized as GitHub project and is the result of merging the formerly separate GoogleTest and GoogleMock projects.

One interesting aspect of this framework is that its design focuses on the creation of high quality tests. Therefore the documentation names aspects of good tests and describes how the framework supports these aspects.[2] Many of these aspects are relevant to this work and were already listed in the problem statement.

First of all tests have to be repeatable and independent. This is a necessary aspect and not only a property of good tests. The Google Test framework guaranties these properties by the execution of each test on different objects. In addition to that a failed test can be executed again in isolation to allow for quick debugging.

In addition, tests should be well organized and should reflect the structure of the tested code. In order to support that, Google Test provides the opportunity to organize tests into test cases, that is, groups of tests that share data and subroutines.

The next aspect is that tests should be portable and reusable. This is supported by the possibility to use the framework on different operating systems and with different compilers.

Another aspect is that a test should provide as much information as possible if it fails. In order to provide these information the framework does not stop the execution of the whole test suite if a test fails. There is also the possibility to define non fatal tests that makes the framework continue the execution even of the failed test. Therefore there is a complete overview about the situation after the test run is completed.

The next aspect is that the test developers should focus on the tests and should not bother with the technicalities. In order to support the developers in this aspect the framework automatically keeps track of all defined tests. Therefore a new test does not need to be added manually by the developer.

[1] https://github.com/google/googletest

[2] https://github.com/google/googletest/blob/master/googletest/docs/Primer.md

```
// Tests factorial of 0.
TEST(FactorialTest, HandlesZeroInput) {
  EXPECT_EQ(1, Factorial(0));
}

// Tests factorial of positive numbers.
TEST(FactorialTest, HandlesPositiveInput) {
  EXPECT_EQ(1, Factorial(1));
  EXPECT_EQ(2, Factorial(2));
  EXPECT_EQ(6, Factorial(3));
  EXPECT_EQ(40320, Factorial(8));
}
```

Algorithm 3.1. A small example for a test using the Google Test framework from their homepage [Goo17]

The last aspect is that the tests should be fast. A feature of Google Test is the possibility to reuse shared resources across tests to make multiple set-up and tear-down of these resources unnecessary.

As JUnit, the Google Test Framework uses assertions. The framework provided many assertions and it is possible to add user defined assertions. There are *fatal* and *non fatal* assertions and accordingly the result of an assertion can be a *fatal failure*, a *non fatal failure*, or success. There are basic assertions, binary comparisons, and string comparisons. Each assert in each category is available as `ASSERT_*` and as `EXPECT_*`. The first version is a fatal assertion and the second one is a non fatal. An example for an assertion is `ASSERT_TRUE(condition)` that verifies whether the condition is true. In addition a custom failure message can be provided.

Algorithm 3.1 shows the general structure of a test. `FactorialTest` is the test case name and `HandleZeroInput` is the test name.

Several test cases can be grouped in *test fixtures*, if they use the same test data. In this fixture a shared `SetUp()` and `TearDown()` method can be defined for all the contained test cases.

The tests are executed in a `main()` function with `RUN_ALL_TESTS()`. This will lead to the execution of all tests and to the presentation of the results. A feature of Google Test is the generation of an XML test report.

3.1.2 Bandit

An alternative to Google Test is the *bandit*[3] test framework. This framework can be used to test C++ code and aims to make the testing process pleasant.

Algorithm 3.2 shows an example bandit test. The principle the framework is based on is the creation of specification based tests that are explained in Section 3.4.1. There are already other frameworks for specification based development and testing available that are used by bandit as orientation, such as RSpec[4]. Very central in a bandit test is the usage of `describe()` and `it()` that are well suited for specification based tests. Inside of `describe()` it is specified what should be tested and inside of `it()` the tested feature is specified. For the example test in Algorithm 3.2 the underlying specification states that a calculator needs to be able to add and to subtract. These two features are checked by the test. Like Google Test this framework uses assertions to compare the current and the expected results.

[3] `http://banditcpp.org`

[4] `rspec.info`

```
#include <bandit/bandit.h>
using namespace bandit;

go_bandit([](){

describe("a calculator", [](){
  calculator calc;

  it("can add", [&](){
    AssertThat(calc.add(2, 3), Equals(5));
  });

  it("can subtract", [&](){
    AssertThat(calc.subtract(2, 3), Equals(-1));
  });
});

});
```

Algorithm 3.2. A small example for a test using bandit from their homepage [Bey17]

In addition to the described test cases the test needs to include bandit, use the `namespace` and the `go_bandit()` construct.

In order to execute tests with bandit a command line application is created that uses bandit to execute the registered tests. The results are printed to `stdout` and like Google Test the possibility is provided to create a XML file with the results. The overall result of the execution is an error level. If this level is 0 the test run was *successful*, otherwise it *failed*.

Like Google Test the framework provides the possibility to skip tests and in addition there is the opportunity to execute a subset of tests.

3.2 Similar Projects

This work analyzes two similar projects in order to have a look at how testing can be organized in such projects.

3.2.1 Open Graph Drawing Framework

One project that can be used as orientation is the Open Graph Drawing Framework (OGDF)[5] [CGJ+13]. Similar to the ELK project it provides automatic layout of diagrams, but it is developed as a self-contained C++ class library. The library is also a university project and supported by TU Dortmund, Osnabrück University, Monash University, University of Cologne, and TU Ilmenau. Another similarity is that the library focuses on the usage in other applications and scientific projects. In the context of this work the latest release from 2015 and the latest snapshot from 2017 has been studied.

First the release from 2015 will be analyzed. Besides the *regression tests* there is a *generator test* and a *file format test*. For these tests the Google Test framework is used. The *generator test* verifies the

[5] `http://www.ogdf.net/ogdf.php?id=`

generators of different graphs. The *file format test* is used to verify that the graphs can be stored to and imported from a file. This is tested with different graphs and different read and write functions. After this procedure it is for example verified that the graphs have the same number of nodes.

In addition to these tests there are the regression tests. These tests are used in order to evaluate the results of different layout algorithms and planarity tests. The algorithms are executed with different graphs. The properties of the graphs are varied in the number of nodes and the ratio of nodes and edges. For each of these input graphs the execution time of the tested layout algorithm is measured and the results are evaluated. The results of the Sugiyama approach are evaluated in terms of the number of crossings. With these results it is possible to compare the quality and performance of the algorithm in the evaluated aspects to old results. The planarity test on the other hand verifies the correctness of the result and leads therefore to a succeeded or failed test execution.

There is the possibility to execute the tests from the command line with the option to execute the tests using Google Test or all the regression tests. If the regression tests are executed all the results of the regression tests are printed and the failures are counted. Therefore there is only minimal support for the execution of tests and no specialized framework for this release.

In addition to this release the snapshot from February 2017 is analyzed. In this version of the project the testing infrastructure is different to the one used in the release. To execute the tests bandit[6] is used. There are tests in the categories *basic*, *graphalg*, *layout*, and *planarity*. For the tests in *layout* there are helpers that provide features to create and execute tests more comfortably. There are functions to create graphs and to add a random layout to these graphs. In addition there is the functionality to execute a layout on a graph with a random layout and to measure the execution time of the layout algorithm. There is the possibility to execute a layout algorithm with different graphs, but the assertions are not yet implemented. After the layout the average execution time is printed for a type of graphs, such as trees or planar graphs. This helper is used to execute tests on the Sugiyama, planar, energy and planarization algorithms.

The basic tests are tests that verify the basic features of the library. There are for example tests for generators, `math.h`, and the basic `graph` class. For the basic tests a helper class is introduced that is created in order to test all array classes.

The tests in *graphalg* verify graph theoretical algorithms, such as min-cost-flow algorithms and maximum adjacency ordering. In the *planarity* category are amongst others tests that verify the planarity tests and embeddings.

The test infrastructure seemed not to be completed at the time of this snapshot, but there were major improvements compared to the last release. Obviously there was a need for a test infrastructure in that project as well. Especially the helper class for the layout tests is interesting as orientation to this thesis. The class calls layout algorithms with different kinds of graphs, executes all the tests, and prints the results afterwards.

3.2.2 GraphViz

Another similar project is the Graph Visualization Software (Graphviz) project[7] that is available as open source software for Linux, Solaris, Windows, and Mac. The input is described using a simple text language and the output is available in different formats, such as images, `.svg`, or `.pdf`. Therefore the output of the layout algorithms is very different to the one in the ELK project. Consequently the supported options are different to the ones available in the ELK project.

[6] `http://banditcpp.org/`

[7] `http://www.graphviz.org/`

The tests implemented in the Graphviz project are regression tests. Therefore the output of the tested layout algorithm is compared to the output of old versions. The old output is stored for the comparison in the formats named above. The framework provides tools for the execution of all specified tests and the comparison of the results.

In the ELK project this kind of regression tests is not sensible and therefore the testing done in the GraphViz project is not that sensible as orientation. The regression tests in Graphviz expect the algorithms to compute exactly the same layout. This is not claimed for the ELK project. There are changes allowed that lead to another output as long as the layout has the expected properties. The expectations for the output in the ELK project are formulated as properties, such as "An edge is not allowed to overlap with a node." and not in fixed positions and sizes. Therefore it is not possible to compare the results with results of older versions, but it is necessary to implement methods that verifies that the results have the expected properties.

3.3 Standards in Software Testing

The software testing is an important part of the software development process and therefore there are standards for the software testing in any type of software development project. The authors of the standards aimed to design an internationally-agreed set of software testing standards that is universally usable. The five parts of the standards are:

29119-1-2013 - Software and systems engineering Software testing Part 1:Concepts and definitions [ISO13a] This part introduces the concepts and vocabulary necessary as foundation for the standard These vocabulary are also used by authors of books and publications. In addition there are examples of the standards application provided. The purpose of this part is to enable and improve the usage of the standard by being informative, providing a starting point, context, and guidance for the other parts.

29119-2-2013 - Software and systems engineering Software testing Part 2:Test processes [ISO13b] This part addresses the testing process considering different levels and different kinds of tests, such as functional and non-functional tests and manual and automated tests. This standard claims to be usable with any software development lifecycle model. It is designed following a *risk-based* approach that allows the prioritization of test cases with the focus on the most important features and functions.

29119-3-2013 - Software and systems engineering Software testing Part 3:Test documentation [ISO13c] This part of the standard gives guidance on how test documentation should be created. There are examples stated and the second part of the documentation.

29119-4-2015 - ISO/IEC/IEEE International Standard for Software and systems engineering–Software testing—Part 4: Test techniques [ISO15] This part of the standard addresses the testing techniques including the design of test cases and their execution. Additionally it considers the different phases and types of tests, such as unit, integration and system tests. In addition this part discusses the strategies for the derivation of test cases, such as *specification-based* or *structure-based* testing.

29119-5-2016 - ISO/IEC/IEEE International Standard - Software and systems engineering – Software testing – Part 5: Keyword-Driven Testing [ISO16] The fifth part of the standard describes an approach to describe test cases in a modular way, the *Keyword-Driven Testing*. This includes the main concepts and attributes of the approach together with the creation of test specifications, corresponding

frameworks, and test automation. In addition it states requirements on frameworks, tools, and a common data exchange format.

3.4 Test Techniques

For this work the interesting part of testing are test techniques. Software testing has already been discussed for a long time and therefore there are many publications and books available. This work concentrates on relevant subdomains because it is neither possible nor sensible to illustrate the whole field of software testing or test techniques.

Two aspects are important regarding test cases: the determination of test cases and the test coverage.

Myers published his fundamental and often cited book in 1979 [Mye97] and the strategies proposed for test case design in this book are still used and specialized by actual frameworks. He advised to combine the presented strategies to create a set of test cases because no strategy alone leads to a thorough set.

3.4.1 Specification-Based Testing

One sensible source for test cases is the specification of the product. No knowledge about the implementation, but only about the specified functions, is needed and therefore the created tests are functional black box tests. Many different strategies are sensible to derive test cases out of specifications. Which strategy can be used is affected by the underlying specification. Model-based testing for instance is only possible if there are models defined in the specification.

A kind of specification that is used by many publications and frameworks are formal specifications for example using the Z notation. There were fundamental discussions about software testing and formal verification techniques. Dijkstra's statement that testing can only show the presence of errors and never their absence is often cited as argument against software testing [DDH72]. Tanenbaum on the other hand stated that testing will nevertheless always be a necessary verification technique [Tan76]. Carrington and Stocks combined testing techniques with formal methods and aim to take advantages out of both paradigms [CS94].

One early publication from Hall discusses the manual and automated derivation of test cases from formal specifications [Hal88]. They present how test cases can be derived manually and used the Z notation as an example for formal specifications. They depict how the automatic derivation of test cases can be realized, but named research problems that had to be solved before.

Model-Based Testing

Model-based testing uses models stated in the specification to derive test cases. Such models can be for example finite-state machines. Besides of the test cases it is possible to derive test oracles from such models. A test oracle computes the output the system should have and compares it to the real output. There are different coverage criteria possible to rate the completeness of model-based sets of test cases. One possible demand is that every transition in the model has to be covered by at least one test case [PY09].

There are frameworks using model-based specifications as basis. One example for such a framework is the *Test Template Framework* [SC93]. This publication describes a structured strategy and a formal framework for refinement of test data, regression testing, and test oracles.

Scenario-Based Testing

Another possibility to state specifications is the usage of scenarios. These are used in the scenario-based testing.

An example for a scenario-based object-oriented framework is presented by Tsai at al. [TYS+03]. The publications aims to formalize scenario-based testing based on a specification using semi-formal specification languages. The proposed framework takes care for the preparation of test data for execution, the execution of tests, and the evaluation of test results using a database. The design of the framework is done using object-oriented design patterns and based on JUnit.

Equivalence Partitioning

One fundamental technique to derive test cases from specifications is the usage of equivalence classes. This was already explained by Myers [Mye97] and also in [PY09]. In mathematics sets can be divided in equivalence classes regarding an operation. One equivalence class contains all the elements that are equivalent regarding the operation and each element of the set is in exactly one equivalence class. Adapted to testing the input values can be grouped in equivalence classes and each class contains values that lead to the same output. If there is for example a program with one integer as input and there are values from 10 to 100 valid there are three equivalence classes possible: the first one with values less than 10, the second one with values from 10 to 100, and the third one with values greater than 100. It is also possible to group the first and the third, but often it is sensible to separate them, because they can lead to different program executions and errors. For testing there has to be one test case defined for each equivalence class and any value from the class can be used to represent it.

This strategy becomes more complex as soon as there are several input values and especially if the combination of the values is important. One example for an input with several values is a program expecting three integers and returning the maximum. All the possible combinations of values are interesting and therefore many test cases have to be created. If the combination of the values in irrelevant it is only necessary to have each of the equivalence classes for each single value represented in at least one test case, but it is not necessary to cover each combination. Only equivalence classes with values that are out of range have to be handled with special care. Such equivalence classes have to be tested separately that means that for all the other input values a value has to be selected that is in the specified range.

For object-oriented programs it is necessary to consider which states an object can have and which ones affect the programs execution. Therefore the definition of equivalence classes can become very complex for objects with many variables.

Random Testing

Random testing uses random input values for test cases. Myers stated that this method is probably the least effective methodology [Mye97].

There are publications addressing this topic and for example Duran and Ntafos examined how effective random testing is in real world examples [DN81]. They concluded that random testing can be cost effective for many programs and that it is especially sensible if it is combined with other techniques.

3.4.2 Structure-Based Testing

Structure-based testing uses the structure of the program as source for test cases instead of the specification [PY09]. Therefore there is knowledge about the implementation used and these tests are white box tests. Often a control flow graph is used and the inputs are selected in a way that a predefined coverages are reached. The common strategies are the coverage of all statements, conditions, paths, or branches. This is as well helpful to determine whether there is an adequate test coverage reached.

As source for structure based tests not only control flow charts usable, but as well other information about the structure, such as the partitioning of the layout algorithm in processors, as described in Section 2.3.2.

These tests should not be used as unique source of test cases, but it is sensible as addition to specification based tests. Generally the specification based tests are used as basis and the structure based tests are added.

3.4.3 Experience-Based Testing

Another commonly used source of test cases is the knowledge and experience of the tester [Mye97]. Often the testers gain the ability to guess probable sources of errors and some people become really good in that. This ability should be used in addition to specification and structure-based testing. It is difficult to define a procedure for this strategy. Myers suggested to create a list with possible errors or error-prone situations. This list can be used as orientation while the experience-based tests are created.

3.4.4 Test Coverage Measurement

The test coverage measurement can be used to evaluate whether there are enough tests in a test suites [Kle13]. All the tests are observed together and are evaluated regarding certain coverage criteria are met. For this evaluation it is sensible to use control flow graphs as this is done for the structure-based testing and the coverage criteria are in general also the same. There are tools available that measure the coverage. One example for such a tool is *EclEmma*[8] that shows for a programm after execution which lines of code were executed. This is interesting for the measurement of the statement coverage. Besides of that it is possible to measure the *method coverage*, the *decision or branch coverage*, and the *condition coverage*. The decision or branch coverage measures whether the `if(...)` statements have evaluated true and false and the condition coverage measures how many boolean sub-expressions of a conditional statement have been evaluated to true and false.

[8] `https://github.com/google/googletest`

Chapter 4

Testing Layout Algorithms

Concrete decisions on the testing strategy depend on the structure of the ELK project. In addition the development team and the project's organization and objectives have to be considered. This chapter provides an overview about the ELK project's situation and a proposal about the testing strategy.

4.1 Situation Assessment

The development of the ELK project is located at Kiel University. The developers are partly PhD students, partly employed at external companies, and partly bachelor or master students. The most active basis of the project consists of less than five developers. The development is mainly motivated by research and therefore not governed by the standard project workflow starting with a customer's order and ending with the delivery of the project. An already working version of the product is available for download and regular changes and releases are carried out. The project uses GitHub, Maven, and Hudson for continuous integration this purpose.

The whole source code is organized in Eclipse plugin projects and written in Java or languages that compile to Java code. The layout algorithms available in ELK at the time of writing are:

Layered The implementation of the layered based approach.

MrTree A layout algorithm that is specialized on trees.

Force and Stress Algorithms that use physical models to compute a layout.

4.1.1 Existing Test Infrastructure in the KIELER Project

As described in Section 2.2.4 the ELK project arose from the pragmatics part of the KIELER project. Therefore there are many similarities between the projects and parts of the KIELER project had been changed and adjusted in order to be integrated in the ELK project. One fundamental part of the code that had to be adjusted was the data type used to represent the graphs. Some test runners had been integrated in the KIELER project. They are introduced together with some implemented tests in this section.

One of the implemented JUnit runners pictured in Figure 4.1 is the `ModelCollectionTestRunner` that extends JUnit's `Suite` runner. This runner executes test methods on several input models stored in a model database. It can not only be used to test a layout algorithm, but to test every class or method with a model as input. A test class that should be executed with this runner needs to be annotated with `@RunWith(ModelCollectionTestRunner)` and has to provide the input graphs or the location of the graphs.

The test runner provides annotations that have to be used by the test classes in order to provide the information related to the models. Models that should not be imported by the test runner can be directly provided by a method in the test class and have to be tagged with the `@Models` annotation.

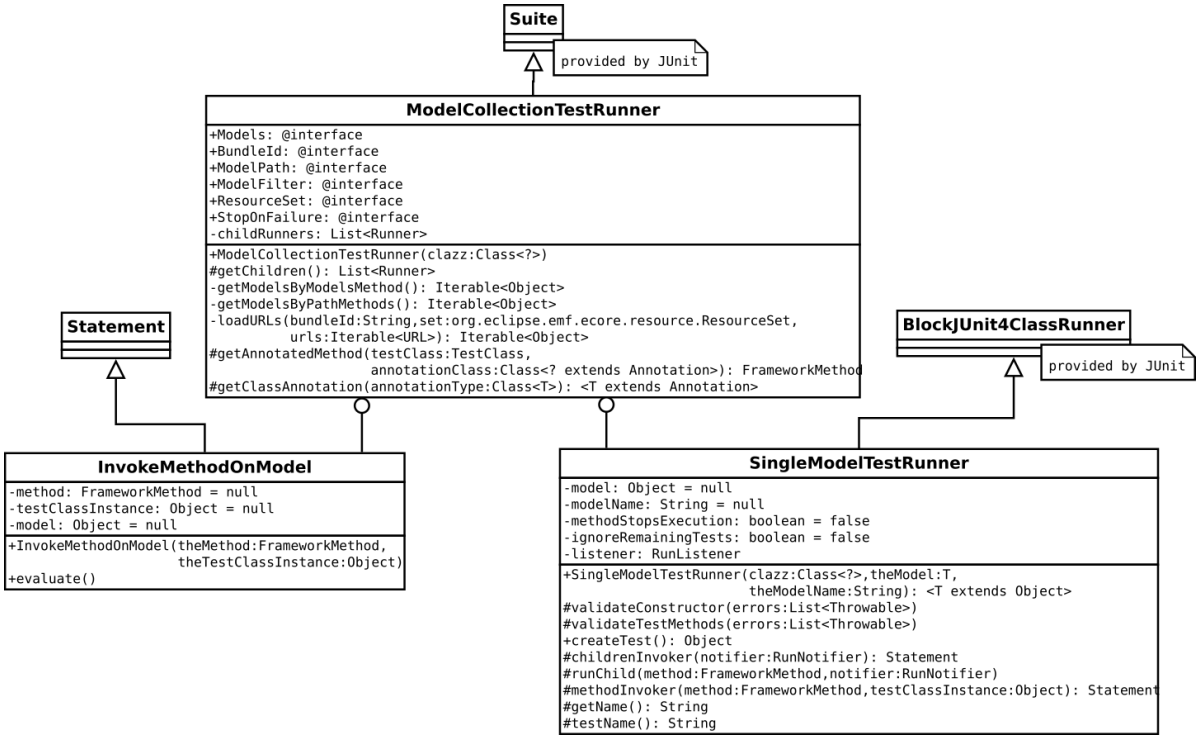


Figure 4.1. The ModelCollectionTestRunner

In order to do this a method has to be implemented that is annotated with `@Models` and returns the models in an `Iterable<Object>`.

The other possibility is to provide information that have to be used by the test runner for the import of the models. First of all the runner needs information about the location of the models. This has to be specified with the annotations `BundleId`, `ModelPath`, `ModelFilter`, and `ResourceSet` that are provided by the `ModelCollectionTestRunner`. The first three annotations can be used as a method or as a class annotation and the `ResourceSet` annotation is only usable as a method annotation. The annotations `BundleId` and `ModelPath` are mandatory for the specification of a location while the other two are optional. The `BundleId` annotation is used to specify the bundle in which the models are stored. The term *bundle* can be used interchangeably with the term *plugin* and is a unit of modularization. In general a bundle is a self-contained unit with defined dependencies to other components. The `ModelPath` annotation is used to specify the path of the models. If the path ends with `/**` or `/**/` the runner additionally looks for models in sub-directories. The model filter can be specified by a regular expression. One possible use case for the filter is to restrict the import to `.kgt` files stored in the specified directory. The `ResourceSet` annotation can be used in order to specify the resource set that should be used to import the models. During the execution of the tests the models are passed to the tests by a constructor of the class with one parameter or by a parameter of the test methods. Therefore either the constructor or the test methods have to have one argument.

Another feature of the test runner is the annotation `@StopOnFailure` that causes the runner to skip the remaining test methods waiting to be executed with the current model and to continue execution with the next model. This can be useful, if the execution of the remaining tests is no longer sensible after the annotated test fails.

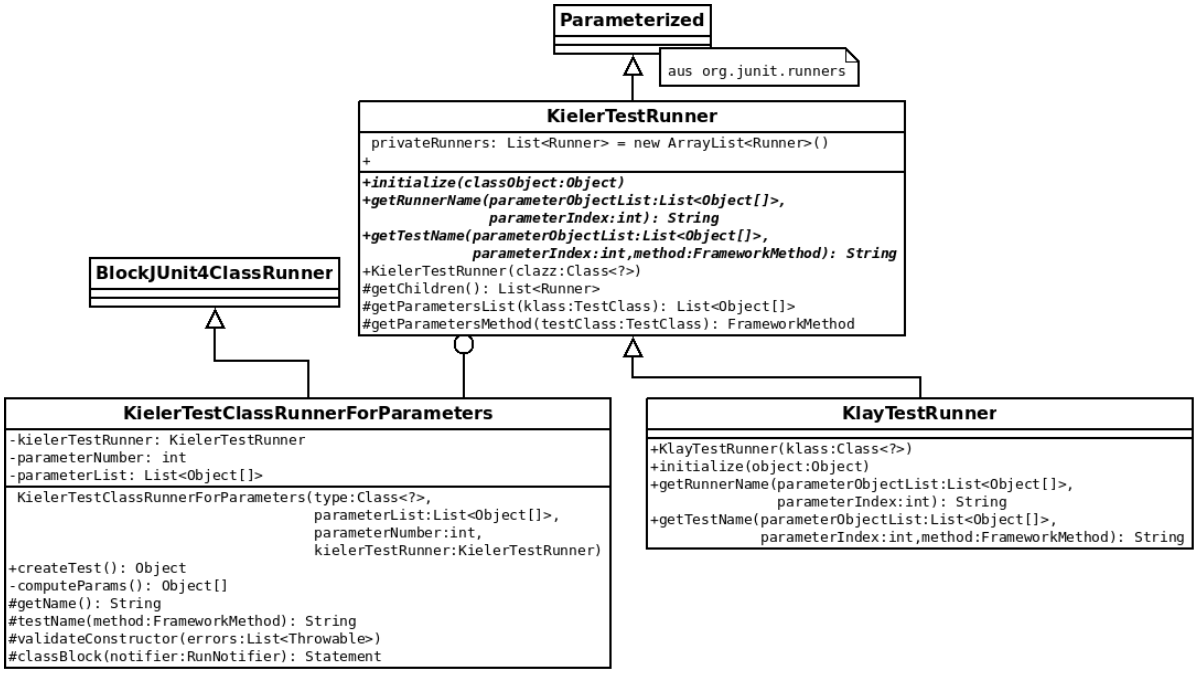


Figure 4.2. The KielerTestRunner and the KlayTestRunner

Internally the test runner uses the `SingleModelTestRunner` to execute the tests with a model. This runner is implemented as an inner class of the `ModelCollectionTestRunner`. It creates an instance of the test class and executes the test methods on the model.

Another test runner implemented in the KIELER Project is the `KielerTestRunner` that is pictured in Figure 4.2. It is a runner implemented for the purpose of layout algorithm testing in the KIELER project. This runner extends JUnit's `Parameterized` runner that provides the annotation `@Parameters`. This annotation has to be attached to a method in the test class with the return type `List<Object[]>`. Each array of objects in this list is one set of parameters used as test input for the execution of each test method in the test class. The parameters are passed to the test class through the constructor of the class.

Like the `ModelCollectionTestRunner` this class uses an inner class in order to execute the test methods, the `KielerTestClassRunnerForParameters`. The `KielerTestRunner` is an abstract class that has to be extended to implement a test runner for a concrete layout algorithm or a set of algorithms. An example for such a test runner is the `KlayTestRunner` pictured in Figure 4.2. This runner is specialized on the execution of a `KlayAutomatedJUnitTest` and calls the initialization method of this test to trigger the test class to initialize the list of parameters. A test executed by this runner has to extend the `KlayAutomatedJUnitTest` and does not need to be annotated with `@RunWith(KlayTestRunner)`, because the `KlayAutomatedJUnitTest` is already annotated accordingly.

The `KlayAutomatedJUnitTest` is an abstract class that helps to write tests for several executions of a KLay layout algorithm. The test holds a list of `GraphTestObjects` and a list of `ILayoutConfigurators`. The `GraphTestObjects` hold the location of a graph and the test runner uses a helper class that imports the graph and adds it to the `GraphTestObject`. An `ILayoutConfigurator` has the purpose to add options to a graph, such as the spacing between its components. An extending test class has to implement methods returning the configurators and locations of graphs. The `KlayAutomatedJUnitTest` combines each of

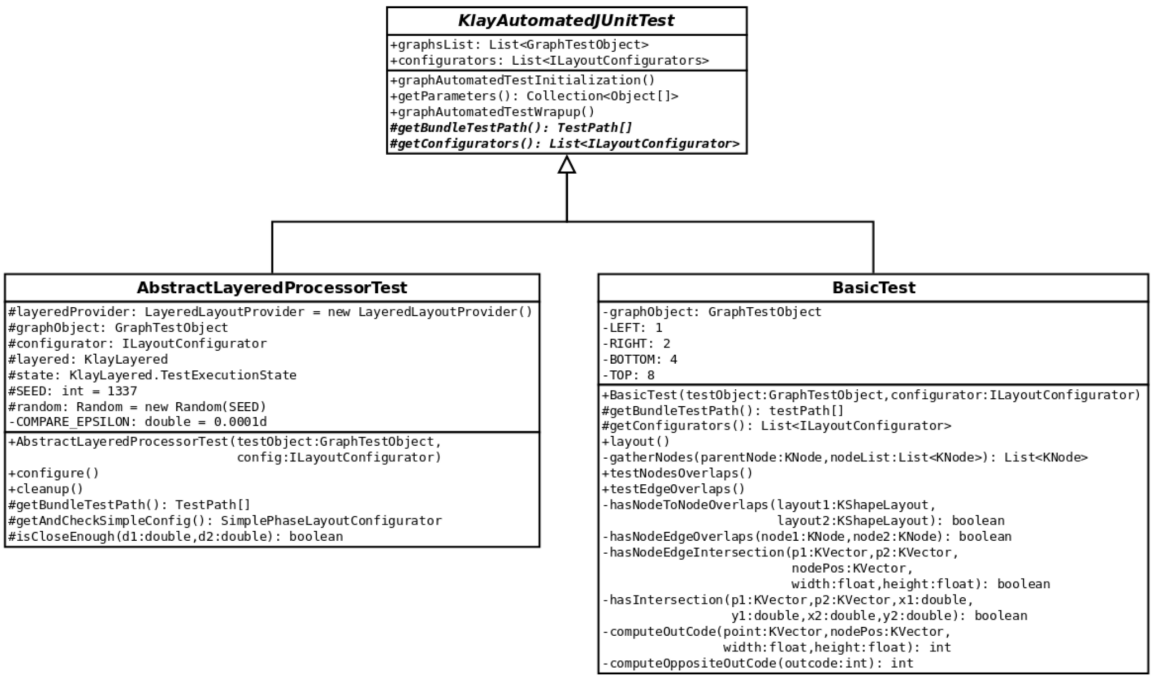


Figure 4.3. Tests for the execution with the KlayTestRunner

the graphs with each of the configurators separately and returns these combinations as parameters to the test runner.

The graphs used for the tests are specified by a path that has to be relative to the repositories root folder of the Git repository that contains the test class. Therefore there is no possibility to load graphs out of another repository. The formats of graph files are limited to `kgraph`, `kgx`, and `kgt`. It is possible to specify whether graphs should be imported from sub-directories and which sub-directories should be excluded. This is done in another way than it is done in the `ModelCollectionTestRunner`. Here, these values are set in variables of the class `TestPath` and not with the help of annotations in the test class. Furthermore the bundle identifier is not specified and whether to load the sub-directories is specified directly with a variable and not indirectly over the suffix of the pathname.

One example for a test extending `KlayAutomatedJUnitTest` is the test class `BasicTest`, pictured in Figure 4.3. This test is executed on a list of graphs and no configurators. Therefore a dummy configurator is used that does not change any attributes of the graph. The test graph is passed to the test class through the constructor and stored in a private variable. This makes it possible to use the graph already in a method annotated with `@Before` in which the layout of the test graph is performed. The other methods in the class are either test methods or helper methods. The test methods verify whether nodes overlap with other nodes or edges.

Another class extending the `KlayAutomatedJUnitTest` is the `AbstractLayeredProcessorTest` that tests the layered layout algorithm before or after a specified part of it is executed. The layered layout algorithm is structured in phases and intermediate processors, as explained in Section 2.3.2, and it provides the possibility to execute the layout algorithm until a specified point of the algorithms execution is reached. This point can be specified before or after a specified phase or processor is executed. If the execution of the algorithm is paused and there is a processor that waits to be

executed next, there is the possibility to execute only this processor. An example for a test extending the `KlayAutomatedJUnitTest` is the `InvertedPortProcessorTest` that executes the algorithm until the `InvertedPortProcessor` finished execution and tests the graph after that processor.

The main features implemented in these test runners have to be supported by the new test framework as well:

- the possibility to execute tests on several imported or otherwise provided graphs
- the execution of tests before or after phases or intermediate processors
- the execution of tests after a completed layout run

Therefore there should be the possibility to migrate the old tests on layout algorithms to the new framework.

4.2 The Testing Strategy

In this section a strategy for the testing in the ELK project will be derived. This will be based on the following questions:

- What should be tested?
- What kind of tests should be executed?
- How should the test cases be derived?
- How and when should the tests be executed?
- How should the results be presented and processed?

With the knowledge about the project's team, structure, and source code the testing can be planned. Normally the creation of the test plan should be done very early during the testing process and the main questions should be answered together with the development of the general plan of the project's workflow. After that is done the different types of tests such as unit, integration, and system test have to be planned. As soon as all these tests are planned, executed, and the errors are removed the product can be delivered. After that the developers are only responsible for the care and maintenance of the product and sometimes a new version of the software is planned. In the ELK project the development of the software has other motivations and goals. There are no customer's orders specifying the project's objectives, but there are new research results and publications that are interesting to the project, and there is room for improvement discovered by the developers. These newly added features have to be tested together with the existing code. That answers the first question: *What should be tested?*

The next interesting question is: *What kind of tests should be executed?* The added features can be viewed as part of the system and their behavior should be tested embedded in a whole layout run. The tests examined here are not whole system tests, but tests verifying the *automatic layout algorithms*. In order to use ELK for the layout of graphs, the easiest solution is to use the `DiagramLayoutEngine`. The `DiagramLayoutEngine` uses a class implementing the `IGraphLayoutEngine` interface in order to perform the actual automatic layout on the graph. The `RecursiveGraphLayoutEngine` is such an engine and can consequently be used to test the automatic layout. A test that verifies the output of this engine can be seen as a subsystem test with the engine as subsystem of the whole system provided by ELK. The engine is normally used in order to execute the layout algorithms and therefore they are executed in the usual way. This leads to more meaningful tests and therefore this solution is interesting for the

framework. If the test only verifies the output of the engine, it can be considered as a black box test. If there are tests executed after parts of the layout algorithm are executed they can be seen as integration tests. They test the integration of the parts of the algorithm in the subsystem. These kinds of tests will be explained below in Section 4.2.2 and Section 4.2.1.

Both kinds of tests are normally executed each time a new feature is integrated. In addition tests are executed in order to verify that old features still work in the designated way. These tests are *regression tests* and can be executed as black box and white box test.

Another kind of tests that should be executed are *analysis tests*. These tests are used in order to ensure the constant quality of the layout results. These tests are explained in Section 4.2.3

Now that the kinds of tests are known the next question is: *How should the test cases be derived?* In order to derive test cases the specification of the implemented system can be used. This is called *specification based* testing. The tests verify that the features are implemented in the specified way. For this purpose test cases with different inputs and verification methods are implemented. How this can be done will be discussed in Section 4.2.4.

With the test cases ready the next step can be planned: *How and when should the tests be executed?* To manage the source code and build the software, GitHub and a continuous integration tool are used. Therefore the tests pushed on the master branch can be executed automatically every time the build of the software is triggered. In addition to that the possibility to comfortably execute the tests on the local machine is important in order to find bugs before the software is integrated into the master branch. The execution of the tests on a local system contains more and other options for feedback.

After the execution of the tests just one more question has to be considered: *How should the results be presented and processed?* This obviously depends on the platform the tests are executed on. If the tests are executed on a local machine the developer who executed the tests has to take care of the removal of the error, or has to report the error, if it is located in a part of the system developed by another developer. Any errors that occur during testing on the build server have to lead to a failed build. That makes the build system report the failure to the responsible developers and the error or failure message is logged in the build log. The developers notified by the build system have to take care of the removal of the error.

4.2.1 Black Box Tests

In general a black box test can be written without any knowledge about the implementation of the system. Therefore usually only the input-output behavior of the system is verified. In most of the projects the information about the valid input and expected output is documented in the specification of the system.

In this case the automatic layout subsystem of the ELK project is inside of the black box and a graph with the contained configurations is the input. After the layout is done according to its configuration it is returned with attached layout information. The layout algorithm that should be used is specified in the input graph and a black box test can be used in exactly the same way for every layout algorithm.

Some of the old tests are written in the way of a black box test. The `BasicTest` explained in Section 4.1.1 executes the layout algorithm on the graph and tests the output for overlaps.

The example in Figure 4.4 shows the execution of a black box test. The graph with its attached configuration is laid out with the layout algorithm and the resulting graph is verified by the test methods.

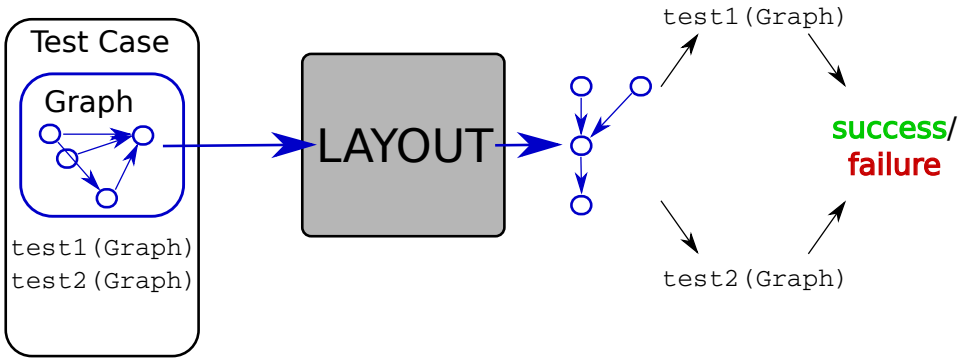


Figure 4.4. An example for a black box test

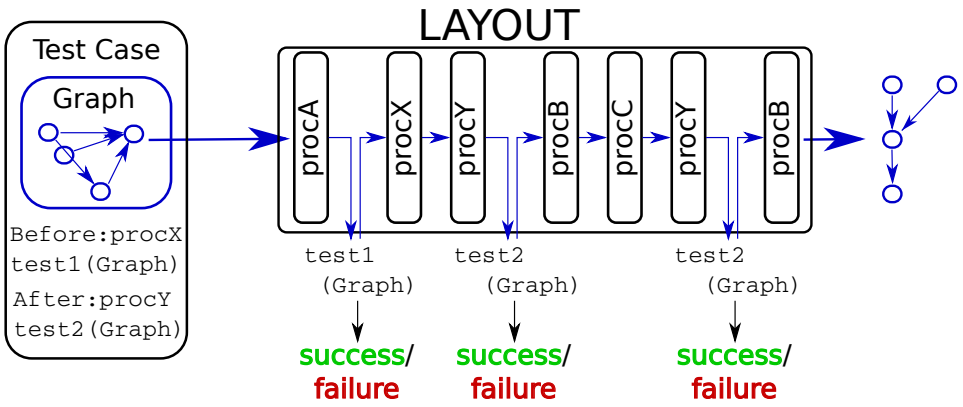
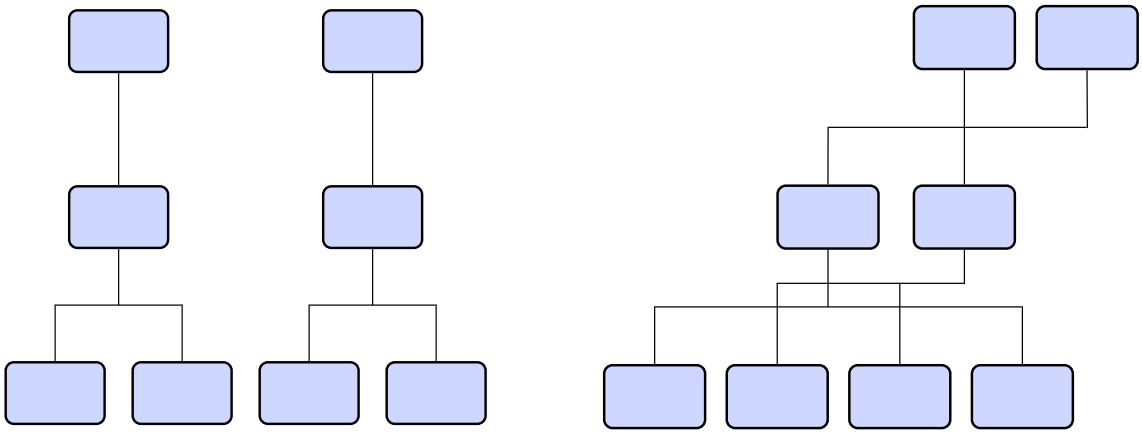


Figure 4.5. An example for a white box test

4.2.2 White Box Tests

A white box test is a test designed with knowledge about the system. For a white box test in the context of this work not the knowledge about the whole implementation of a layout algorithm is used, but only the knowledge about the division of the layout algorithm in phases and intermediate processors as explained in Section 2.3.2. With a white box test in this context the input and output of the processors should be verified. Therefore the test cases have to specify before or after which processors they have to be executed. This is helpful in order to test the processors integrated in the whole layout algorithm. Support for white box tests has to be provided by the tested layout algorithm and is therefore not possible for every new algorithm out of the box, in contrast to black box tests. The possibility for white box tests can just be integrated in layout algorithms using the structure explained in Section 2.3.2. Figure 4.5 shows the execution of white box tests. The test case has the blue graph as input and two test methods that are executed with the current graph. The first method should be executed each time before the processor `procX` is executed and the second one after the processor `procY`. The same processor can be executed several times in the same test run and therefore the tests can be executed several times or not at all, if the processor is not executed.



(a) The output of the layout algorithm before the changes are pushed

(b) The output of the layout algorithm after the changes are pushed

Figure 4.6. An example for a case that should cause an analysis test to fail

4.2.3 Analysis Test

The purpose of an analysis in general is not the verification of the correctness of the results, as with white and black box tests, but to test the quality of the results. The black and white box tests give no indication as to the extent to which a layout algorithm fulfills the demands, but the answer is just *success* or *failure*. Often this characteristic is essential. If the tests are executed automatically on the build server, the build should fail and the responsible person should be notified.

An analysis test also have to result in a succeeded or failed test, if it should be executed together with the other tests. Because of that an analysis is used in tests not to evaluate every small change in the quality of the results, but to take care that the quality does not fall under a minimal acceptable value. These tests will succeed as long as the quality of the results is high enough and fail otherwise. Sometimes it cannot be avoided that the quality of the results falls a bit and this should not lead to failed builds. This should happen only if there is a change that cannot be tolerated. Therefore these tests cannot replace the normal analysis described above, but they can avoid that there are changes that lower the quality drastically and unnoticed by the developers.

Such an analysis test that results in the answer success or failure can be execute every time the other tests are executed. Their execution is also sensible on the build server. In addition to that the analysis should be executed manually to evaluate the quality of the layout more detailed and explicit.

The example in Figure 4.6 shows a graph before and after a change had been pushed on the master branch. The quality of the latter layout is not acceptable and therefore the builds should fail. If none of the normal tests fail, because the crossing minimization does not work properly, an analysis test evaluating the number of crossings should fail. This will lead to a notification of the developers and the bug in the layout algorithm can be searched and fixed.

4.2.4 The Test Cases

An important part of the whole testing process is the derivation of test cases. One part of a test case is the *set of input data* and the other is the *set of pass or fail criteria*. Often a basic set of test cases is derived from the specification. As already explained above the current version of ELK is continually extended by new features. The work on these is done in the form of bachelor, master, and doctoral theses. The

features are implemented because there is room for improvement, new research results are available, or interesting open research questions should be tackled. Therefore the basis for tests is quite special and the test cases can be derived out of other resources. For bachelor and master theses the problem that has to be solved should be discussed in detail with the supervisor and written down after that by the student. This written problem statement can be used as basis for the tests. Another possibility is the derivation of test cases out of publications used as basis for the implementation. Tests derived like that are called *specification-based functional tests*.

Another relevant kind of test case is the one derived from failures. Often after a failure has been found and solved it is sensible to add appropriate test cases to the set of automatically executed test cases. That makes sure that a similar error will be found automatically the next time.

The input data for a layout algorithm test case is a graph. Basically a graph consists of a set of nodes and edges. In this case there can exist *ports* and different kinds of *labels* as well. All these elements have a position and other information attached to them, such as a size. In addition to that there can be layout options attached to the whole graph or parts of it. The layout options are used to configure the used layout algorithm. Another important aspect are the graph theoretical properties. For some layout algorithms for instance it can be interesting, if the graph contains cycles or is a tree or a planar graph. For the derivation of specification based test cases, equivalence classes can be used as explained in Section 3.4. A graph with its components, options, and properties can be treated similar to an object with its values. For a complete set of test cases it would be necessary to derive for each relevant aspect the equivalence classes and to combine them in the right way.

The combination of all these aspects leads to many test cases. Hence there are many graphs required in order to test the layout algorithms properly. In general graph algorithms should be tested with real world examples. For the ELK project that can be amongst others graphs derived from SCCharts models. If there are not enough real world examples with the needed properties available, a *random graph generator* can be used to generate additional graphs. This is very useful, but these graphs are not always that suitable and should only be used additionally [PTK16].

Chapter 5

The Test Process

This chapter describes how the layout algorithms available in the ELK project can be tested with the new framework. It is explained how the test classes should be implemented and how and when they are executed. An example test constructed in the progress of this chapter illustrates how a test can be designed. While this chapter surveys the design of test cases from the test engineer's perspective, the next chapter will describe how the features provided by the framework are implemented and why they are selected and implemented in that way.

5.1 The Design of Tests

In general the purpose of a test is the verification of features provided by one or more layout algorithms. In addition to that it is possible to verify that the behavior of a part of a layout algorithm is implemented as specified. Another kind of tests are the analysis tests that are not used in order to test the features, but to ensure an acceptable quality of the results.

The first property of the test to be specified is the tested layout algorithm and, optionally, the part of it that should be tested. Next, the test input has to be determined and after that the test methods have to be designed. With these components the test class is nearly complete. In addition it is possible to adjust the output of the test results.

Basically the attributes of a test class are:

(Optionally) The Layout Algorithm The layout algorithm that should be tested. It is possible to specify several algorithms. If no algorithm is specified the default one is used or it has to be set using properties on the graphs.

(For a white box test) The Processors The part of the algorithm the test should be executed before or after.

The Graphs The graphs that should be used as input for the layout algorithm.

(Optionally) Configurators The configurators can be used to configure the graphs.

The Test Methods The test methods are used to verify the output of the layout algorithm.

The attributes listed above are explained in detail in this chapter.

The framework is designed in a way that annotations are used in order to specify the test cases and configure the test execution. In this chapter the annotations that have to be used in order to specify a test case are explained. The first annotation that has to be used in order to execute a test class with the framework is the `@RunWith` annotation. Each test class that should be executed by the framework has to be annotated with `@RunWith(LayoutTestRunner)`.

In order to execute the tests JUnit is used. Like the annotations `@RunWith` the other annotations provided by JUnit are available in addition.

```
@RunWith(LayoutTestRunner.class)
@Algorithm(LayeredOptions.ALGORITHM_ID)
public class NodesOverlapElkTest{
...
```

Algorithm 5.1. The beginning of the NodesOverlapElkTest.

5.1.1 The Tested Algorithm

The tested layout algorithm can be specified with a class annotation, the `@Algorithm` annotation. In most cases a test class is created in order to test one layout algorithm. This algorithm is specified in the class annotation and each node of graph is configured with this algorithm as property. Therefore every hierarchy level of the graph is laid out by the specified algorithm. The test methods are invoked on the graphs after they are laid out by the algorithm.

Some tests are sensible for more than one algorithm, as for example a test that verifies that no edges overlap with nodes. For these tests more than one algorithm can be specified. In order to do that the `@Algorithm` annotation is used repeatedly with each tested layout algorithm. Another possibility is to test all layout algorithms known by the framework. This can simply be done by the usage of the `@Algorithm` annotation with `"*"` instead of an algorithm identifier.

It is also valid to specify no algorithm by omitting the annotation. This is sensible if different parts of the graph should be laid out by different layout algorithms. How this can be done is explained shortly in Section 5.1.2.

The example test class constructed during this chapter is a test class that verifies that no nodes overlap with other nodes or edges after a graph has been laid out by the `ElkLayered` algorithm. Its declaration is illustrated in Algorithm 5.1.

5.1.2 The Test Input

As explained in Section 4.2.4, one part of each test case is the test input. This input is basically a graph that has to be laid out by the tested layout algorithm. It has been explained additionally that not just the basic components of the graph, such as nodes and edges, characterize a graph but also the options specified for each graph element. In ELK there is the possibility to use a `LayoutConfigurator` in order to configure a graph. In this configurator options can be specified and by applying this configurator on a graph these options are set on the graph. Examples for the options are the layout algorithm that should be used to layout the graph or a part of it, the spacing between a node and another node, or the way the ports of a node should be placed.

The framework provides the possibility to specify graphs and configurators as test input. Each of the graphs is combined with each of the configurators in order to derive the test cases out of the test classes.

Graphs

The first part of the test input is the graph itself. There are different possibilities available to specify a graph.

For all these possibilities annotations are used. The annotations can be used several times in every test class in order to test the layout algorithm with an appropriate set of input graphs.

The possibilities are:

Create a Graph in a Method Implement a method in the test class that returns a graph.

Import a Graph Import a graph that is stored in a file.

Generate a Graph Create a graph using the `RandomGraphGenerator`.

Create a Graph

If a small or very specific graph is needed it is sensible to create the graph in a method in the test class. This method has to be annotated with `@Graph` and has to return an `ElkNode`. The advantage of this method is that the graphs can be created exactly in the way they are needed and therefore a minimal set of graphs with minimal size can be used. Especially if these graphs are not needed for other tests it is more sensible to create the graph in the method instead of a file.

Import a Graph

The most comfortable way to specify the input graphs is to import them from a file. Graphs can be stored in files and can be used by as many tests as wanted. The files can be created manually or exported. The manual creation of a graph allows to create graphs with special properties that can be used in order to illustrate a special problem. This is especially useful if there are faults found and test cases are created in order to ensure that the failure will be found in latter implementations. Graph files can be exported out of examples created in projects using ELK, such as the SCCharts project. This leads to more realistic test input representing real problems.

These stored graphs can be used for several tests and can be managed in a way that there are *standard graphs* for all algorithms or for a particular one. One possibility is for example to create a folder with the test graphs that should be used for all tests of the layered algorithm. If these graphs should not be specified in every layered test it is possible to create an abstract class `LayeredTest` that specifies the directory as test input and that the tests testing this algorithm should extend.

The graphs that should be imported are specified by the usage of the annotation `@ImportGraphs` that can be attached to a method or a field. The return type of the method or the type of the field has to be `List<ResourcePath>`. The class `ResourcePath` is used in order to specify a single file or a set of files located in a directory. If it is used to specify a directory in which the files are stored there is the possibility to look for files in the sub-directories as well and to specify a filter used to reject files that should not be used as input.

The `ElkRepositoryResourcePath` and the `ModelsRepositoryResourcePath` are two examples for classes extending `ResourcePath`. The class `ElkRepositoryResourcePath` has to be used in order to specify the location of a graph stored in the ELK repository and the class `ModelsRepositoryResourcePath` for graphs stored in a models repository. There was a models repository available in the KIELER project, but the graphs stored in this repository do not have a format that can be imported as `ElkNode`. Most likely there will be such a repository available in the ELK project in the future and for this purpose the class is created. In order to use one of the extensions of `ResourcePath` it is necessary to set the relating system property or environment variable. `MODELS_REPO` needs to be set to the location of the models repository and `ELK_REPO` to the location of the ELK repository. The reason why it is implemented in this way is described in Section 6.2.1.

If a graph is imported out of an `.elkt` file there may be some properties of the graph elements be missing. Therefore the basic properties of these elements are automatically set to default values, unless explicitly requested otherwise. The components that are initialized with default values are the nodes, ports, and edges. The values set for all the nodes are a size, a label based on the identifier and a strategy for the label placement. For a port the size and a label are set and for an edge the label

```
@RunWith(LayoutTestRunner.class)
@Algorithm(LayeredOptions.ALGORITHM_ID)
//use no default configurations for the edges
@UseDefaultConfiguration(edges = false)
public class NodesOverlapElkTest{

   //A special graph to test that there are no node edge overlaps.
   @Graph
   public ElkNode getSpecialGraph(){
      ElkNode layoutGraph = ElkGraphUtil.createGraph();
      ElkNode node1 = ElkGraphUtil.createNode(layoutGraph);
      ElkNode node2 = ElkGraphUtil.createNode(layoutGraph);
      ElkNode node3 = ElkGraphUtil.createNode(layoutGraph);

      ElkGraphUtil.createSimpleEdge(node1, node3);
      ElkGraphUtil.createSimpleEdge(node1, node2);
      ElkGraphUtil.createSimpleEdge(node2, node3);
      return layoutGraph;
   }

   //The cycle free graphs representing sccharts that should be used for testing
   @ImportGraphs
   public List<ResourcePath> getImport(){
      List<ResourcePath> list = new ArrayList<>();
      FileExtensionFilter filter = new FileExtensionFilter(".elkg");
      list.add(new ModelsRepositoryResourcePath("graphs/cycle-free/sccharts/**")
                                                .withFilter(filter));
      return list;
   }
...
```

Algorithm 5.2. The `NodesOverlapElkTest` with a graph created and graphs imported.

placement strategy. It is possible to turn the default configuration off for the components separately by the usage of the class annotation `@UseDefaultConfiguration`. This annotation has the three attributes `nodes`, `ports`, and `edges` each with the default value `true`. If the attribute related to a type of graph element is `true`, this element will be configured with default values. Therefore the annotation has to be used only to turn the default configuration off. This is done for the edges in Algorithm 5.2. In addition there is always a size calculated for labels that contain a text and have a size of zero. This can not be prevented.

The `NodesOverlapElkTest` example is extended by the creation and importation of graphs in Algorithm 5.2. In this example the edges of the graphs should not be configured with default configurations. Figure 5.1 shows how the test cases are derived in this example. In the folder are not just files that can be used as input for the test and therefore the test uses a filter to import only the files with the extension `.elkg`. The test methods implemented in `NodesOverlapElkTest` are executed four times, once on each graph.

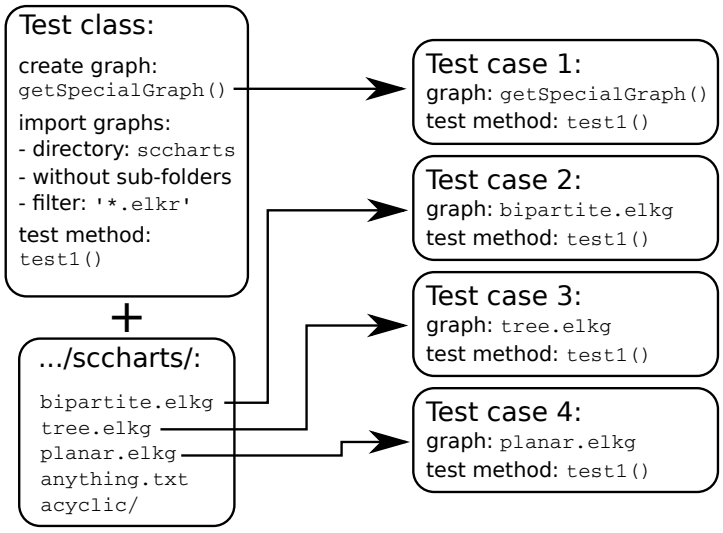


Figure 5.1. An example for the creation and import of graphs.

Random Graph Generator

If the graphs do not need to be very specialized and if no or not enough appropriate files are available, the *RandomGraphGenerator* can be a good alternative. One possibility to use the random graph generator is the usage of an `.elkr` file. In this file the graphs that should be created can be specified. One very small example is "create 2 trees". The properties of these trees can be specified in the file as well. It is for example possible to specify the approximate number of nodes with a Gaussian distribution or to instruct the generator to create node labels. The location of this file can be specified with a `ResourcePath` and the related annotation `@RandomGraphFile` has to be used.

The other possibility is to create an instance of `GeneratorOptions` together with the annotation `@RandomGeneratorOptions`. This class is used to hold the options for the generation of the graphs and can be configured as necessary.

The usage of the random graph generator is useful to create many graphs as input data without much work. The random graph generator is especially useful to test that the layout algorithm can handle an adjustment of a characteristic that can be specified in the file or generator option. One drawback is that the generated graphs might not be realistic. On the other hand that can be viewed as an advantage, because it ensures a higher flexibility and not a design that works only for the projects already using ELK. Another drawback is that the results gained with random graphs are not reproducible and comparable to later test runs.

In Algorithm 5.3 the continuation of the example it stated. In each example the methods showed in the examples before are omitted to gain a better readability. The two different possibilities to generate graphs are used in the example. Assuming that the referenced `.elkr` file causes two graphs to be generated, there are three graphs added to the four from the last example. Especially if there are many graphs imported and options for many graphs specified in `.elkr` files the number of graphs can grow very high. In later examples the `@GeneratorOptions` will not be used in order to keep the examples short.

```
@RunWith(LayoutTestRunner.class)
@Algorithm(LayeredOptions.ALGORITHM_ID)
...
public class NodesOverlapElkTest{

        ...

   //The options set for a generated graph
   @RandomGeneratorOptions
   public GeneratorOptions getGeneratorOptions() {
      GeneratorOptions options = new GeneratorOptions();
      options.setProperty(GeneratorOptions.NUMBER_OF_NODES, RandVal.minMax(15, 30));
      options.setProperty(GeneratorOptions.OUTGOING_EDGES, RandVal.minMax(1, 3));
      return options;
   }

   //The generation of cycle free graphs
   //(as long as there are not enough examples for cycle free graphs)
   @RandomGraphFile
   public ResourcePath getRandomFilePath(){
      FileExtensionFilter filter = new FileExtensionFilter(".elkr");
      return new ModelsRepositoryResourcePath("graphs/cycle-free/**").withFilter(filter);
   }
...
```

Algorithm 5.3. The `NodesOverlapElkTest` with created, imported, and generated input graphs.

Configurators

As mentioned above a `LayoutConfigurator` can be used in order to set the options of graph elements. The first possibility to specify a configurator is the usage of the annotation `@Configurator`. This annotation can be used as method annotation to tag a method returning a configurator. This configurator will be applied to configure the whole graph. Sometimes it is necessary to configure a graph in a more flexible way, as in the example mentioned above: sometimes a graph contained in a hierarchical node should be laid out with a different algorithm than the root graph. Therefore the same layout algorithm can not be set for all the nodes. To configure a graph in this way is not possible with the helper method that is used by default to apply a specified configurator and therefore a method has to be provided that configures the graph in the needed way. A method like that has to be annotated with `@ConfigFacility`. This is the second possibility to specify a configurator.

If the configurator in Algorithm 5.4 is applied on the graph, the option `SPACING_NODE_TO_NODE` is set to `30.0` for each node of the graph. The configuration facility method in that example sets different layout algorithms for the child nodes. If these child nodes contain graphs these are laid out by different algorithms. Each of the graphs from the last examples is combined with the configurator and the configuration method separately and therefore there are 12 test cases generated for this test class.

It is possible to use the annotations several times and consequently every graph can be configured in different ways by the different configurators. Every configurator and configuration method is combined with each graph separately, as shown in Figure 5.2.

```
1  @RunWith(LayoutTestRunner.class)
2  @Algorithm(LayeredOptions.ALGORITHM_ID)
3  ...
4  public class NodesOverlapElkTest{
5
6      ...
7
8      //Set the node to node spacing to 30.0 for all nodes
9      @Configurator
10     public LayoutConfigurator getConfigurator() {
11         LayoutConfigurator layoutConfig = new LayoutConfigurator();
12         layoutConfig.configure(ElkNode.class).setProperty(CoreOptions.SPACING_NODE_NODE, 30.0);
13         return layoutConfig;
14     }
15
16     //Set the node to node spacing to 30.0 and for one node to 0.5
17     @ConfigFacility
18     public void setNodeSpacings(ElkNode graph) {
19         LayoutConfigurator layoutConfigActual;
20         int i=1;
21         LayoutConfigurator layoutConfigLayered = new LayoutConfigurator();
22         layoutConfigLayered.configure(ElkNode.class).setProperty(CoreOptions.ALGORITHM, LayeredOptions.
               ALGORITHM_ID);
23         LayoutConfigurator layoutConfigForce = new LayoutConfigurator();
24         layoutConfigForce.configure(ElkNode.class).setProperty(CoreOptions.ALGORITHM, ForceOptions.
               ALGORITHM_ID);
25         for (ElkNode elkNode : graph.getChildren()) {
26             if(i++%2 == 0){
27                 ElkUtil.applyVisitors(elkNode, layoutConfigLayered);
28             } else {
29                 ElkUtil.applyVisitors(elkNode, layoutConfigForce);
30             }
31             setNodeSpacings(elkNode);
32         }
33     }
34 ...
```

Algorithm 5.4. The `NodesOverlapElkTest` with the input graphs and configurators.

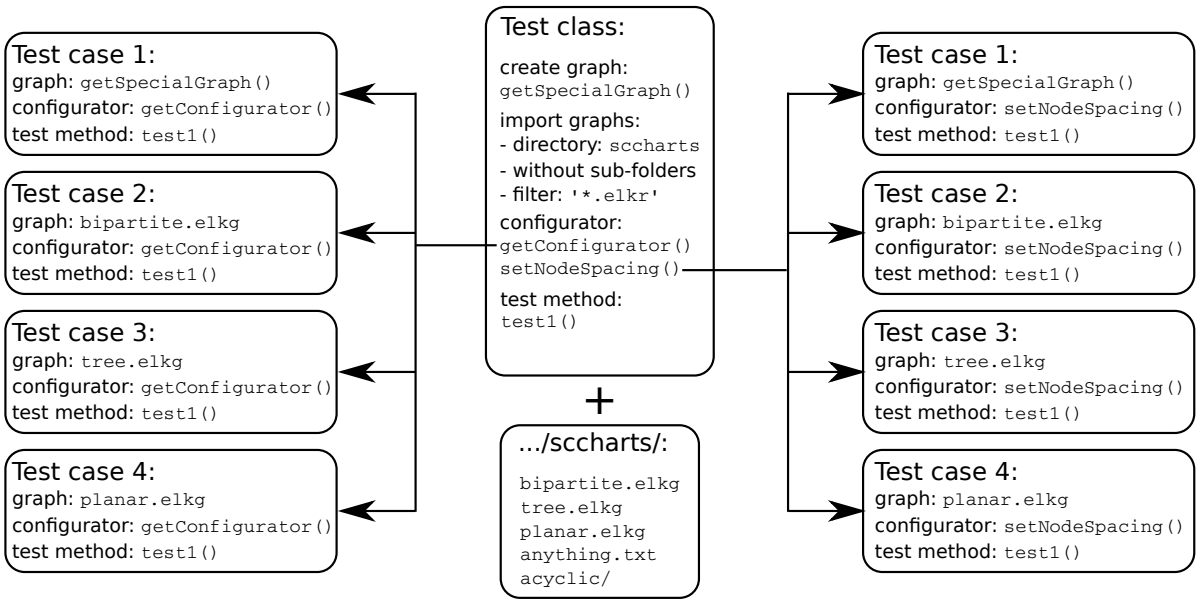


Figure 5.2. An example for the combination of configurators and graphs.

```
@RunWith(LayoutTestRunner.class)
@Algorithm(LayeredOptions.ALGORITHM_ID)
...
public class NodesOverlapElkTest{

   ...

   //Test that the nodes do not overlap with other nodes
   @Test
   testNodeToNodeOverlap(ElkNode graph){...}

   //Test that the nodes do not overlap with edges
   @Test
   testNodeToEdgeOverlap(ElkNode graph){...}
...
```

Algorithm 5.5. The `NodesOverlapElkTest` with the input graphs, configurators, and test methods.

5.1.3 Black Box Tests

A black box test is executed after the layout algorithm finished its execution. Hence the final layout of the graph can be verified.

How the test input and the algorithm can be specified had been explained. Consequently only the verification is needed in addition to define black box tests. In Algorithm 5.5 the example that had been constructed during this chapter is extended by two test methods that verify the output of the layout algorithm. The method `testNodeToNodeOverlap(ElkNode graph)` verifies that the nodes do not overlap with other nodes and the test method `testNodeToEdgeOverlap(ElkNode graph)` verifies that the nodes do not overlap with edges. The actual implementation of the test methods is not relevant to explain how tests are written with the framework and is therefore not shown.

The execution of two example test cases is illustrated in Figure 5.3. First of all the first test case is executed. In order to do that the input of this test case, the graphA, is laid out with the `ElkLayered` algorithm and the test method is executed with this graph. After that the second test case is executed and the methods are one after the other executed with the graph. After all the test methods are executed the output is combined. The test run fails, because one of the methods failed.

5.1.4 White Box Tests

A white box test is used in order to verify the preconditions and results of phases or intermediate processors the layout algorithm consists of. Therefore the layout algorithm has to be structured into layout phases and intermediate processors, as explained in Section 2.3.2. Both phases and the intermediate processors implement the interface `ILayoutProcessor<G>` and are therefore in the following text both referred to as *processors*.

As soon as a processor is specified that the test methods should be executed before or after, the whole test class is recognized as a white box test.

As a black box test the example test is complete like it is outlined in Algorithm 5.5. In order to change it to a white box test, the annotation `@RunBefore` or `@RunAfter` has to be used in order to specify a processor the test methods should be executed before or after, respectively. The result is shown in

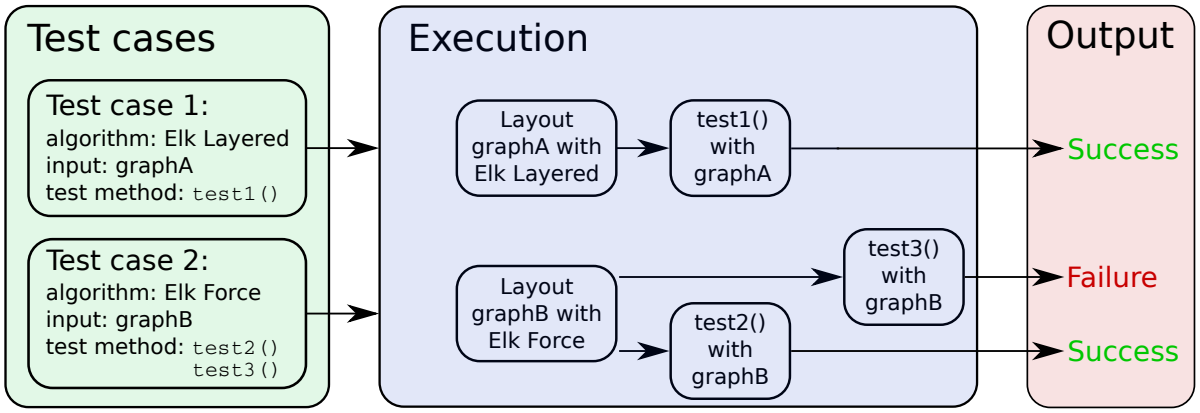


Figure 5.3. An example for the execution of black box tests.

```
@RunWith(LayoutTestRunner.class)
@Algorithm(LayeredOptions.ALGORITHM_ID)
...
//execute the test methods after the processorX had been executed
@RunAfterProcessor(processorX)
public class WhiteNodesOverlapElkTest{

   ...

   //Test that the nodes do not overlap with other nodes
   @Test
   testNodeToNodeOverlap(ElkNode graph){...}

   //Test that the nodes do not overlap with edges
   //This test method is executed only before the processorY
   @RunBeforeProcessor(processorY)
   @Test
   testNodeToEdgeOverlap(ElkNode graph){...}
...
```

Algorithm 5.6. The `WhiteNodesOverlapElkTest` with the input graphs, configurators, test methods and processors.

Algorithm 5.6.

There are two possibilities to specify the processor and both of them are used in the example Algorithm 5.6:

1. Specify the processor as class annotation.
2. Specify the processor as method annotation.

If the first possibility is used, all methods in the class are executed before or after the specified processor, except for the methods that are annotated specifically with one of the two annotations. Therefore the method `testNodeToNodeOverlap(ElkNode graph)` in Algorithm 5.6 is executed each time after `processorX` is executed. The annotation of a method is the second possibility that can be used

in order to execute only this method solely before or after the processors specified in annotation. Therefore the method `testNodeToEdgeOverlap(ElkNode graph)` in the example Algorithm 5.6 is executed each time before the `processorY` is executed. The annotations can be used several times. Accordingly the test methods can be executed before and after several processors, if for example the test methods can be used to verify a precondition that has to be valid for several processors.

Apart from the graph to run the test on, the test methods can optionally expect the processor they are executed before or after as a second parameter. If for example a test method's annotation specifies that it should be executed before `processorX` and after `processorY`, this parameter can be used to identify the processor that triggered the current invocation. There is also the possibility to specify that a test should just be executed before or after a processor that is executed on the highest hierarchy level of a hierarchical graph. This feature is not necessarily supported by every layout algorithm.

Summarized a black or white box test class that should be executed with the test framework has in general the structure illustrated in Figure 5.4.

5.1.5 Analysis Test

An analysis test is a test that verifies that the quality of the graph after layout does not fall below a minimal acceptable level. In order to specify this threshold, old results of the execution of the analysis with the same combination of algorithm, graph, and configurator are used together with a maximal acceptable deviations. An analysis test has to be a black box test.

One example for an analysis test is the verification that there are not too many bend points in a graph after the layout with `ElkLayered`. This example is outlined in Algorithm 5.7. A test like that can be designed like the test in Algorithm 5.4. It is not necessary to implement a test method if the test extends the abstract `AnalysisTest`, which implements a test method that compares the new and the old results. The test only has to define an *analysis configuration* to be used by this test method.

It makes no sense to use random graphs for analysis tests, because two random graphs are different and the results of the analysis can not be compared. Therefore an analysis test run on a random graph will result in a failed test. Because of that the random graph that has been used in the examples before is not used in this example.

The verification in `AnalysisTest` is specialized on analyses that return an array of values. This is done because it is the general case. The allowed derivations and the specifications of their relevance are specified in arrays as well. These arrays need to have the same length as the array returned by the analysis. The length of the result returned by the analysis is checked by the test and compared to the specified arrays.

The analysis configuration is used by the test method in the `AnalysisTest` to verify that the quality of the graph after layout is acceptable. The information that has to be present in the analysis is:

analysis The analysis that should be executed on the graph after layout.

oldResultsFile The path to the file that contains the old results for comparison.

allowedDeviationNeg The negative deviation that the values in the result are at most allowed to have in comparison to the old results. The value at index `i` is used to compare the value that is at index `i` in the analysis result with the value at index `i` in the old results array.

allowedDeviationPos The positive deviation that the values in the result are at most allowed to have in comparison to the old results.

The Test Runner
```
@RunWith(LayoutTestRunner.class)
```
The Algorithm(s) Under Test
```
@Algorithm(LayeredOptions.ALGORITHM_ID)
```
The Processors (For White Box Tests)
```
@RunBeforeProcessor(processor = MinWidthLayerer.class)
@RunAfterProcessor(processor = LongestPathLayerer.class)
```
Default Configurations
```
@UseDefaultConfigurations(edges = false)
```
public class ExampleElkTest{

Test Input:

The Graphs:
```
@Graph
public ElkNode getGraph(){...}
@ImportGraphs
public List<ResourcePath> getImportSources(){...}
@RandomGeneratorOptions
public GeneratorOptions getOptions(){...}
@RandomGraphFile
public ResourcePath getFilePath(){...}
```
Configurators:
```
@Configurator
public LayoutConfigurator getConfig(){...}
@ConfigFacility
public void configure(ElkNode graph){...}
```
Test Methods:
```
@RunBeforeProcessor(processor = MinWidthLayerer.class)
@RunAfterProcessor(processor = LongestPathLayerer.class)
@FailIfNotExecuted
@Test
public void testSomething(final ElkNode graph){...}
```

Figure 5.4. The structure of a test class

```
1  @RunWith(LayoutTestRunner.class)
2  @Algorithm(LayeredOptions.ALGORITHM_ID)
3  ...
4  public class BendsAnalysisElkTest extends AnalysisTest {
5
6      ...
7
8      //The configuration of the analysis
9       @AnalysisConfig
10     public AnalysisConfiguration getConfiguration() {
11
12         Class<? extends IAnalysis> analysis = BendsAnalysis.class;
13
14         int resultLength = 5;
15
16         Object[] allowedDeviationPos = { 3, 6, 10, 5 };
17         Object[] allowedDeviationNeg = { 0, 1, 2, 3 };
18
19         boolean[] isRelevantPos = { true, true, false, true, true };
20         boolean[] isRelevantNeg = { false, false, false, false, true };
21
22         ResultsResourcePath oldResultsFile = new ResultsResourcePath("analysis1.txt");
23         boolean shouldStore = true;
24
25         return new AnalysisConfiguration(analysis, resultLength, allowedDeviationPos,
26                 allowedDeviationNeg, oldResultsFile, isRelevantPos, isRelevantNeg, shouldStore);
27     }
28 ...
```

Algorithm 5.7. The analysis test with the input graphs, configurators, and the analysis configuration.

isRelevantNeg The boolean values in this array specify whether the deviation with the same index in `allowedDeviationNeg` is relevant. For example it is not sensible to specify a maximal acceptable negative deviation for the number of edge crossings and this should be marked as not relevant.

isRelevantPos The boolean values in this array specify whether the deviation with the same index in `allowedDeviationPos` is relevant.

shouldStore Specifies whether the results of an analysis should be stored as comparison values for subsequent tests in the `oldResultsFile`. If all the test cases of the analysis test are successful the results of the analysis can be stored in the file as new comparison values for the next test runs.

The abstract class `AnalysisTest` provides the facility to verify the results of the analysis. In order to evaluate whether an analysis test should fail, the results of the analysis executed with the graph after layout are compared with reference results stored in a file with old results. In this file the combination of the layout algorithm, the test graph, and the configurator is used as identifier for the result. If the file with the old result or the related old result in the file can not be found, the test will fail. For each test case the `AnalysisTest` looks for the related old result in the file and compares it with the new one using specified maximal derivation. If a value is marked as not relevant it is ignored and the verification continues with the next one.

Figure 5.5 shows the execution of Algorithm 5.7. The actual test method is implemented in `AnalysisTest` and therefore `edgeCrossingAnalysisTest` is a test class without a test method.

5.1.6 Execution of Several Test Classes

In many settings and especially on the build server it is realistic that not only one test class should be executed, but several tests or all tests that are available.

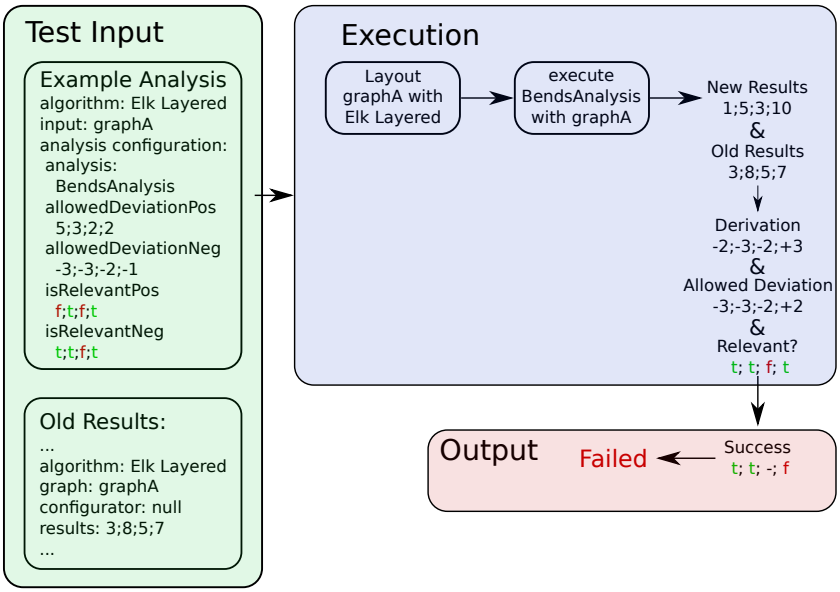


Figure 5.5. An example for the execution of an analysis tests.

```
1 @RunWith(LayoutTestRunner.class)
2 @TestClasses({NodesOverlapElkTest.class, WhiteNodesOverlapElkTest.class, BendsAnalysisElkTest.class})
3 @TestPath("test")
4 public class TestConfig{}
```

Algorithm 5.8. An example for a TestConfig class.

This can be done with the help of an extra test class that is called `TestConfig` in the following. This class is the holder for class annotations that can be used to specify the tests that should be executed. The available annotations for this purpose are:

`@TestClasses` The test classes can be explicitly listed with the `@TestClasses` annotation.

`@TestPath` A path to the `.class` files of the test classes can be specified with the `@TestPath` annotation.

These two annotations can be used by themselves or be combined. In addition to these annotations the class needs to be annotated with `@RunWith(LayoutTestRunner)` and it is possible to configure the output behavior for the test run. That is explained in Section 6.7.

Algorithm 5.8 shows an example for a `TestConfig` class. If JUnit is invoked with this class the test cases specified in `NodesOverlapElkTest`, `WhiteNodesOverlapElkTest`, and `BendsAnalysisElkTest` are executed together with the ones defined in all `.class` files that are located in the ELK repository in the folder `test` or one of its sub-folders. The naming convention used for the gathering of the tests in a folder is that the name of a test class has to start or ends with `ElkTest`. Therefore every test that should be executed on the build server should have a name that has this property.

One advantage of the usage of a `TestConfig` class is that the test cases can be grouped. How this is done internally is explained in Section 6.8.

```
@RunWith(LayoutTestRunner.class)

@StoreResults(true)
@StoreFailedGraphs(true)


@TestPath("test")
public class TestConfig{}
```

Algorithm 5.9. An example for a TestConfig class that configures the test run in a way that the test results are stored.

5.1.7 Manual and Automated Execution

The manual execution in Eclipse can be done easy and fast and in exactly the same way as it is done for normal JUnit tests. It can be executed with *Run as JUnit Test* and the results are presented as explained above in the Eclipse JUnit view, on the console, and in files.

The tests can also be executed automatically on the build server. In order to build the source code the build tool Maven is used, which in turn uses Surefire to execute the tests. Surefire has to be configured in the `pom.xml` file in a way that the execution of the *TestConfig* class is triggered, but not the execution of each *Elktest* class, because their execution is done while the *TestConfig* class is executed.

The execution of JUnit tests is the purpose of Surefire and the `Description` class is used together with the failure messages to log and present failures that occur while the tests are executed. In this description the layout algorithm, test class, test method, test graph, and configurator are present in order to define the test case. The failure message that is used is the one declared in the assert statement that failed. The description class and the failure message implemented in JUnit are used by this framework in order to pass the information about succeeded and failed tests to the build tool, because they can be understand are are handled by these tools.

There is one *TestConfig* class present on the build server that should be used to execute all *ElkTest* classes. This *TestConfig* class specifies that the results should not be stored in a file and the graphs should not be stored either. This class names the `test` directory as source directory for the test classes. Therefore the tests that should be executed on the build server should be located in the `test` directory or one of its sub-directories and have to follow the naming convention introduced in Section 5.1.6.

If there is a test located in another directory, another *TestConfig* class can be specified with the tests in this directory. Alternatively the tests can be specified with the `TestClasses` annotation in the original *TestConfig* class or the tests are named in another way and are not executed with a *TestConfig* class. If the tests should be executed without a *TestConfig* class their name must not end with `ElkTest`, but with `Test`. The reason for that is the configuration of Maven in a way that it ignore all the test starting or ending with `ElkTest`. This is done because the test runner should not be invoked with each of the tests separately, but only on the *TestConfig* class, to enable the grouping.

5.2 The Test Results

The results of a layout run can be displayed and stored in different ways. If the tests are executed in the Eclipse IDE, the built-in view provided by Eclipse can be used. What is displayed in the view will be explained in Section 6.7. In addition to the Eclipse view there are information about failed and

succeeded tests printed on the console. The output of the results is displayed in Figure 6.5.

In addition it is possible to configure the test run in a way that the results and graphs that lead to failed tests are stored in files. The default configuration is that no results are stored in files. The example Algorithm 5.9 is configured in a way that the results are stored in a file and that those graphs are stored that causes test methods to fail.

Chapter 6

The Test Framework

After describing the creation of test cases from the perspective of a test engineer in the previous chapter, this chapter continues with the framework designer's perspective. The framework aims to make the design of tests as described in Chapter 5 as comfortable and easy as possible. The design decisions reached with respect to these aims are explained in this chapter. In addition, some design decisions are compared to other possible solutions and the advantages of the chosen implementations are mentioned.

6.1 Structure

The test framework is based on the JUnit framework. That leads to a comfortable execution in the Eclipse IDE and on the build server. In JUnit a *runner* is used to execute the test methods implemented in the test classes. The runner that should be used is specified by the class annotation `@RunWith`. An instance of this runner is constructed with the test class as input. A description of the runner is requested that contains information about the runner and its children. After that the `run(...)` method of the runner is invoked in order to execute the tests. A notifier is handed over to this method that is used together with the description to collect and present the results.

The test framework is designed in a way that features are implemented in test runners. Another possibility would be to use the test runners provided by JUnit with as few changes as possible and to implement the features in abstract test classes. In order to do that it would be possible to use the `@Before`, `@After`, `@BeforeClass`, and `@AfterClass` annotations. The layout algorithm could be executed in methods called before the test methods and the results can be printed or stored in methods executed afterwards. Actual test classes would have to extend these abstract classes to use the features they provide. It would be possible to provide different abstract test classes for different layout algorithms or types of tests in order to provide different features. The possibilities of this approach however are limited, especially if information from other test classes should be used, and the grouping of test cases and results, as it is done in the current implementation, is not possible. Therefore the framework uses the approach to implement customized test runners. Nevertheless it is possible to create abstract test classes in order to provide additional features. One possible use case is the creation of an `AbstractLayeredTest` that specifies the locations of the graphs that should be used by default to test the layered algorithm. Other advantages of the chosen approach are smaller and neater test classes that can be designed without the restrictions imposed by having to extend certain framework classes.

The features provided by the framework are accessible through annotations. Using annotations is more comfortable than the usage of naming conventions, because the annotations can be omitted or repeated and names can be used that are meaningful to the current test.

The *root test runner* that is used to execute the test classes by the framework is the `LayoutTestRunner`. Accordingly a test class that should be executed with the framework has to be annotated with `@RunWith(LayoutTestRunner.class)` and has to be executed as JUnit test. In order to execute a JUnit test a `run(...)` method of the class `JUnitCore` has to be invoked. This can be done with the help of

an IDE or with the command line. `JUnitCore` will construct a new `LayoutTestRunner` and calls its `run()` method.

The framework can be used to execute one or several test classes at once. If only one test class should be executed, the `@RunWith` annotation has to be attached to this test class and the test class can be executed like a normal JUnit test. If more than one class should be executed there is a principle used that some JUnit users will know from the JUnit `Suite` runner: A special class has to be created in order to specify the tests that should be executed. This class will be called *TestConfig* in the rest of this work. It has to be annotated with `@RunWith(LayoutTestRunner)` and in order to execute the tests this *TestConfig* class has to be executed as a JUnit test. Consequently the constructor of the `LayoutTestRunner` will be invoked with the *TestConfig* class.

There are two possibilities provided to specify test classes in the *TestConfig* class:

1. List the test classes with the `@TestClasses` annotation.

2. Specify the location of the test classes with the `@TestPath` annotation.

The usage of the annotations was explained in Section 5.1.6.

The first possibility is consistent with the implementation of the JUnit `Suite` runner. Therefore it is a principle some JUnit users are used to and it provides a fast and easy method to specify exactly the tests that should be executed. The drawback of this solution is that every time a new test is implemented it has to be specified explicitly in every *TestConfig* class that its execution should be added to. Therefore the *TestConfig* class used for the execution of all tests on the build server has to be changed for every new test. This would make the automatic execution too failure prone.

The second possibility is more comfortable and fail safe especially for the execution of the tests on the build server. Besides these advantages there are two small drawbacks: The tests that should be executed on the build server have to be stored at the same location and have to follow a naming convention. That the tests have to be stored at the same location is not a big drawback, because that follows the common practice of the ELK project. Naming conventions are commonly used in the context of automatic execution of tests and a good practice. The naming convention used by this framework is that the names of the test classes have to end or start with `ElkTest`. If this is the case and the tests is in the directory specified in the *TestConfig* class, the tests will be executed automatically by the test framework on the build server.

The configuration of the build tool Maven can be done in accordance with the naming convention. If this is done, Maven ignores the test classes that start or end with `ElkTest`, because the test framework should not be invoked on every test class separately, but only on the *TestConfig* class. Therefore each test that should be executed by this framework on the build server has to be executed by a TestConfig class. The invocation of the test framework on all tests together has advantages described later. Generally the tests implemented in the ELK project are located somewhere in the directory `test`. Therefore it is sensible to create a default `TestConfig` class for the execution of tests on the build server that has this directory specified as location of the test classes. Therefore each test stored and named in the usual way is executed without any problems automatically each time the project is built. Both possibilities to specify test classes are illustrated in Figure 6.1.

As soon as all the test classes are loaded they are divided into *black box tests* and *white box tests*. Black box and white box tests are used as explained in Section 4.2.1 and Section 4.2.2. In order to execute the tests the `LayoutTestRunner` uses the classes `WhiteBoxRunner` and `BlackBoxRunner`. One instance of a `WhiteBoxRunner` is created for each algorithm that is specified by a white box test class and accordingly a `BlackBoxrunner` is created for the black box tests. These runners use the `GraphProvider` to load the test cases. After that they execute the layout algorithm with the input, and trigger the execution of

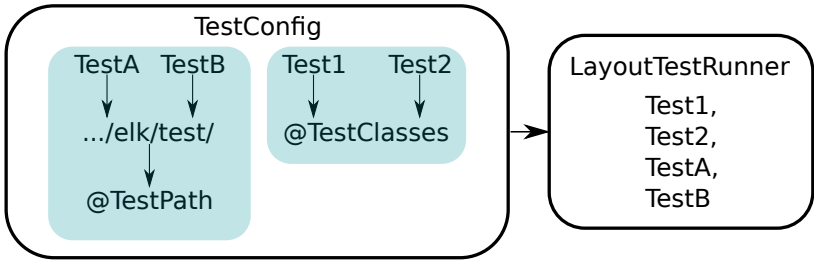


Figure 6.1. An example for the specification of test classes

the tests by another runner. The runner that is used to execute the tests is a `ActualTestRunner` that executes the test methods with the graphs.

Figure 6.2 shows an example for the execution of tests. The black box tests one and two are executed by the same `BlackBoxRunner`, because they test the same layout algorithm. The runner executes the layout algorithm with each graph separately and then triggers the execution of the test methods by an `ActualTestRunner`. The actual test runner gets the graph after layout and executes every test method of the test class with that graph.

The `run()` methods of the test runners are called with a `RunNotifier` that has to be notified about all important events, such as the execution of tests or failures. This `RunNotifier` is used by the `LayoutTestRunner` to collect the results of the test run in order to present them after all the tests have finished execution. More about the results is explained in Section 6.7.

6.2 Test Input

As soon as a `WhiteBoxRunner` or `BlackBoxRunner` is created this runner prepares the execution of the tests. On that account first of all the test cases are derived, which describe a graph, its configuration, and the test methods to be run. For each test case the input graph and configurator has to be loaded and combined. In addition to the input there are the test methods as part of the test case. If the `run()` method of the runner is triggered it iterates over all test cases. For each test case its graph is laid out and its test methods are executed.

Figure 6.3a shows how the `BlackBoxRunner` prepares the test cases. First the runner uses the `GraphProvider` to derive the test cases from the test classes. The `BlackBoxRunner` calls the method `loadConfigGraphs(List<TestClass> testClasses)`, which returns a list of all test cases. Each of these test cases contains a graph and optionally either a configurator or a configuration method that is used by the runner to configure the graph. If there is a layout algorithm specified, this is set as an option on the graph and the configurator or configuration method is used to set further options. After that is done the nodes, ports, and edges are configured with default values, if not specified differently, and the labels with size zero and a text have a more sensible size computed for them. With these configurations done the test cases are ready for the layout and the execution of the test methods. The preparation of the test input is implemented in an abstract runner class that is extended by the `BlackBoxRunner` and `WhiteBoxRunner`.

Figure 6.3b shows the workflow in the `GraphProvider`. The `GraphProvider` does the import and generation of graphs, the combination of graphs and configurators, and the grouping of test cases. The grouping is described in Section 6.8. First the provided specifications of graphs, generator options, paths, configurators, and configuration methods are loaded. The specified `.elkr` files are considered paths as well. The paths can denote directories or files. After that the graphs and configurators are

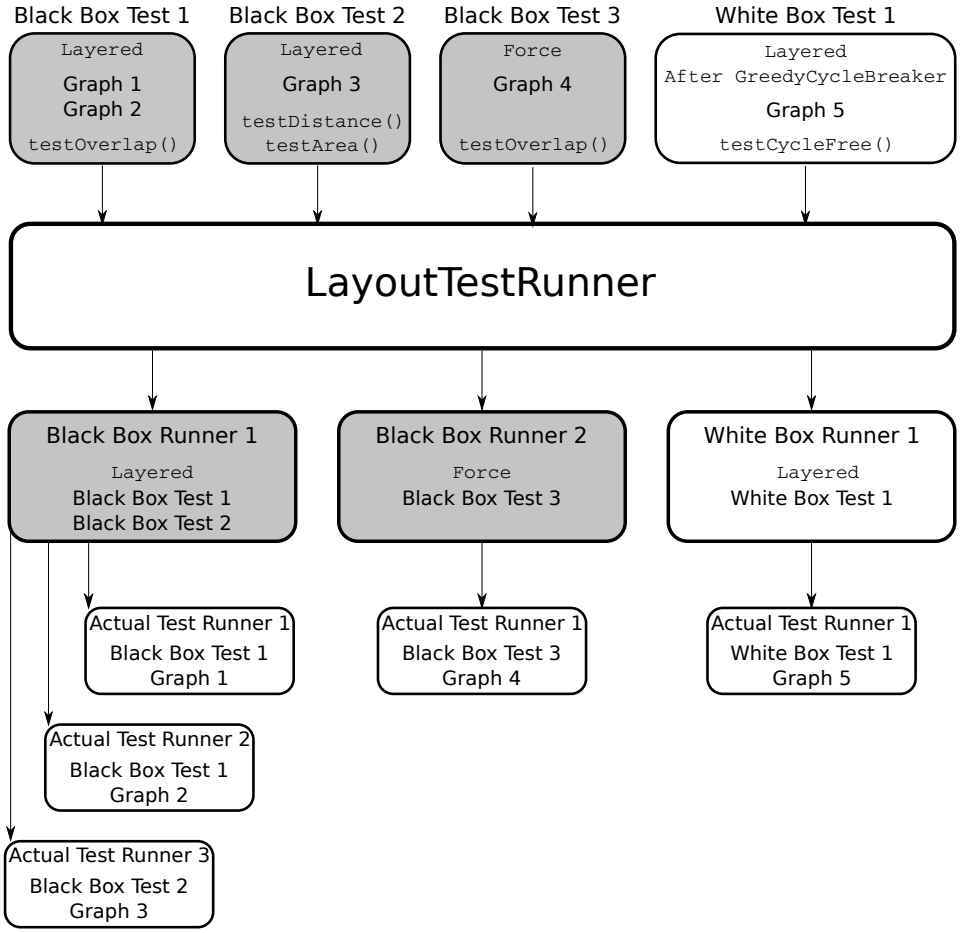
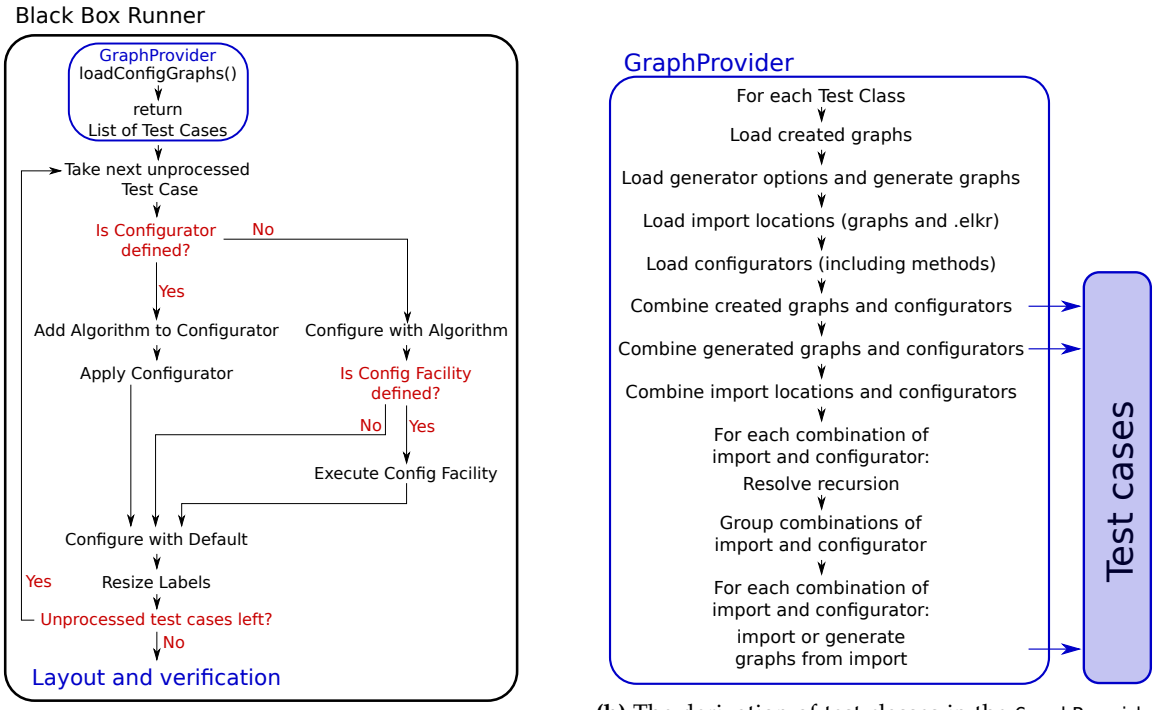


Figure 6.2. An example for the structure of the test runners

combined into test cases. The test cases with graphs that are directly returned by methods or generated can be added to the list of test cases. For the ones with the paths first of all the paths that lead to a directory are resolved and for each of the files that should be loaded from the directory a test case is created. These test cases are grouped and after that the graphs are imported or generated. In order to group the tests first of all the paths are compared. At this point all the paths point to files and therefore that maximal possible amount of equal paths can be found. If there are several test cases with the same path found, those that have the same configurator are grouped as well. If two test cases have no configurator or `LayoutConfigurators` with the same properties they can be grouped. Test cases with a configuration method cannot be grouped. After the grouping all the test cases are complete and can be returned to the runner.

6.2.1 Graphs

The graphs are the first part of the input of each test case. The three different strategies to specify graphs that are implemented in the framework are handled by the `GraphProvider`. Each of the graphs has a name added to it that is used for the presentation of the results. This name should be unique for each test class, because otherwise the descriptions of the test cases have no unique identifier and the



(a) The preparation of the test input for the execution of black box tests

(b) The derivation of test classes in the `GraphProvider`

Figure 6.3. The creation and configuration of test cases

results can not correctly be displayed in the Eclipse IDE. The description contains the names of the test class, test method, graph and configurator. Therefore methods with the same name in different classes are no problem.

Create a Graph

First of all the `GraphProvider` creates a list of all graphs and adds the graphs created in methods. The creation of a graph in a method is supported by the annotation `@Graph`. For each test class the `GraphProvider` looks up all methods annotated accordingly. The `GraphProvider` tries to invoke each of these methods without parameters. If the method returns an `ElkNode` this is added to the list of graphs. The name of a graph created in a method is the method's name.

Random Graph Generator

A version of a *random graph generator* was already available in the old code of the KIELER project. Before this was available for the framework, it had to be migrated to ELK. In the context of this work the generator was ported to the new graph format and transferred to the ELK project. The random graph generator is invoked with an instance of `GeneratorOptions` that holds the desired properties of the graph to be generated. These options can be provided directly by a method in the test class. The other possibility is to derive the `GeneratorOptions` from a file. In this file the options can be specified in a very comfortable way and the file can be used by several different test classes.

For the random graph generator these two possible opportunities have to be distinguished. The graphs generated out of directly specified generator options are added to the list of graphs that is used for the graphs created by methods. The files with generator options are added to the list of import paths.

First the methods annotated with `@RandomGeneratorOptions` are looked up and executed. If this was successful, the options are used in order to generate the graphs and the graphs are added together with the name of the method as identifier to the list of graphs.

The methods annotated with `@RandomGraphFile` are looked up and executed as well. The returned `ResourcePaths` are added to the list of paths.

If there are several random graphs created from the same `GeneratorOption` object, the name of the graph that is based on the method or file name is numbered in order to provide a unique name.

Import a Graph

The feature to import of graphs can be used by the annotation `@ImportGraphs`. A method annotated accordingly has to return a list of `ResourcePaths`. There are different classes provided by the framework that extend `ResourcePath` and can be used to specify a location of a graph. One of these classes is the `ElkRepositoryResourcePath` that has the purpose to specify locations in the ELK repository. The `ModelsRepositoryResourcePath` can be used to specify locations in a *models repository*. This kind of repository can be used to store files representing graphs and models from other projects, such as SCCharts. The models are not part of the source code of the ELK project and therefore they should not be stored in the ELK repository. There was such a models repository available in KIELER. At the time of writing such a repository is planned for the ELK project, but not yet available.

In order to use these two classes the related environment variable or system property has to be set that specifies the location of the respective repository. It is important that the tests can be executed without changes on different systems, because it must not be necessary to change them in order to execute them on the build server. Therefore the location of a file cannot be specified absolutely. To specify a file relative to the test class or the class file of a test runner makes a demand on the system as soon as there are files needed that are not in the ELK repository. This would demand that the relative location of the specified file is the same on each system. Therefore system properties or environments variables are used in order to specify the location of a directory the specified paths are relative to. The directories that can be specified in this way are for example the ELK and the models repository.

The variables of the abstract class `ResourcePath` are:

`File file` The file specified with the path. This can be a directory.

`String filePath` The path to the file. This is in general relative to a path specified in an environment variable or system property. With which path it is combined depends on the used class extending `ResourcePath`.

`boolean isDirectory` If the `filepath` ends with `/**` or `/**/` the file should be a directory and this boolean value denotes that potentially several files should be imported out of a directory.

`boolean recursive` If the `filePath` end with `/**/` this variable is set to true and the files are looked up in the directory and its sub-directories.

`FileFilter filter` The `ResourcePath` allows the specification of a filter instead of a file extension, because that provides more possibilities. To filter the files dependent on their extension is certainly the most frequently used case and therefore there is a `FileExtensionFilter` implemented that can be used. Another use case for a filter is the exclusion of special sub-directories.

The classes that extend `ResourcePath` have to implement a method that returns the absolute path of the file and an `equals(...)` and `hashCode(...)` method. The `equals(...)` method is needed in order to group test cases with the same input.

6.2.2 Configurators

The configurators are applied as shown in Figure 6.3a. The test cases contain a graph each. If a graph is used for several test cases, because it has to be combined with different configurators, the graph is cloned and there is a different instance contained in each test case. This has to be done in order to make sure that there are no options set on the graph that had been set in the layout run triggered by the previous test case. It is not possible to delete all old options, because usually options that are important are contained in the original graphs. The same is valid for the layout information attached to the graph.

If the test case also contains a configurator this will be used or alternatively a configuration method. If an algorithm is specified for the test runner, it is set with a configurator on the graph. If there is a configurator in the test case the algorithm will be added to this one. If there is a configuration method a configurator with the algorithm will be created and applied before the configuration method is executed. This provides the possibility to set a different layout algorithm for different parts of the graph in the configuration method. It is recommended to set an algorithm in a configurator or configuration method only if there is no algorithm specified for the test class. Otherwise this can make the test harder to comprehend.

After the graph is configured the default values are set and the graph is laid out. During the layout the options that are set come into effect. The execution of the layout algorithm is influenced by the options set on the graph. The option that makes that most obvious is the one that sets the layout algorithm. There are many other options provided by the layout algorithms. The `ElkLayered` algorithms for example provides the usage of the different implementations of the phases as options. Therefore this option set on the graph can cause or prevent the execution of a white box test method.

For the creation of test cases each graph is combined with each configurator. Why this is done is explained now. One example for an option that can be set with a configurator is the `NODE_TO_NODE_SPACING`. Obviously a negative node spacing is not sensible. During the creation of test cases it is important to test for such values that do not fit in the specification and have to be handled especially by the algorithm. Such values that are out of range have to be tested separately. Therefore each configurator that sets values that are out of range has to be combined with at least one graph without other problematic values. Such a graph with problematic properties can be a graph that contains a cycle as input for an algorithm that can only handle acyclic graphs. Consequently the combination of just one graph with one configurator can lead to fewer test cases and the combination of each graph with each configurator can lead to more test cases than necessary, but it is the safest and easiest possibility and therefore the selected one. That still does not lead to the combination of each characteristic with each other.

6.3 Black Box Tests

As already explained in Section 6.1 a `BlackBoxRunner` is created for each algorithm that is specified in a black box test. Therefore the black box runner knows the algorithm and the test classes that should be executed in order to test the graphs after layout.

First of all the black box runner needs to have a complete list of test cases. In order to do that the `GraphProvider` is used that returns a list of test cases derived out of the specifications of graphs and

configurators made in the test classes, as explained in Section 6.2. A test case consists of the following information:

The Graph The graph that has to be used as input for the layout algorithm. This graph is configured by a test runner, laid out by the layout algorithm, and used as test input afterwards.

The Path for Import For a graph that is imported or generated from an `.elkr` file this is the location of the file.

The Name of the Graph The name of the graph is needed to present and store the results.

The Random Graph Flag This indicates whether the graph is a randomly generated graph.

The Default Configuration Flags These flags indicate whether the nodes, ports, or edges should be configured with default values.

The Configurator The configurator that has to be used to configure the graph. This can be `null`.

The Configuration Facility The method that has to be used to configure the graph. If the configurator is not `null` this one should be `null` or is ignored otherwise. It is possible that there is no configurator and no configuration facility specified.

The Test Classes The test methods of these test classes are used to verify the graph after layout.

As soon as the test cases are available the graph is configured. The layout algorithm is set as option on the graph as well. With this configured graph an instance of the `RecursiveGraphLayoutEngine` is created and used to compute the layout for the graph. This takes care that all hierarchy levels of the graph are laid out in the way specified in its options. After that is done the tests are executed. In order to do that the `run()` method of the `ActualTestRunner` for the test class is invoked.

6.4 White Box Tests

For the execution of white box tests with the same algorithm a `WhiteBoxRunner` is used.

Like the `BlackBoxRunner`, the `WhiteBoxRunner` first loads the test cases and then uses an instance of the `RecursiveGraphLayoutEngine` in order to layout the graph.

The test cases that represent a white box test are more complex than the ones that represent a black box test. The test cases for the white box tests contain the same information, but in addition to the test classes there are *test breakpoints* defined. These breakpoints define before or after which processor which test methods have to be executed.

For each of these test cases the `WhiteBoxRunner` constructs a `TestController` that is handed over to the `RecursiveGraphLayoutEngine` in order to execute the tests.

The `TestController` holds four `Multimaps` of listeners that are notified when the processor they are waiting for is about to execute or has finished execution. These maps are:

`procPreListeners` The map with processors and the related listeners that should be notified each time before the processor is executed.

`procPostListeners` The map with processors and the related listeners that should be notified each time after the processor is executed.

`procRootPreListeners` The map with processors and the related listeners that should be notified each time before the processor is executed on the highest hierarchy level.

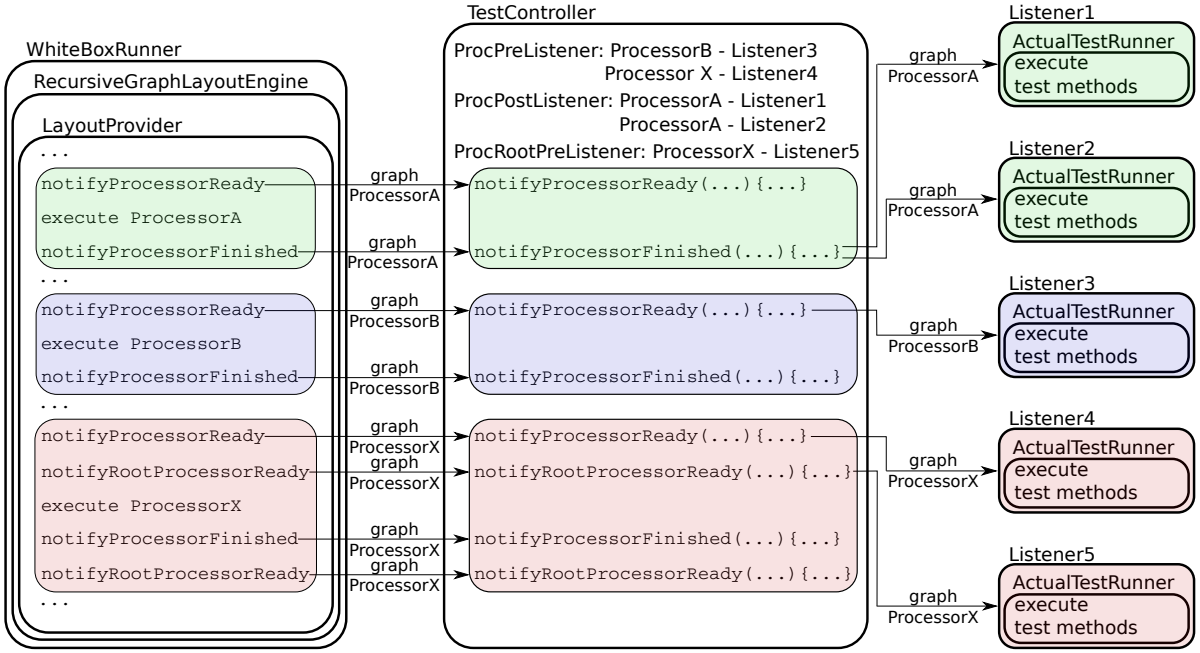


Figure 6.4. An example for the execution of a white box test

procRootPostListeners The map with processors and the related listeners that should be notified each time after the processor is executed on the highest hierarchy level.

The tested layout algorithm has to notify the TestController each time before and after a processor is executed. If the layout algorithm supports the execution of tests on the root graph it notifies the `TestController` in addition if the processors are executed on the root.

The `WhiteBoxRunner` hands this `TestController` together with the tested layout algorithm's ID over to the `RecursiveGraphlayoutEngine` and sets its boolean value `test` to `true`. This causes the `RecursiveGraphLayoutEngine` to execute the layout as white box test. This means that the `TestController` is handed over to each executed `AbstractLayoutProvider` that corresponds to the algorithm ID. The `AbstractLayoutProvider` has to ensure that it notifies the `TestController` before and after each execution of a processor. How this is done is illustrated in Figure 6.4.

For instance the map `procPreListeners` holds the listeners that should be notified before a processor is executed. Therefore each time the `TestController` is notified that a processor is about to be executed it looks in this map for the listeners waiting for this processor. If there are any, they are notified and the graph handed over to the `TestController` is in turn handed over to the listener together with the processor. The listener takes care of the execution of the test methods that should be executed at this point.

The `TestController` and the `WhiteBoxRunner` are designed in a way that it is possible to integrate the possibility for white box tests into every layout algorithm based on the execution of processors, as explained in Section 2.3.2. Until now white box tests are only supported by `ElkLayered`, but will hopefully be integrated in other algorithms as well.

During the layout of a graph the layout algorithm does normally not use all the different processors. Therefore it is possible that a white box test method is never executed, because it would only be triggered by processors that are never executed. The number of executions is not counted for one

test case, but for all test cases together. There is an output on the console that shows how often the methods are executed on how many graphs. By default there will appear a failure if a test method is never executed, because the feature that should be verified by this method has not been examined by the method. If this failure should not occur the test method can be annotated with the annotation `@FailIfNotExecuted(false)`.

6.5 Analysis Test

The black and white box tests verify the correctness of the algorithms respecting its specification. In addition to that the quality of the results is important and therefore the framework is extended by *Analysis tests*. The used analyses are part of GrAna that has been developed in the context of the KIELER project. The migration of this project is not yet completed and therefore the analysis tests are integrated as proof of concept.

Especially the execution of the tests on the build server has to lead to the answers *success* or *failure* and therefore the analysis results have to be interpreted in that way. This can be done by the definition of limits. Therefore the layout algorithm will be categorized as faulty if the quality of the results crosses a specified limit.

In order to integrate the analysis results in the framework there were some design decisions to be made:

What should be verified by the analysis tests? Theoretically it would be possible to design an analysis test as a white box test or a black box test. Therefore the output of a processor or the graph after the complete layout could be verified. In the framework the analysis tests are integrated as black box tests, because it seems much more sensible to analyse the results of the whole layout and not just of a single phase.

How should the tests be integrated in the framework? It would be possible to analyze every graph that is laid out for the verification tests as part of the framework. If this is done the random graphs would have to be omitted and the failure of an analysis would have to be handled separately. Another possibility is to create an analysis test as a black box test with some additions. This is the implemented solution. This allows to create an analysis test with the highest amount of control. The analysis is just executed on the specified input and the evaluation of the analysis can be done exactly the way it is wanted.

When should an analysis test fail? It would be possible to average the analysis results of the specified test cases and to verify whether or not they are in range. With this strategy it can stay unnoticed if a value exploded in a small number of graphs. Therefore the results of each analysis test case are compared to comparison results. The valid range for a new result can be specified as a percentage or as an absolute value. There already is an evaluation of analysis results implemented that uses absolute values, but it is no problem to implement an own evaluation that uses percentual ones.

How should the comparison values be stored? In some way the tests need to access values for comparison. If these are stored in the ELK repository that would lead to many conflicts, because the results are updated on the local systems. If a special repository is used this problem would not be observable that often, because the ELK repository is used more regularly, but the problem would still be present. A nice alternative would be the usage of a database, but this is much effort and therefore not implemented. The chosen solution is the storage of the old results in a file located in the local file system.

Because an analysis test is a black box test it is executed by the `BlackBoxRunner` and after the layout is completed the analysis test is triggered in order to run the analysis and verify its results, as explained in Section 5.1.5. In addition to this usual layout and test execution the `BlackBoxRunner` has to

ensure that the analysis test is not executed with a random graph and that the results of the analysis are stored only in the correct cases.

If the input of the analysis test is specified as random graph there will occur a failure, because the results gained on generated graphs are not reproducible and can therefore not be compared to the old comparison results. In order to provoke this failure there is a special test executed that always fails. This test has an appropriate failure message that tells the developer that analysis tests are not allowed to have a randomly generated input. There is such a special test used instead of an error message printed on the console, because such an invalid test should not be possible in a successful test run and an error message can be missed too easily.

The storage of the new results in the file should be done, if explicitly requested in the analysis test configuration and if there is no failure. The only failure that should not prevent the runner from storing the files are failures because the file with old results does not exist or the combination of input and algorithm can not be found in the file. The new results are stored after all the analysis tests in the `BlackBoxRunner` are executed and the failures of all analysis tests are taken into account. Therefore the results are stored in a temporary file until all the tests had been executed. After all the tests are executed, the `BlackBoxRunner` replaces the old results file by the temporary results file or deletes the temporary file, dependent on the failures that occurred.

6.6 Running Test Methods

To execute the test methods the `BlackBoxRunner` and the `WhiteBoxRunner` use the `ActualTestRunner`. This runner adds the features required to execute a test method in the context of the framework to the `BlockJUnit4ClassRunner` provided by JUnit. The annotations provided by JUnit, such as the `@Before` and `@After` annotation, are handled by the `BlockJUnit4ClassRunner`. In JUnit it is not supported that a test method has parameters. Therefore the `ActualTestRunner` checks that the parameter types are the expected ones and invokes the test methods with the parameters. This is for a black box test the graph as `ElkNode`, for a white box test a graph of the type used in the algorithm and optionally the processor, and for an analysis test the graph together with the specifications needed to execute the analysis test and information about the test case. Another feature provided by the `ActualTestRunner` is the storage of the graph a test failed on. At last the `ActualTestRunner` can optionally execute a method annotated with `@After`.

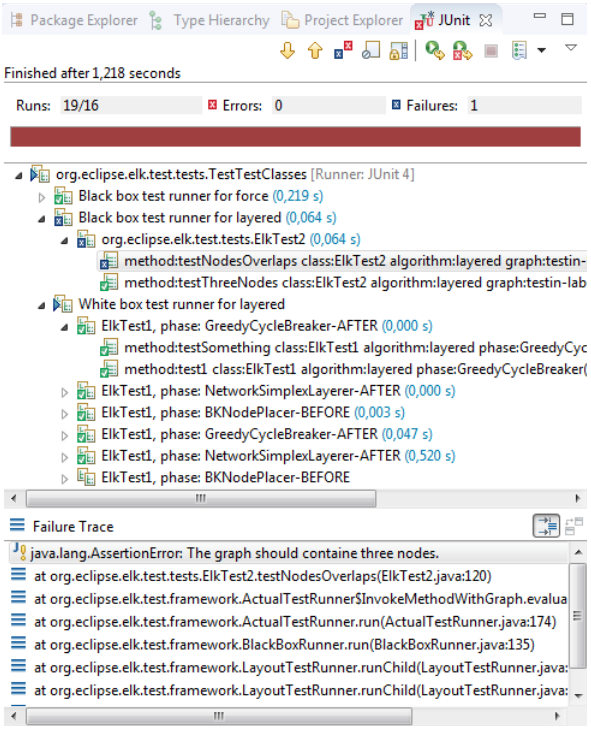
The procedure for each test method is the following:

1. Execute `@Before` methods, if present
2. Execute the test method
3. Optionally store the failed graph
4. Execute `@After` methods, if present

6.7 Test Results

The overall result of a test run is either *success* or *failure*. If all the tests succeeded there is normally no more information needed, but as soon as at least one test failed the first information needed is how many tests failed and which tests. This information is printed on the console and displayed in the *Eclipse JUnit view*. The view in Eclipse is certainly first used by most developers in order to gather an overview of what failed. There is the typical green or red bar included that turns red in the

presence of a failure. Beneath of that is a tree view of the executed tests. The root of the tree is the `LayoutTestRunner`, at the next level are the `WhiteBoxRunners` and `BlackBoxRunners`, at the level after that are the test cases and after that the test methods. One drawback of this view is that the number of executed tests does not neccessarily match the expected number of executions, even if all tests are executed in the right way. The reason for this lies in the execution of white box tests. If these tests are executed it is not possible to name the expected number of test executions because it depends on the input of the layout algorithm how often the processors are executed that lead to the execution of the white box tests.



(a) The JUnit view in Eclipse.

```
SUCCEEDED on graphs:
    graph:hierarchy01.elkg(org.eclipse.elk.test.tests.ElkTest1)

algorithm:layered
phase:BKNodePlacer(BEFORE)
method:test1
SUCCEEDED on graphs:
    graph:hierarchy01.elkg(org.eclipse.elk.test.tests.ElkTest1)

algorithm:layered
phase:BKNodePlacer(BEFORE)
method:test2
SUCCEEDED on graphs:
    graph:hierarchy01.elkg(org.eclipse.elk.test.tests.ElkTest1)

algorithm:layered
method:testNodesOverlaps
FAILED on graphs:
  graph:testin-label-resize.elkg(org.eclipse.elk.test.tests.ElkTest2)
    failure: The graph should containe three nodes.
```

(b) The output on the console.

Figure 6.5. The output of the results in Eclipse.

It is printed on the console on which graphs a test succeeded and failed together with the failure message. This failure message is also displayed in the Eclipse view.

This information makes it possible to figure out on which input which test failed. In order to find the fault it is often helpful to examine the input of the test and therefore the output of the layout algorithm. In this use case this output can be best made available by storing the graph in a file that can be opened and examined visually. For each test run the `LayoutTestRunner` checks whether the graphs should be stored and this information is handed over to the `ActualTestRunner` that actually stores the graphs after a test failed.

It is as well possible to automatically store the information which tests succeeded and failed in a text file. This feature and the feature to store the failed graphs can be easily turned on and of in the *TestConfig* class.

There are two *special failures*: If a white box test is never executed and if there is a random graph used as analysis test input. If one of these situations occurs not that test fails, but there is another

test displayed beneath of the other tests in the Eclipse view that indicates what has happened. This is done because the tests are not executed and therefore there is a special test executed that fails with the respective failure message. There is not just an error printed on the console, but a test failure enforced, because in these situations the automatic build should fail and the error should not stay unnoticed.

Technically the results are collected by the `LayoutTestRunner` by adding a listener to the test notifier. This notifier is handed over to each `run(...)` method that is called on each runner in order to execute the tests. The runners notify the notifier if they start or finish execution and if a failure occurs. These information are collected by the listener and as soon as all the tests finished execution the results are grouped, printed, and optionally stored.

In order to collect and present the results the `Description` class implemented in JUnit is used. Every test runner has a description that has a name that is displayed for example in the JUnit view of the Eclipse IDE and optionally a list of children. The descriptions returned by the test framework are:

`LayoutTestRunner` The description of the `LayoutTestRunner` has the name of the test class it is invoked to represent the runner and the `WhiteBoxRunner` and `BlackBoxRunner` as children.

`WhiteBoxRunner` Each `WhiteBoxRunner` has as representation the string "White box test runner for ..." with the name of the tested layout algorithm. The children of this description are the test cases executed by the `WhiteBoxTestRunner`.

Test Case The white box test cases have the different combinations of test classes and the execution before or after the processors. Each of these children corresponds to one entry in the lists of the `TestController` used in the execution of the test case. The children of the test case are the test methods.

`BlackBoxRunner` For a `BlackBoxRunner` the displayed description is "Black box test runner for..." and the children are the test classes. This test classes have the test methods as children. The test methods are displayed together with the test input.

A failure occurs in the test method and it can be looked up in the Eclipse view in which test method a failure occurred. Because of that the complete information about the test input is displayed together with the method. Therefore the complete information about the failed test is available together.

6.8 Grouping Test

One feature integrated in the test framework is the grouping of tests in order to run them more efficiently. As explained above a test case consists of the test input and the verification. In each test class a set of inputs is specified. For two test classes it can happen that their input sets intersect. With the input in the intersection test cases can be created together with the test methods of both test classes as verification. This can also be done for more than two test classes. If the tests are not grouped several test cases are created each with the same input and the test method of one of the test classes as verification.

In the example shown in Figure 6.6 two black box test classes test the same layout algorithm, import the graph `graphWithBigNodes.elkg`, and use a configurator that just sets the node to node spacing to 30.0. The test methods in these tests can be executed after the same layout run and the graph does not need to be imported twice, but just once. The third test case has another configurator and can consequently not be grouped with the other two.

Test cases are grouped if they import a graph from the same import file or use the same file to configure the random graph generator. Test cases with graphs created in a method of the test class

are not grouped, because the comparison of the graphs is more complex. The situation that two test classes create the same graph should not occur that often, because this method should only be used if the graph is specialized for the test class. If the graph is useful for other test class it should be stored in a file in order to make it accessible for other test classes. Random generator options are not compared, because most likely the overhead for comparison would be bigger that the gain in efficiency.

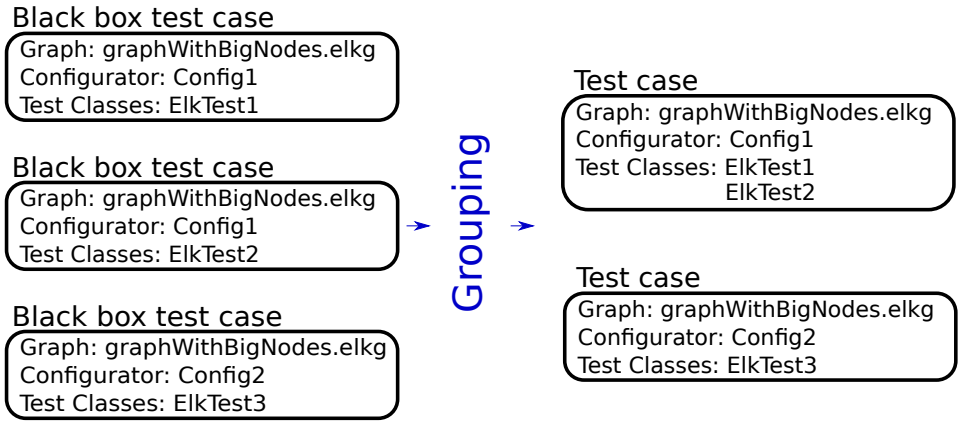


Figure 6.6. An example for the grouping of test cases

Chapter 7

Evaluation

This chapter presents a short evaluation of the test framework. First of all it is important to do research and to analyze how a test framework can be evaluated and what can be adapted to this specific case. With this background the framework is evaluated and a small case study is presented.

7.1 Evaluation Strategy

Before a software product can be evaluated it is necessary to study which criteria are important for the specific product. In some cases there are metrics available and a rich set of publications can be used in order to evaluate an algorithm or product with approved scientific methodologies. In other cases it is harder or even impossible to provide metrics and rate the quality of a solution objectively and precisely. In these cases it is important to derive the important criteria from the situation and the available sources. After the criteria are known it has to be determined how the software can be evaluated with respect to them.

In the case of this framework the *performance* and the *usability* are viewed as important. The usability has been stated an essential part of the problem statement and the performance is especially crucial because the tests are executed regularly and often.

In order to evaluate the usability of a system it has to be clear how the usability of the specific system can be defined. The usability cannot be stated in an absolute sense, therefore it is sensible to define it with respect to comparable methods, tools developed by competitors, or predecessors. It is always important to have the purpose of a system in mind. Therefore the users of the system, the tasks that are performed with it, and the environment in which it will be used are important. Often the suggestions made by the ISO 9241-11 standard are used [ISO98]. It names the components *effectiveness*, *efficiency*, and *satisfaction* as parts of usability. Effectiveness and efficiency have to be measured as explained above with respect to the evaluated project. In contrast subjective satisfaction can be evaluated in a more generalized way. That enables the creation and publication of questionnaires and attitude scales that are not developed for a specific case study, but for a wide range of systems. One such scale that can be used to rate subjective satisfaction is the System Usability Scale (SUS) [Bro96]. This scale consists of ten statements that aim to grade the subjective usability in a comparable way. For each statement the user has to rate how strongly he or she agrees or disagrees on a scale from 1 to 5. The answers can be summed up to a total score between 0 and 100 in order to rate the system with a single number. The fact that usability is defined by the context in which a system is used is neglected in order to provide such a general measure that can be used for global assessment of system usability. Consequently the SUS provides a quick and dirty way to evaluate the usability, because such low cost solutions are needed in the industry.

In order to measure the usability it is sensible to use a *case study*. A case study is used often to evaluate whether another method or tool should be used in a project. There are some guidelines available that help organize and analyze a case study in order to achieve meaningful results [KPP95]. Apart from case studies it is possible to use formal experiments or surveys to evaluate a new method

or tool. A case study is done with only one team and one project. This team and project should be typical for the evaluated method or tool and can therefore be used to give a first indication as to how it works in typical situations. In this case the project is the ELK project that is not just typical, but the exact and only project the tool will be used for. Neither an experiment nor a survey were possible in the frame of this thesis. In addition to that they are not necessary, because the adequacy of the framework has to be shown only for the ELK project. Kitchenham et al. state the following steps as the components of a case study: define the hypothesis, select the pilot project, identify the method of comparison, minimize the effect of confounding factors, plan the case study, monitor the case study against the plan, and analyze and report the results. Beside of the guidelines there are also checklists available that can help to perform a good case study [HR07].

As already explained it is important to evaluate the usability in a way that is appropriate for the evaluated system and the case study has to be adjusted to the system, the team, and the project. Vos et al. presented a framework for the evaluation of software testing techniques that can be used as an orientation for the realization of a case study for a test framework [VME+12]. Their framework can be used in order to evaluate the adequacy of a method or tool for a project and compare the obtained results more easily. If this framework is used it is easier to define case studies and it is ensured that guidelines and checklists are met. The authors name the following sections that should help to define a case study:

Objective - What to achieve? The criteria named by ISO 9241-11 are used in this framework: efficiency, effectiveness, and subjective satisfaction. The satisfaction of these criteria has to be measured in the real testing environment and should be compared to the current testing practices used in the project.

Cases or treatments - What is studied? The classification of the tested method or tool is important in order to compare it with other ones.

Subjects - Who applies the techniques/tools? In the best case the subjects are developers that will potentially use the evaluated method or tool afterwards.

Objects - What are the pilot projects? The system that is tested with the evaluated method or tool should be representative for systems that should potentially be tested by it afterwards.

Variables - Which data to collect? The following variables are important:

Independent The testing method or tool that is evaluated, the team and the project that evaluate the tool or method, the level of experience of the evaluating developers.

Dependent Effectiveness, efficiency, and satisfaction.

The authors listed the aspects that should be analyzed in order to evaluate the effectiveness, efficiency, and satisfaction. Some points named in these lists are used in this work. The effectiveness can be measured by the number of test cases, failures and faults found, and by the number of false positives and false negatives. The efficiency can be measured in terms of time needed to design test cases, set up the infrastructure, test and observe failures, and identify fault types and causes for each observed failure. The satisfaction can be measured by the usage of the SUS, other questionnaires, or the subjective opinions.

Protocol - How to execute the study? First of all the testers are trained and faults are injected, if possible. With this preparation the tester develops tests and the data necessary for the study are collected. How the collected data can be evaluated depends on different factors. If there are known faults the

fault detection rate with respect to the known set of faults can be analyzed. If there is a company baseline available this can be used a basis for an analysis. If there is an existing test suite available this can be used as comparison.

Threats to Validity of the Studies Performed The threads to validity should be identified and studied with care in order to gain reliable results.

How the proposed guidelines, checklists, and the framework are used in this project is explained in Section 7.2. The results of the study are presented in Section 7.3

Besides the usability the performance should be measured. This is done in terms of the execution time and the results of the measurements are presented in Section 7.4.

7.2 Case Study

This section explains how the case study is planned and performed.

The methodological framework presented by Vos et al. [VME+12] is used as orientation for the case study, especially the questions used for the evaluation. The usage of this framework as orientation should guarantee that the guidelines and checklists that exist in this field are considered. The case study that is done in the frame of this work is quite small, but the opinions of two of the three main developers are captured. The parts named in the paper by Vos et al. that are viewed as important for this case study are named in the following together with their realization.

Objective - What to achieve? The case study is done in order to evaluate the appropriateness of the test framework for the ELK project. Therefore the usability of the framework is evaluated in terms of effectiveness, efficiency, and satisfactions. In order to evaluate these aspects it is considered to what extent the problem statement is met.

The hypothesis is that the usability of the framework is satisfying. This includes that the framework is more suitable than its predecessors and that the demands specified in the problem statement are met.

Subjects - Who applies the techniques/tools? The framework is applied by two of the three main developers of the project. The three will be the ones using the framework the most and therefore the developers of the project are well represented. The other group of developers that will probably use the framework are students writing their master or bachelor thesis in the context of the ELK project. There were no students participating in the study, because there were no students in the ELK project that could be asked. The reasons for that were that the students would need to have the required technical background and enough time to participate.

Objects - What are the pilot projects? As a pilot project the ELK project has been used. This is the project the framework was designed for. The advantage is that the final project is perfectly represented, but the drawback is that the setup is not a system with known or injected faults.

Variables - Which data to collect? This part of the publication by Vos et al. is most interesting as orientation. It is proposed in which forms the effectiveness, efficiency, and subjective satisfaction can be measured.

Effectiveness The effectiveness can be measured in terms of the number of test cases that are designed or generated. In addition the number of failures or faults that are found is interesting. It can also be measured how many of these test cases are invalid or repeated and how many are

marked as false negatives or false positives. In the context of this study the absolute number of test cases and faults and failures is not informative. A failure is the wrong behavior of the program that differ from the specification. A fault is an incorrect part of the design or implementation that may lead to a failure. There is no specified time limit in which the test cases had to be designed and there are no comparison values. The developers designed the tests in addition to their every day work, as will be done if the framework is normally used. Therefore it is more interesting how the developers subjectively score the effectiveness of the framework and the scores should not be given in total numbers, but in the grade of the agreement to statements.

Efficiency The efficiency can be measured as the time needed to learn the testing method, generate test cases, set up the testing infrastructure, test and observe failures, and time needed to identify fault types and causes for each failure. This time should be reviewed on a scale between "few" or "much" and not in absolute numbers.

Subjective satisfaction The subjective satisfaction is the most important part for this study. It can be measured with the SUS and in terms of the subjective opinion. Some of the questions asked here are inspired by the SUS and some are derived from the problem statement. The questions chosen for this study are named below.

Protocol - How to execute the study? In order to execute the study no faults are injected, but faults existing in the actual implementation of the ELK project are searched. The conditions under which the framework is evaluated are as close to the normal situation as possible that leads to meaningful results.

The questions used in order to evaluate the framework are the following:

Effectiveness The agreement to each statements had to be stated by as choice of one of five grades from "I Agree" to "I Disagree". The questions were:
I think the framework provides the facility to efficiently ...

... design a high number of test cases.
... observe many failures.
... observe many faults.

In addition to that the developers were asked for their subjective estimate of the effectiveness.

Efficiency The assessment of each aspect had to be stated by as choice of one of five grades from "little" to "much".

- Time needed to understand the framework.
- Time needed to design test cases.
- Time needed to get started writing tests.
- Time needed to test and observe failures.
- Time needed to identify fault types and causes for each failure.

In addition to that the developers were asked for their subjective estimate of the efficiency.

Satisfaction The agreements to each statement had to be stated by as choice of one of five grades from "I Agree" to "I Disagree". There are statements derived from the SUS and statements derived from the problem statement:

- ▻ I thought the system was easy to use.
- ▻ I would imagine that most people would learn to use this system quickly.
- ▻ I felt confident using the system.
- ▻ I needed to learn a lot of things before I could get going with this system.
- ▻ It is easy to write tests with this framework.
- ▻ The framework is comfortable to use for the execution of tests in Eclipse.
- ▻ The test input can be specified easily and fast.
- ▻ The feature of black box tests is easy to use.
- ▻ The feature of white box tests is easy to use.
- ▻ The framework provides the needed features.
- ▻ The results contain all necessary information.
- ▻ The results are presented in a comprehensible way.

In addition the developers were asked for their subjective opinion about the framework.

7.3 Results

7.3.1 Results of the Case Study

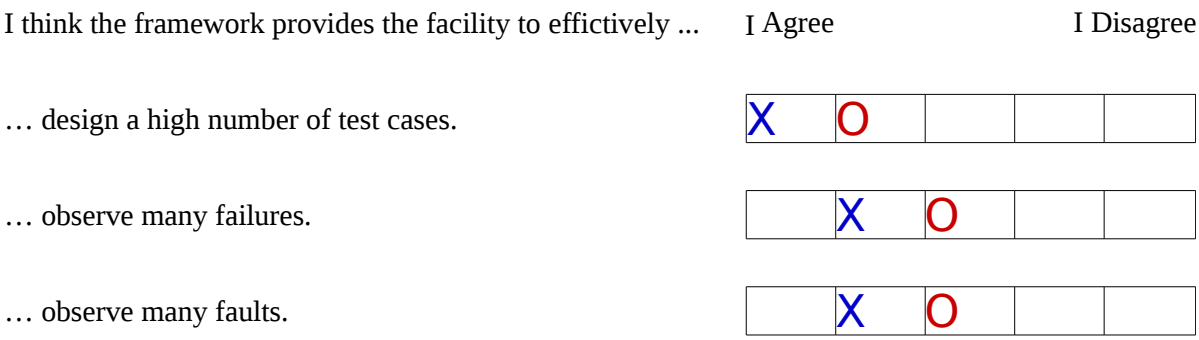
Two of the three main ELK developers already used the test framework in order to develop some tests. Their feedback was positive overall. They remarked as positive that there is very little unnecessary overhead needed to design the test classes. Furthermore the design of a black box test was perceived as easy and the tests can be designed freely. One advantage of the usage of annotations was the possibility to name the methods in the desired way. The opportunity to write tests with as few lines as possible was named as important. This had been considered during the implementation of many features, but can still be improved in some cases. One big improvement was suggested: It should be possible to execute a white box test after a phase independent of the phase's actual implementation. This is explained more detailed in Section 8.1. The other feature of the framework that was perceived as uncomfortable was the presentation of the results in the Eclipse view. The names of the test cases are quite long, because there is a lot of information required. How these information is presented and structured should be reconsidered.

The results of the case study are presented in Figure 7.1 and Figure 7.2. For the answers of each developer a different symbol is used. If there is a symbol missing this developer has not answered this question and if there are two same symbols the developer has marked two fields. These answers are based on the creation of few tests and are consequently only a rough estimate, but they are useful to deduce a hint on the strengths and weaknesses of the designed framework.

Effectiveness The effectiveness was overall evaluated as good. The facilities to observe many faults and failures were evaluated as good but not very good. The facility to generate a high number of test cases were evaluated even better. This seems to be the strength of the framework in terms of effectiveness.

Efficiency The aspects named in terms of efficiency were evaluated as good and very good. The assessment of the time needed to create test cases varies the most and the time needed to get started was rated best together with the time needed to test and observe failures. It was remarked

Effectiveness



Efficiency

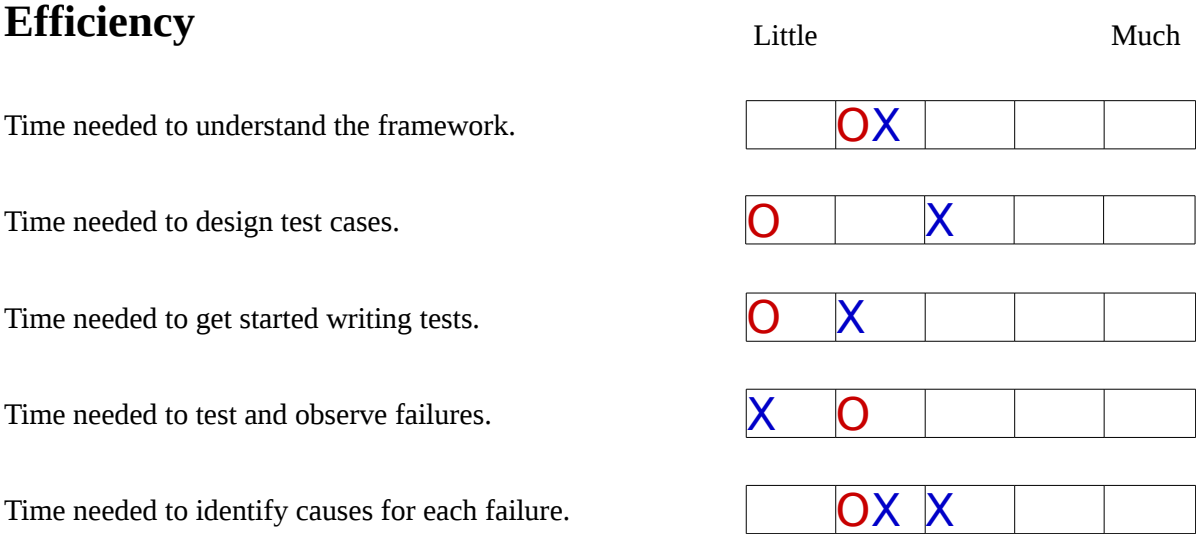


Figure 7.1. The results of the case study, part 1

that the time needed to test and observe failures and to identify the causes for failures depends on the implementation of the tests and the underlying JUnit framework. There are two crosses for the last question because of that reason, as the developer remarked.

Satisfaction Many of the points mentioned in the satisfaction part have been evaluated as very good and some as good. The white box tests had been evaluated by only one developer, because the other did not use white box tests. The three points that had been evaluated as moderate, but not as good as the other ones are:

I need to learn a lot of things before I could get going with this system. It had been remarked that special knowledge is needed to write white box tests and that this is to be expected. For black box tests some knowledge about the ELK project and JUnit is needed as well, leading to more work before the tests can be designed. However this cannot be avoided.

The test input can be specified easily and fast. That the test input can be specified easily and quick

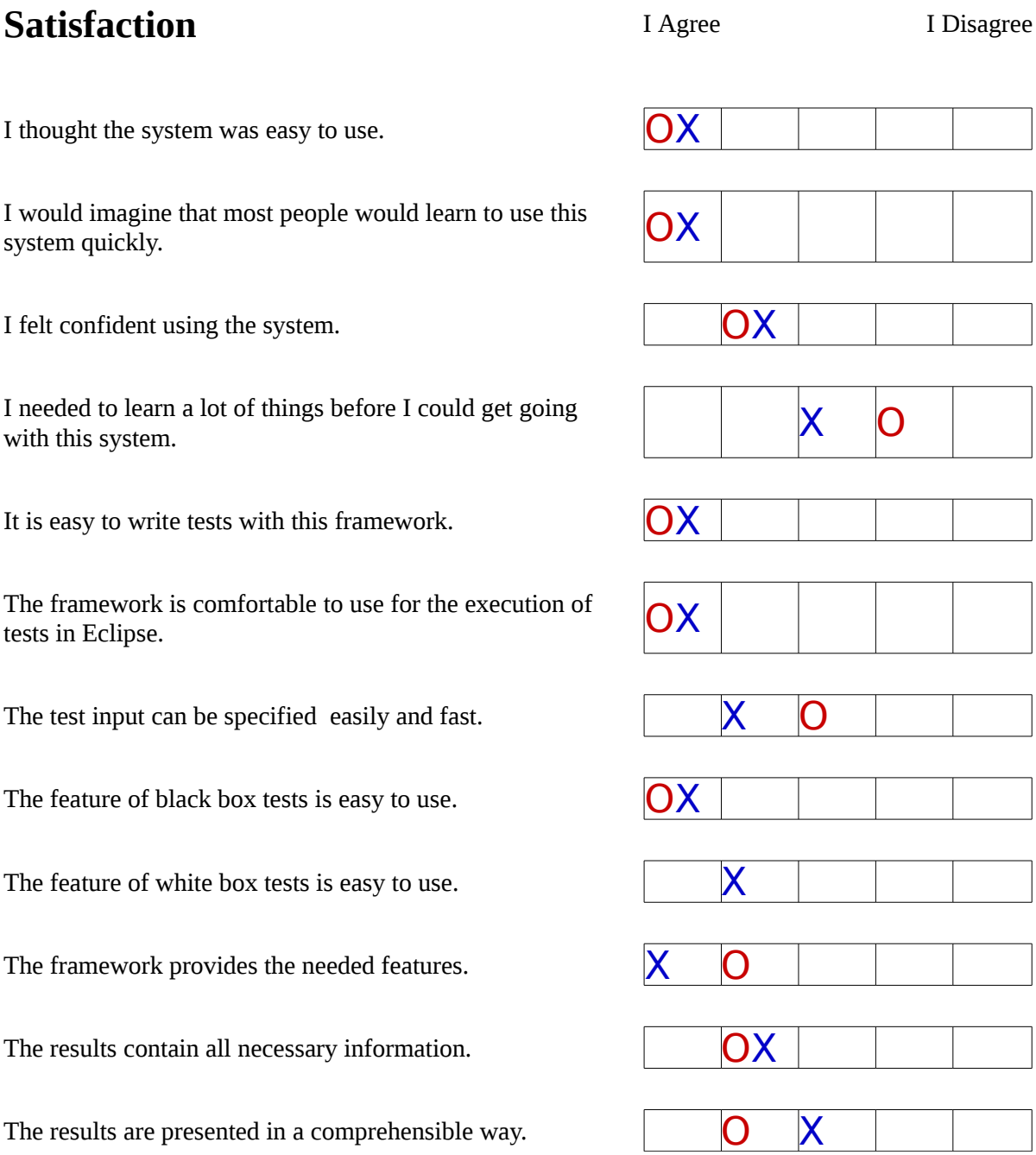


Figure 7.2. The results of the case study, part 2

was one of the main design goals. One remark was that it depends on the input how easily and quickly it can be specified. A feature that was missing was the possibility to specify a graph with a path relative to the test. It should easily be possible to add this feature.

The results are presented in a comprehensible way. The presentation of the results is not that easy to design, because there is so much information needed to identify the test cases. This leads to a long description in the Eclipse JUnit view. The output on the build server and on the console is divided into several lines and hence more readable. In addition there are often many test cases and that makes the output confusing and not very readable. The presentation of the results has changed a bit due to of this feedback, but there is still room for improvement.

In addition to these questions there were three free text answers as listed below.

How do you personally estimate the effectiveness of the framework?
The answers given here are:

- Does exactly what it should do: let me test layout algorithms rather easily.
- I think the framework is well suited for most test workflows. For better automatic test generation some kind of generator patterns would be nice.

The implementation of generator patterns can be kept in mind for future work. It would be possible to implement a wizard that provides this feature.

How do you personally estimate the efficiency?
The answers given here are:

- Getting started was rather easy and would have been even easier with yet more code samples. Had to write little boilerplate code.
- The framework allows for fast test generation. Evaluating the results is more dependent on the underlying JUnit.

In order to get started there is documentation that explains how to use the framework that is attached in Appendix A. The code examples should be added to this documentation. Therefore the efficiency seems to be good and not many changes will be necessary, before the framework can be used.

My personal opinion about the framework:
The answers given here are:

- I liked the way I can quickly specify tests using annotations. I also like the flexibility of being able to use different annotations and configurations depending on what exactly I want to test. All in all, it seems to me to be a massive improvement over any framework we had before.
- Though my experience with the framework is limited, I feel confident in specifying the tests in the framework. All the required features are presented in a useful and accessible way.

The framework seems to fulfill the main goals and is evaluated as good and suitable by the developers.

7.3.2 Example Tests

In addition to to the test classes there is the `TestConfig` class that can be used to execute several test classes in one test run. The experience acquired until now has shown that this concept is not as intuitive as expected. Since the developers are already used to JUnit tests and often have a scheme of the test already in mind, it is reasonable that they start writing tests and execute them separately without a `TestConfig` class. Therefore it would be better to provide all the annotations for the test classes and none solely for the `TestConfig` class. The `TestConfig` class will still be available in order to execute several tests with one instruction and to group these tests. Furthermore the configurations that can be made in the `TestConfig` class can still be done there for all the tests. On the build server the `TestConfig` class should be used, because there are many tests and therefore the grouping is most effective. On the local system the `TestConfig`class created for the build server can comfortably be used to execute all the tests and not only the own tests before the local changes are uploaded.

Most of the annotations present in test classes are used for fundamental parts of the test class. Therefore they are looked up and used during the test design process as soon as they are needed. The only annotation with a higher risk to be forgotten are `@RunWith` and `@UseDefaultConfiguration`. The test algorithm, the processors, the test input in the form of the graph and the configurator, and the test methods are the basic components of the test. If one of these components should not be used it can simply be omitted. The annotations that configure the output of the test can be specified as soon as this output is needed. Otherwise there is no output stored in order to waste no disc space.

That there are only the needed components present makes the tests readable and understandable. One observed drawback is that the classes can become very long, if there are many methods returning graphs. In these cases it can be sensible to create an abstract class with these methods and let the test class extend this class, or create an appropriate utility class.

In the following, two tests are presented that were created during the case study by one of the developers. There is one black box test and one white box test presented in order to examine how the tests are written, if they are written in the expected way, and whether there are weaknesses that can be observed.

Black Box Test

The test class presented in Algorithm 7.1 is created in order to test the node label placement. The ELK layered layout algorithm is executed in order to call the node size calculation, but the test is not specific for this layout algorithm.

As input there are graphs imported out of a directory that contains graphs especially created for the tests validating the node label placement. This directory is a sub-directory of the `klay_layered` directory. That there is a directory with graphs for a special layout algorithm corresponds to the expectations. It was furthermore expected that the tests are executed on different examples created for the algorithms and derived out of tools that use ELK. This is not done in this test, but there are no such graphs available until now. These graphs can be used by other test classes as well, but they will most likely be used only for a quite specialized kind of tests that concentrate on node labels. To use graphs that are specialized on the kind of tests they are used in is sensible, but as soon as there are more models that represent real world examples it is sensible to use some of them in addition to these specialized graphs.

In addition a layout configurator is used in order to make sure that all nodes are resized. Otherwise the test would not be sensible. With this configurator the test input is completely defined and there are not many lines necessary. Consequently the code is readable and comprehensible. Examples for unnecessary lines that can be avoided by another implementation of the framework are two lines in

```
1  @RunWith(LayoutTestRunner.class)
2  @Algorithm("org.eclipse.elk.layered")
3  public class NodeLabelElkTest {
4      @ImportGraphs
5      public List<ResourcePath> getImportSources() {
6          return Lists.newArrayList(
7                  new ModelsRepositoryResourcePath("graphs/klay_layered/node_label_placement/**"));
8      }
9      @Configurator
10     public LayoutConfigurator getLayoutConfigurator() {
11         LayoutConfigurator configurator = new LayoutConfigurator();
12
13         // Make sure that all nodes are resized such that everything fits into their bounding box
14         configurator.configure(ElkNode.class)
15                 .setProperty(CoreOptions.NODE_SIZE_CONSTRAINTS, SizeConstraint.free());
16
17         return configurator;
18     }
19     @Test
20     public void testLabelsInNode(final ElkNode graph) {
21         for (ElkNode child : graph.getChildren()) {
22             EnumSet<NodeLabelPlacement> nodeLabelPlacement = child.getProperty(CoreOptions.
                   NODE_LABELS_PLACEMENT);
23
24             for (ElkLabel label : child.getLabels()) {
25                 EnumSet<NodeLabelPlacement> actualLabelPlacement = nodeLabelPlacement;
26                 if (label.hasProperty(CoreOptions.NODE_LABELS_PLACEMENT)) {
27                     actualLabelPlacement = label.getProperty(CoreOptions.NODE_LABELS_PLACEMENT);
28                 }
29
30                 System.out.println(label.getWidth() + ", " + label.getHeight());
31
32                 if (actualLabelPlacement.contains(NodeLabelPlacement.INSIDE)) {
33                     Assert.assertTrue(isLabelInsideOfNode(label, child));
34                 }
35             }
36         }
37     }
38     private boolean isLabelInsideOfNode(final ElkLabel label, final ElkNode node) {
39         return label.getX() >= 0
40                 && label.getY() >= 0
41                 && label.getX() + label.getWidth() <= node.getWidth()
42                 && label.getY() + label.getHeight() <= node.getHeight();
43     }
44     @Test
45     public void testLabelsOutsideNode(final ElkNode graph) {
46         for (ElkNode child : graph.getChildren()) {
47             EnumSet<NodeLabelPlacement> nodeLabelPlacement = child.getProperty(CoreOptions.
                   NODE_LABELS_PLACEMENT);
48
49             for (ElkLabel label : child.getLabels()) {
50                 EnumSet<NodeLabelPlacement> actualLabelPlacement = nodeLabelPlacement;
51                 if (label.hasProperty(CoreOptions.NODE_LABELS_PLACEMENT)) {
52                     actualLabelPlacement = label.getProperty(CoreOptions.NODE_LABELS_PLACEMENT);
53                 }
54
55                 if (actualLabelPlacement.contains(NodeLabelPlacement.OUTSIDE)) {
56                     Assert.assertTrue(isLabelOutsideOfNode(label, child));
57                 }
58             }
59         }
60     }
61     private boolean isLabelOutsideOfNode(final ElkLabel label, final ElkNode node) {
62         return label.getX() >= node.getWidth()
63                 || label.getY() >= node.getHeight()
64                 || label.getX() + label.getWidth() <= node.getX()
65                 || label.getY() + label.getHeight() <= node.getY();
66     }
67 }
```

Algorithm 7.1. An example for a black box test

```
1  @RunWith(LayoutTestRunner.class)
2  @Algorithm(LayeredOptions.ALGORITHM_ID)
3  @RunAfterProcessor(
4                  processor = LongEdgeSplitter.class,
5                  onRoot = false)
6  public class LongEdgeSplitterElkTest {
7
8     @RandomGeneratorOptions
9     public GeneratorOptions getGeneratorOptions() {
10       GeneratorOptions options = new GeneratorOptions();
11
12       options.setProperty(GeneratorOptions.NUMBER_OF_NODES, RandVal.minMax(15, 30));
13       options.setProperty(GeneratorOptions.OUTGOING_EDGES, RandVal.minMax(1, 3));
14       options.setProperty(GeneratorOptions.NUMBER_OF_GRAPHS, 5);
15
16       return options;
17    }
18
19    @Test
20    public void ensureShortEdges(final LGraph lgraph) {
21       // Number all layers
22       List<Layer> layers = lgraph.getLayers();
23       Map<Layer, Integer> layerIndexMap = Maps.newHashMap();
24       for (int i = 0; i < layers.size(); i++) {
25          layerIndexMap.put(layers.get(i), i);
26       }
27
28       // Iterate over the graph's layers
29       for (Layer sourceLayer : layers) {
30          int sourceLayerIndex = layerIndexMap.get(sourceLayer);
31
32          // Iterate over all outgoing edges
33          for (LNode node : sourceLayer) {
34             for (LEdge edge : node.getOutgoingEdges()) {
35                int targetLayerIndex = layerIndexMap.get(edge.getTarget().getNode().getLayer());
36
37                Assert.assertTrue(
38                      targetLayerIndex == sourceLayerIndex
39                      || targetLayerIndex == sourceLayerIndex + 1);
40             }
41          }
42       }
43    }
44
45 }
```

Algorithm 7.2. An example for a white box test

a method that specifies a configurator. This is the line in which the configurator is created and the one with the return statement. For the example in Algorithm 7.1 these are the lines 11 and 17. If the configurator was handed over to the method as a parameter these two lines could be avoided.

The test method that verifies the graphs after layout iterates over the child nodes of the graph and asserts that every label is inside of its node. The assert has no failure message. This is more often the case than expected and has to be considered while the results are grouped and presented.

In general this test fits well into the framework and shows its advantages. Nevertheless even this small example shows room for improvement. Therefore there are small changes requested, but the general structure of the framework seems to be suitable for black box tests.

White Box Test

The white box test in Algorithm 7.2 is a very short and understandable example for a white box test. This test uses the random graph generator that provides a fast and easy way to specify graphs and there seem to be few demands on the generated graphs. Therefore this test does not need to be

executed on specialized graphs and as soon as there are directories with basic examples for the layout algorithms these could be used. The test method again uses an assert statement without a failure message.

The test class is a very simple white box test and shows how easily such a test can be written. There is another example in Section 8.1 that shows that there is a missing feature as well, but overall a white box test can be defined without much overhead.

7.3.3 Comparison to Predecessors

The new framework should replace the old solutions presented in Section 4.1.1 and should consequently provide the possibility to comfortably migrate the old tests. The framework is specialized on the verification of layout algorithms and therefore only the tests regarding layout algorithms can be migrated or a layout algorithm has to be used to call the tested feature.

For these tests it would have been possible to design the API of the new framework compatible to one of the old frameworks. A new framework designed in this way would execute the old tests without changes and would be able to interpret the old annotations. This was not intended by this work and is therefore not implemented. Consequently there are changes necessary and these are discussed in this section.

Most of the features provided by the `ModelCollectionTestRunner` are provided by the new framework as well. The annotation `@Models` can be replaced by a potentially repeated usage of the `@Graph` annotation. The annotations `@BundleId`, `@ModelPath` and `@ModelFilter` can be replaced by a method that is annotated with `@ImportGraphs` and returns a List of `ElkRepositoryResourcePaths` with the same bundle identifier and relative path. The new framework does not provide the possibility to specify a resource set. That could be added, but had not been considered necessary. The feature of the annotation `@StopOnFailure` is also not integrated in the new framework. The tests using this feature can be migrated, but all the test methods are executed after a failure. In comparison the features implemented in both frameworks can be used in the new framework with more freedoms. The filter can be specified more flexibly and it is possible to import graphs from other repositories as well.

The differences between the `KielerTestRunner` and the new framework are a bit bigger. Annotations are less used by the `KielerTestRunner` and the import of graphs is configured with variables of special classes. Nevertheless the graphs used in the `KielerTestRunner` are specified with a path and can therefore be imported in the new framework as well. The exclusion of sub-directories is integrated as an explicit feature in the `KielerTestRunner`, but not in the new framework. There a `FileFilter` can be used for this purpose. The configurators that are specified in the tests that are executed with the `KielerTestRunner` can be specified in the new test framework as well. All the possibilities provided by the old framework to specify the test input are supported by the new framework also, but in other ways. The execution of the layout algorithms is now done by the test runner and not by a `@Before` method in the test class. Black and white box tests are supported by both frameworks, but in the new framework the layout algorithm or the processor is specified by an annotation and not handled in a method in the class or in an abstract test class.

Taken all together the new framework provides nearly all the features that were provided by the old test frameworks and adds new features. That everything is done with annotations and not with naming conventions, as it was the case for some tests executed with the `KielerTestRunner`, leads to more freedom, more comprehensibility, and less overhead. In addition to that everything is done by the framework and the test classes do not need to extend any abstract test classes, but there is the freedom to provide additional features that way. One answers given by an experienced developer in the case study confirmed that the new framework is an improvement over the old solutions.

7.4 Execution Time Evaluation

In order to evaluate the performance, execution time measurements can only be used in order to give an impression of how much time the execution of a fixed number of test cases can take. The reason for that is the dependence of the execution time on the number and scope of the test cases.

There are two design decisions made with respect to the performance of the framework: the usage of a normal JUnit test, instead of a JUnit plugin test, and the grouping of several test cases with the same input and layout algorithm in one test case.

In the beginning the tests executed with the framework had to be executed as *JUnit plugin test*. This was justified by the fact that the ELK project is organized into plugin projects that are therefore normally executed in an Eclipse instance. The layout algorithms with their options are registered automatically if the test is executed as a plugin test. For such a test first the Eclipse instance is started, which takes a lot of time, and after that the test is executed, taking a fractional amount of the execution time. The time needed to start Eclipse can be saved if the test is executed as normal JUnit test. On that account the layout algorithms are registered manually in a separate class. In this class a new layout algorithm that should be tested by the framework can easily be added. The algorithms listed in this class are also the ones tested, if all layout algorithms known by the framework should be tested. After the tests had been changed to normal JUnit tests the execution time was noticeably reduced.

The next decision that was made due to efficiency reasons is the grouping of test cases. How this is done technically is explained in Section 6.8. The graph that is used as input of the grouped test case is imported or generated once instead of several times. In addition the test black box and white box runners iterate over the test cases in order to configure the graphs and to execute the tests. Therefore the reduced number of test cases leads to less configurations of graphs and less executions of the layout algorithm. It is important that the overhead of the comparison of test cases in order to group them does not outweigh the time saved by the reduced number of iterations. Therefore only the used resource paths are compared, because this is less complex than the comparison of `ElkNodes` and `GeneratorOptions`.

There were some measurements done in order to estimate the gain in efficiency. These were done in five categories: Imported graphs with and without grouping, generated graphs with and without grouping, and graphs created by a method as comparison. For each of the categories the measurements were done twelve times, with the first two times were not taken into account. The time is measured in the `run()` method of the `LayoutTestRunner`. Therefore the import of the test classes that is done in the constructor of the runner and the grouping of the results and their output is not taken into account. This is no problem, because these parts are executed with very small differences for the different categories.

The execution times of the categories are listed in Table 7.1.

Imported graphs There are ten graphs imported in a test class with one test method that iterates twice over the graph. The graphs are stored in different files, but the resulting `ElkNodes` are the same. The graphs consist of 10 nodes and 20 edges. The test class is executed 10 times by the usage of a `TestConfig` class. Without grouping there are 100 test cases with one test method and with grouping there are 10 test cases with two test methods.

This setup is equivalent to 10 test classes created for one layout algorithm and all of these test classes are executed on the same ten test graphs and configurators. Executing the test classes on a set of default graphs would lead to the usage of exactly the same graphs for several test cases. These graphs are most likely stored in a folder created for the graphs used as default for a layout algorithm. If there is a set of default configurators for a layout algorithm as well they can be

specified in an abstract test class for the algorithm that can be extended by the test classes created for this algorithm.

It is expected that there are such sets of default graphs with more than ten graphs and a higher number of test classes. Assume there are n test classes all using the same set of default graphs and a graph is added to this set. If the tests are not grouped there are n new test cases added and if the tests are grouped there is only one new test case. Therefore there is the possibility to save more time if there are more test classes. In addition for each graph in this set time is saved and therefore the absolute value of saved time is higher for a higher number of graphs.

On the other hand the tests are also executed on some specialized graphs besides of the default set and that will lower the effect. Therefore this setting is selected in order to gain an indication as to the possible impact of the grouping that will be influenced by many factors in the final setting.

The times measured are:

Without Grouping Without grouping the execution time was on average 1494.6ms. The minimal time was 1381ms and the maximal time was 1776ms.

With Grouping With grouping the execution time was on average 449.8ms. The minimal time was 427ms and the maximal time was 472ms. The average execution time with grouping is 30 percent of the average execution time without.

Generated graphs The other case in which tests are grouped are generated graphs with a file specified in order to load the generator options. There are ten files with random generator options used by one test class with one test method that iterates twice over the graph. The options specified in the file are all the same and lead to the generation of graphs with 9 to 11 nodes and 18 to 22 edges. There are 100 test cases without grouping and 10 test cases with grouping. The size of the graphs is similar to the ones used above.

One drawback is that there are only randomly generated graphs grouped that were created based on an `.elkr` file. Therefore this setup might not be realistic, because it is more comfortable to specify the options in an instance of `GeneratorOption` created in a method in the test class. Therefore I propose the following approach. Especially in the beginning of the frameworks usage the random graph generator can be useful, because there are not so many test graphs available. In this situation it seems sensible to proceed as follows:

- Create the folders for default test graphs with a structure that can be used as final setting. There can be one folder for each layout algorithm or one folder for each source of graphs or for each sort of graphs.
- For each folder that holds fewer graphs than needed create one or more `.elkr` files in order to generate the files that are not yet available.
- Load all the graph files and random generator files available in the folder as input for the test class.
- As soon as there are more graphs available the `.elkr` files can be reduced or deleted.

Apart from the grouping of test cases, the advantage of this procedure is that there are no changes in the test classes needed as the model database is stocked up.

The times measured are:

Without Grouping Without grouping the execution time was on average 1451.5ms. The minimal time was 1336ms and the maximal time was 1627ms.

With Grouping With grouping the execution time was on average 586.1*ms*. The minimal time was 529*ms* and the maximal time was 645*ms*. The average execution time with grouping was 40 percent of the average execution time without.

Graphs created in a method The test cases with tests created in a method are not grouped. Their execution times are measured as comparison values. The created graph is the same one as the one imported in the setting described above. In the test class 10 graphs are created and the class is executed 10 times hence there are 100 test cases that can not be grouped.

For the creation of graphs no grouping is possible and consequently only the time without grouping is available. The measured average was 1313.9*ms*. The minimal time was 1208*ms* and the maximal time was 1426*ms*.

Table 7.1. The measured execution times in ms.

Setting	Without Grouping	With Grouping
Imported graphs	1494.6	449.8
Generated graphs	1451.5	586.1
Created graphs	1313.9	–

The times measured above show that a great amount of time can be saved. Therefore the tests should be designed in a way that they can be grouped. Especially the creation of abstract test classes with default test graphs imported as explained above and the default configurators defined can help to design the tests in the desired way. The usage of methods that return the graphs directly should be restricted on graphs specialized for one test class.

Chapter 8

Conclusion

The goal of this theses was the development of a test framework that makes it as easy and comfortable as possible to create tests for the layout algorithms available in the ELK project. First of all the testing processes and testing strategies in general were examined, and other test frameworks and the testing strategies in other projects with automatic graph layout were considered as orientation. In addition to that, the situation in the ELK project was assessed together with the preceding test frameworks. The results were used together with the requirements to derive which features are important for the framework and how they should be implemented and be made accessible.

The framework is based on JUnit and provides a test runner that can be used to execute JUnit test classes. The test runner provides annotations that can be used in the test classes to access the features provided by the framework. The features can be categorized into *test input*, *test execution*, and *output of results*.

The test input consists of a graph and optionally a configurator. The are three different possibilities to provide the graphs: creating them in a method, importing them from files, and generating them randomly. The provided graphs can be configured by the usage of a *configurator* or a *configuration method*. These can be used to set options on an input graph. In addition to that the framework provides the possibility to configure the graphs with default values.

The tests executed by the framework can be executed as *black box tests*, *white box tests*, or *analysis tests*. A black box test is used if the result of the whole layout algorithm should be verified. A white box test is used if the output of the *processors* the layout algorithm is composed of should be tested. An analysis test is used in order to verify that the quality of the layout results does not fall under a minimal acceptable level. The tests can be executed by the developer in the Eclipse IDE and on the build server every time the software is built.

The results of the tests are displayed in the Eclipse IDE and printed on the console. In addition to that it is possible to store the results and the graphs that lead to failures in a file. This enables the possibility to debug the problems that caused a test to fail.

The framework was evaluated regarding performance and usability. There are two design decisions that were motivated by performance improvement: the execution of the test framework as a plain Java JUnit test as opposed to a JUnit plugin test and the grouping of tests. Both of these decisions lead to a notable performance improvement. In order to evaluate the usability, the main developers of the ELK project used the framework and estimated the usability in terms of efficiency, effectivity, and satisfaction. Overall the framework was evaluated as an appropriate solution and an improvement compared to its predecessors.

8.1 Future Work

There are some features not yet implemented in the framework that proved desirable during the development and the evaluation of the framework. These features are motivated and explained in this section.

8.1.1 Execution Before and After Phases

One feature that was missed by the developers during the evaluation is the possibility to execute a test before or after a phase. A phase in a layout algorithm normally has different implementations. Each of these implementations is a processor that can be specified in the `@RunBefore` or `@RunAfter` annotation. If a phase has several implementations and a test method should be executed after the phase independent of its implementation, it is necessary to specify each of the processors. Therefore an annotation is needed with the phase as attribute.

One advantage of such an annotation is that it is not necessary to adjust the tests if a new implementation of a phase is added. In the actual implementation the new processor needs to be added to each test that should be executed before or after the phase.

In addition to that it may be useful to specify that each of the graphs should be configured in a way that it will be laid out once with each of the phase implementations. This should again not be done manually with three separate configurators, but with one annotation that specifies the phase, instructing the framework to look for the implementations and generates the configurators.

8.1.2 Default Values

Most developers do not like to write tests and therefore they are happy about each line of code that can be omitted during the implementation of a test. Consequently the usage of default values would be useful. One sensible example would be the specification of a default set of input graphs and configurators for each algorithm. This could be implemented as part of the framework and annotations could be provided, such as `@UseDefaultGraphs(ElkLayered)` for default graphs specified for `ElkLayered`. Accordingly default graphs for all algorithms or sets of graphs derived from other projects could be specified.

An alternative to that are abstract test classes that could define the default inputs for layout algorithms.

8.1.3 Derive Graph Properties

Often the properties of a graph are interesting. One example is a test that must not be executed on graphs with cycles. If the graphs used in this example should additionally be derived from examples in projects using ELK it is necessary to specify exactly the files that contain graphs with these properties. In this situation each graph with cycles has to be filtered out. Specifying the appropriate graph files in the test cases is laborious and error-prone and it would be more comfortable to specify the properties the graphs should have and to add them as a filter to the `ResourcePath`. In addition it can be useful to extract the properties from graphs the test failed on. If for example all the graphs that lead to failed tests contain a cycle it can be useful to display this property together with the results.

For the graphs stored in files it is possible to store the derived properties, but they need to be updated if the files are changed. For the created and generated graphs the properties have to be derived from the graph every time again in order to output the properties of the graphs the tests failed on. It has to be evaluated if the performance of the framework is changed significantly by the derivation of the properties. Especially if the properties of the graphs stored in files are derived every time again for the determination of the input.

8.1.4 Performance Measurement

The performance measurement can be included in the framework in order to ensure that the performance does not fall below a minimal acceptable level. This can be integrated into the framework in a way that the execution time of the algorithm is measured each time the layout algorithm is executed. This would lead to as many measurements as possible without additional executions of the layout algorithm. The drawback of this solution is that an influence of the test cases and the complexity of their inputs on the execution time exists. Therefore the performance can only be compared to other test runs with the same test cases. In addition to that the execution time is influenced by other factors and therefore the limit must not be defined very strictly.

Another possibility is to define separate execution time tests but that leads to more executions of the layout algorithms and there have to be enough executions in order to ensure that the measurements are reliable.

8.1.5 Debug Mode

It is possible to set an option on the graphs in order to execute the layout algorithm in debug mode. This can be useful in order to look for a fault. The automatic execution of the failed test cases in debug mode after the normal execution of all tests could be enabled with an annotation or started by an input on the console.

8.1.6 Skip Test Methods

Sometimes it is not sensible to execute the remaining test methods of a class, if one of the methods executed before failed. For these cases it would be useful to attach an annotation to the method that makes the execution of subsequent ones superfluous. This annotation would be similar to the `@StopOnFailure` annotation implemented in the `ModelCollectionTestRunner` and the implementation should not be very intricate.

Bibliography

[Bey17] Stephan Beyer. 2017. URL: http://banditcpp.org/writingtests.html.

[Bla96] Alan F. Blackwell. "Metacognitive theories of visual programming: what do we think we are doing?" In: *Proceedings of the 1996 IEEE Symposium on Visual Languages*. 1996, pp. 240–246. DOI: 10.1109/VL.1996.545293. URL: http://dx.doi.org/10.1109/VL.1996.545293.

[Bro96] John Brooke. *Sus: a quick and dirty usability scale*. 1996.

[CGJ+13] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. "The Open Graph Drawing Framework (OGDF)". In: *Handbook of Graph Drawing and Visualization*. Ed. by Roberto Tamassia. CRC Press, 2013, pp. 543–569.

[Cry15] James Cryer. *Pro Grunt.js*. Apress, 2015.

[CS94] David A. Carrington and Phil Stocks. "A tale of two paradigms: formal methods and software testing". In: *Z User Workshop, Cambridge, UK, 29-30 June 1994, Proceedings*. 1994, pp. 51–68.

[DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, eds. *Structured programming*. London, UK, UK: Academic Press Ltd., 1972. ISBN: 0-12-200550-3.

[DET+99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph drawing: Algorithms for the visualization of graphs*. Prentice Hall, 1999. ISBN: 0-13-301615-3.

[DN81] Joe W. Duran and Simeon Ntafos. "A report on random testing". In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE '81. San Diego, California, USA: IEEE Press, 1981, pp. 179–183. ISBN: 0-89791-146-6. URL: http://dl.acm.org/citation.cfm?id=800078.802530.

[DS16] Sandeep Desai and Abhishek Srivastava. *Software testing: a practical approach*. PHI Learning, 2016.

[Ecl17] Eclipse. 2017. URL: http://wiki.eclipse.org/Rich_Client_Platform.

[FH10] Hauke Fuhrmann and Reinhard von Hanxleden. *Taming graphical modeling*. Technical Report 1003. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2010.

[Fow17] Martin Fowler. 2017. URL: https://www.martinfowler.com/articles/continuousIntegration.html.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. "Graph drawing by force-directed placement". In: *Software—Practice & Experience* 21.11 (1991), pp. 1129–1164. ISSN: 0038-0644. DOI: http://dx.doi.org/10.1002/spe.4380211102.

[GL16] Chun Guo and Xiaozhong Liu. "Dynamic feature generation and selection on heterogeneous graph for music recommendation". In: *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*. 2016, pp. 656–665. DOI: 10.1109/BigData.2016.7840658. URL: http://dx.doi.org/10.1109/BigData.2016.7840658.

[Goo17] Google. 2017. URL: https://github.com/google/googletest/blob/master/googletest/docs/Primer.md.

[Hal88] P. A. V. Hall. "Towards testing with respect to formal specification". In: *Second IEE/BCS Conference: Software Engineering, 1988 Software Engineering 88.* July 1988, pp. 159–163.

[HR07] Martin Höst and Per Runeson. "Checklists for software engineering case study research". In: *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, ESEM 2007, September 20-21, 2007, Madrid, Spain*. 2007, pp. 479–481. DOI: 10.1109/ESEM.2007.46. URL: http://dx.doi.org/10.1109/ESEM.2007.46.

[ISO13a] ISO/IEC JTC1/SC7. *29119-1-2013 - software and systems engineering software testing part 1:concepts and definitions*. 2013.

[ISO13b] ISO/IEC JTC1/SC7. *29119-2-2013 - software and systems engineering software testing part 2:test processes*. 2013.

[ISO13c] ISO/IEC JTC1/SC7. *29119-3-2013 - software and systems engineering software testing part 3:test documentation*. 2013.

[ISO15] ISO/IEC JTC1/SC7. *29119-4-2015 - iso/iec/ieee international standard for software and systems engineering–software testing–part 4: test techniques*. 2015.

[ISO16] ISO/IEC JTC1/SC7. *29119-5-2016 - iso/iec/ieee international standard - software and systems engineering – software testing – part 5: keyword-driven testing*. 2016.

[ISO98] ISO/TC 159/SC 4. *Ergonomics of human-system interaction - part 11: usability: definitions and concepts*. 1998.

[Key03] Jessica Keyes. *Software Engineering Handbook*. Auerbach, 2003.

[Kla12] Lars Kristian Klauske. "Effizientes Bearbeiten von Simulink Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus". PhD thesis. Technische Universität Berlin, 2012.

[Kle13] Stephan Kleuker. *Qualitätssicherung durch Softwaretests*. Springer, 2013.

[KPP95] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. "Case studies for method and tool evaluation". In: *IEEE Software* 12.4 (1995), pp. 52–62. DOI: 10.1109/52.391832. URL: http://dx.doi.org/10.1109/52.391832.

[KS07] Jochen Ludewig Karol Frühauf and Helmut Sandmayr. *Software-Prüfung - eine Anleitung zum Test und zur Inspektion*. vdf, 2007.

[Man11] Tim O'Brien Manfred Moser. *The Hudson Book*. Oracle, Inc., 2011.

[Mat07] Aditya P. Mathur. *Foundations of software testing: fundamental algorithms and techniques*. Pearson India, 2007.

[Mye97] Glenford J. Myers. *The art of software testing*. John Wiley and Sons, 1997.

[Pau07] Andrew Glover Paul M. Duvall Steve Matyas. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.

[Pet95] Marian Petre. "Why looking isn't always seeing: Readership skills and graphical programming". In: *Communications of the ACM* 38.6 (June 1995), pp. 33–44.

[PTK16] Aaron S. Pope, Daniel R. Tauritz, and Alexander D. Kent. "Evolving random graph generators: A case for increased algorithmic primitive granularity". In: *2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016, Athens, Greece, December 6-9, 2016*. 2016, pp. 1–8. DOI: 10.1109/SSCI.2016.7849929. URL: http://dx.doi.org/10.1109/SSCI.2016.7849929.

[Pur02] Helen C. Purchase. "Metrics for graph drawing aesthetics". In: *Journal of Visual Languages and Computing* 13.5 (2002), pp. 501–516.

[Pur97] Helen C. Purchase. "Which aesthetic has the greatest effect on human understanding?" In: *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*. Vol. 1353. LNCS. Springer, 1997, pp. 248–261.

[PW15] Robert Prediger and Ralph Winzinger. *Node.js: Professionell hochperformante Software entwickeln*. Carl Hanser Verlag GmbH & Co. KG, 2015.

[PY09] Mauro Pezzè and Michal Young. *Software testen und analysieren*. Oldenbourg, 2009.

[RUC+99] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. "Test case prioritization: an empirical study". In: *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*. 1999, pp. 179–188. DOI: 10.1109/ICSM.1999.792604.

[SC93] P. A. Stocks and D. A. Carrington. "Test templates: a specification-based testing framework". In: *Proceedings of 1993 15th International Conference on Software Engineering*. May 1993, pp. 405–414. DOI: 10.1109/ICSE.1993.346025.

[Sch11] Christoph Daniel Schulze. "Optimizing automatic layout for data flow diagrams". Diploma Thesis. Kiel University, Department of Computer Science, July 2011.

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Just model! – Putting automatic synthesis of node-link-diagrams into practice". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)*. San Jose, CA, USA, Sept. 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

[SSH14] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. "Drawing layered graphs with port constraints". In: *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout* 25.2 (2014), pp. 89–106. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2013.11.005.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. "Methods for visual understanding of hierarchical system structures". In: *IEEE Transactions on Systems, Man and Cybernetics* 11.2 (Feb. 1981), pp. 109–125.

[Tam13] Michael Tamm. *Junit-Profiwissen*. dpunkt.verlag, 2013.

[Tan76] Andrew S. Tanenbaum. "In defense of program testing or correctness proofs considered harmful". In: *SIGPLAN Not.* 11.5 (May 1976), pp. 64–68. ISSN: 0362-1340. DOI: 10.1145/956003.956011. URL: http://doi.acm.org/10.1145/956003.956011.

[TLM+10] Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. *Junit in Action*. Manning Publications, 2010.

[TYS+03] W. T. Tsai, L. Yu, A. Saimi, and R. Paul. "Scenario-based object-oriented test frameworks for testing distributed systems". In: *The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems, 2003. FTDCS 2003. Proceedings.* May 2003, pp. 288–294. DOI: 10.1109/FTDCS.2003.1204349.

[VME+12] Tanja E. J. Vos, Beatriz Marin, Maria Jose Escalona, and Alessandro Marchetto. "A methodological framework for evaluating software testing techniques and tools". In: *2012 12th International Conference on Quality Software, Xi'an, Shaanxi, China, August 27-29, 2012*. 2012, pp. 230–239. DOI: 10.1109/QSIC.2012.16. URL: http://dx.doi.org/10.1109/QSIC.2012.16.

[WPC+02] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. "Cognitive measurements of graph aesthetics". In: *Information Visualization* 1.2 (2002), pp. 103–110. ISSN: 1473-8716.

Acronyms

ELK Eclipse Layout Kernel
An Eclipse project providing automatic layout algorithms for the usage in diagram editors. Available at: `http://www.eclipse.org/elk`

GrAna Graph Analysis
A project in ELK that implements analyses that can be performed on graphs.

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client
A research project considering the graphical model-based design of complex systems.

SCCharts Sequentially Constructive Statecharts
A visual synchronous language using a synchronous model of computation. Available at: `https://rtsys.informatik.uni-kiel.de`

OGDF Open Graph Drawing Framework
A C++ library providing different layout algorithms. Available at: `http://www.ogdf.net`

Graphviz Graph Visualization Software
An open source graph visualization software. Available at: `http://www.graphviz.org`

IDE Integrated Development Environment
An application supporting the software development process.

SUS System Usability Scale
A questionnaire for the "quick and dirty" evaluation of a system's usability.

Appendix A

Technical Documentation for the Test Framework

The following documentation provides a short overview of the implemented features of the framework and a description of how to use them.
One thing to keep in mind while naming the tests is the naming convention. A test, that should be automatically executed by Surefire (on the build server) together with the other tests, has to have the prefix or suffix `ElkTest`.

A.1 The Layout Algorithm

The tests in a class have to test the same layout algorithm(s). This has to be specified in a class annotation. It is possible to specify one or more algorithms, no algorithm at all or all, layout algorithms known by the test framework. The algorithm is specified with the help of the `@Algorithm` annotation that is specified in `org.eclise.elk.test.LayoutTestRunner`.
One or more algorithm:

```
@Algorithm("org.eclipse.elk.force")
@Algorithm("org.eclipse.elk.layered")
public class ImportantElkTest { ...
```

The algorithms are simply specified with the `@Algorithm` annotation. This annotation can be repeated.
All known algorithms:

```
@Algorithm("*")
public class UniversalElkTest { ...
```

New algorithms not yet known by the framework, have to be added in the class `PlainJavaInitialization` in the package `org.eclipse.elk.test.framework`.
If the algorithm shouldn't be specified in the annotation, this can be done simply by omitting the `@Algorithm` annotation. In this case the default algorithm will be used or the algorithm or algorithms can be specified in a configurator, as will be explained in Section A.2.2.

A.2 The Input

First of all an input for the tested layout algorithms is required. All the annotations usd to specify the input of tests are implemented in `org.eclipse.elk.test.GraphProvider`.

A.2.1 Graphs

There are different possibilities to provide graphs for the test runs. The graphs can be explicitly built by one or more methods, the import paths can be specified, or the graphs can be generated randomly. Annotatoins are available for all these possibilities and the annotated methods have to return values of the expected type.

Building Custom Graphs

One possibility is to implement one ore more methods with an `ElkNode` as return value. These methods have to be annotated with `@Graph` and return an `ElkNode`.

```
@Graph
public ElkNode getGraph() {...
```

Import Graphs

In case there are files with graphs available they can be used as test input. It is possible to specify single files or directories. To specify the location a class extending `ResourcePath` is used. This class uses a System Property to specify where the framework has to look for the file. In case of an `ElkRepositoryResourcePath` the system property "ELK_REPO" is used. This System Property has to specify the location of the ELK repository. If the files are in a special models repository, the class `ModelsRepositoryResourcePath` can be used and the System Property "MODELS_REPO" has to be specified.
The two parameters passed to the constructors are a String, `filePath`, specifying the path relative to the repository path. In case `filePath` ends with "/**" all the files in the specified directory are imported and in case the `filePath` ends with "/**/" the files in this directory and the sub-directories are imported. The second parameter is a `FileFilter`. This is used to filter the files in case the filePath ends with "/**" or "/**/". The class `FileExtensionFilter` is available to filter the files depending on their extension.
The annotated method has to return a `List` of `ResourcePaths`.

```
@ImportGraphs
public List<ResourcePath> getImportSource() {
   List<ResourcePath> list = new ArrayList<>();
   list.add(new ModelsRepositoryResourcePath("graphs"
                + File.separator + "without-ports/**", null));
   list.add(new ModelsRepositoryResourcePath("graphs"
                + File.separator + "hierarchy01.elkg", null));
   return list;
}
```

Random Graph Generator

Another possibility is to generate the graphs used for the tests. For this purpose the random graph generator is used. It is possible to specify options in an `.elkr` file (like the `.rdg` files before) or to return `GeneratorOptions` in a method.
To use an .elkr file a `ResourcePath` has to be returned and the method has to be annotated with

`@RandomGraphFile`.
The other possibility is to implement a method that returns an instance of `GeneratorOptions` and to annotate this method with `@RandomGeneratorOptions`.

```
@RandomGraphFile
public ResourcePath getFileName(){...
```

```
@RandomGeneratorOptions
public GeneratorOptions getOptions(){...
```

A.2.2 Configurators

As soon as the graphs are present they can be configured. It is possible to implement methods returning a `LayoutConfigurator` or to specify methods, which configure a graph directly.
If a method returns a `LayoutConfigurator` the configurator is applied on every graph specified for the test class. The method has to be annotated with `@Configurator`. If there are several methods with this annotation each of the configurators is separately applied on each graph and the tests are executed on each pair.
The other possibility is to use a method with an `ElkNode` as input parameter and the annotation `@ConfigFacility`. In this method the developer can configure a graph with much more controll. The method is invoked with every graph and there are several methods with this annotation allowed. Therefore each test is executed on each combination of a graphs and a single configuration method.

```
@Configurator
public LayoutConfigurator getConfigurator() {
```

```
@ConfigFacility
public void configure(ElkNode graph) {
```

A.3 The Tests

A.3.1 White Box Tests

A white box test is until now just possible for the ELK Layered algorithm. The algorithm is divided in processors. The processors run one after the other and can be executed several times. Their preconditions and outcomes can be tested with a white box test. There are annotations available to specify the processors as class annotation or as method annotation. The annotations are `@RunBeforeProcessor` and `@RunAfterProcessor` and they are implemented in `org.eclipse.elk.test.LayoutTestRunner`. They have two parameters. One of the parameters is the processor and the other is a boolean value `onRoot`. The value `onRoot` specifies, whether the test should be executed before/after every execution of the processor, or just before/after the invocation of the processor on the root graph. In case one of the annotations is used as class annotation, all the test methods in the class that do not specify something else are executed before/after the processors. In case the annotations are used as method annotations the method is executed just before/after the processors specified in the method annotations. The annotations can be used repeatedly.
For every test method it is counted how often it is executed on how many graphs. In case a method is

not executed at all, because the processor is not executed, it will come to a test failure. The annotation `@FailIfNotExecuted` can be used, if a not executed test should not result in a failure. This annotation is specified in `org.eclipse.elk.test.LayoutTestRunner`.
A white box test has to have one or two parameters. The first one is the graph and the second one is the processor. The graph has the type of the graph used by the tested layout algorithm. This is often not an `ElkNode`, but for example the `LGraph` for the `ElkLayered` algorithm. The processor is the one that is about to execute or finished its execution. This can be used to find out by which processor the execution has been triggered.

```
@RunBeforeProcessor(
  processor=org.eclipse.elk.alg.layered.p4nodes.bk.BKNodePlacer.class,
  onRoot=false)
@RunAfterProcessor(
  processor=org.eclipse.elk.alg.layered.p1cycles.GreedyCycleBreaker.class,
  onRoot=false)
public class ElkTest1 {...
```

A.3.2 Black Box Tests

Every test class without the annotations `@RunBeforeProcessor` and `@RunAfterProcessor` is executed as black box test. In this case the layout algorithm is executed and afterwards the tests are executed on the resulting graph.

```
@Algorithm("org.eclipse.elk.layered")
public class OverlapElkTest {
...
@Test
public void testNodesOverlaps(ElkNode graph) {...
```

A.3.3 Analysis Tests

An analysis test has to be a black box test. An analysis test have to be annotated with `@Analysis` and should extend the class `AnalysisTest`. `AnalysisTest` provides a test method which compares the results of an analysis with maximal and minimal allowed values. The limit values are specified in an instance of `AnalysisConfiguration`, which has to be returned by a method in the analysis test. This method has to be annotated with `@AnalysisConfig`.

```
@AnalysisConfig
public AnalysisConfiguration getConfiguration() {...
```

The test method in `AnalysisTest` expects the analysis to return an array of values. The length of the array is specified in the instance of `AnalysisConfiguration`. There is as well specified which values of the array should be compared to the specified maximal and minimal limits and which analysis should be used.
The other values in this class refer to the results file. This is a file in which the results of an old analysis are stored to be used as comparison values for later analyses. A `boolean` value in `AnalysisConfiguration` specifies whether to store the new results. The results are stored only, if this value is `true` and all the

analysis tests on this algorithm are successful. In case the specified old results file does not exist or does not contain comparison results for the current combination of algorithm, graph, and configurator, the test will fail.
The annotation `@Analysis` is implemented in `org.eclipse.elk.test.GraphProvider` and the annotation `@AnalysisConfig` in `org.eclipse.elk.test.BlackBoxRunner`.

A.4 Run The Tests

With the framework a single test can be executed like a usual JUnit tests and the test class has to be annotated with `@RunWith(LayoutTestRunner.class)` that is provided by JUnit.
In case there should be more than one test class executed by the framework, an additional class can be used. This *TestConfig*-class is used to specify the tests to be executed together with some configurations valid during the execution of all the tests. Like a JUnit *Suite* it provides the facility to execute a group of tests in one test run instead of one test run per test class. The name of the TestConfig class has to have a name that start with "ElkTestConfig" or a name that neither ends nor starts with "ElkTest". The tests selected for execution can be specified by explicitly listing the test classes or by specifying a parent directory in which the `.class` files are located. The path of the directory has to be specified relative to the ELK repository. For Surefire the path should be "test" and all the tests have to be located in its sub-directories. The two options can also be combined and the explicitly specified test classes are added to the classes found in the directory. The annotation used to list test classes is `@TestClasses` and the annotation used to specify the directory is `@TestPath`. This "TestConfig"-class has to be annotated with `@RunWith(LayoutTestRunner.class)`.

```
@RunWith(LayoutTestRunner.class)
@TestPath("test")
@TestClasses({ElkTest1.class, ElkTest2.class})
public class TestConfiguration {...
```

The two annotations are implemented in the class `org.eclipse.elk.test.LayoutTestRunner`.

A.5 The Results

There are results and failed graphs that can be stored after a test run. The paths for the results are specified relative to the path in the system property or environment variable `RESULTS_PATH`. If this is not specified the results and failed graphs can't be stored and the results of the tests are just printed on the console.

A.5.1 A Result File

The results can be written to a text file. In this file the combinations of layout algorithms, graphs, configurators and the tests executed on it are listed. Failed tests are stored in the beginning of the file together with the failure. Whether the results should be written to such a file can be specified by an annotation in the "TestConfiguration" class. This is the Annotation `@StoreResults` implemented in `org.eclipse.elk.test.LayoutTestRunner`. In case this annotation is present with the value `true` the results are stored and with `false` they are not. The default value for the parameter is `true` and in case the annotation is not present the results are not stored.

A.5.2 Faulty Graphs

It is possible to store the graphs a test fails on. The graphs are stored in the sub-directory `failed-graphs` in the directory specified by the system property `RESULTS_PATH`. Whether the graphs should be stored can be specified with the annotation `@StoreFailedGraphs` that is implemented in the Java class `org.eclipse.elk.test.LayoutTestRunner`. This annotation has a boolean parameter with default value true and the graphs are stored, if this parameter is true. In case the "TestConfig"-class is not annotated with this annotation the graphs are not stored.

A.5.3 Analysis Results

The Analysis results are stored in a sub-directory of `RESULTS_PATH`, as explained in Section A.3.3.

A.6 Required System Properties

There are three System Properties used to specify paths.

ELK_REPO Set to the path of the ELK repository

MODELS_REPO Set to the path of the models repository in case the graphs are stored in a separate repository.

RESULTS_PATH Set to the path the results should be stored at

A.7 White Box Tests for other Algorithms

To make it possible to test other algorithms with white box tests, there are some changes in the layout provider and the layout algorithm necessary. Principally this is possible for all layout algorithms structured in `ILayoutProcessors`. The layout provider needs to implement the interface `IWhiteBoxTestable`. The method `setTestController` in this interface is used to pass the `TestController` to the layout algorithm. The `TestController` has to be used by the Layout Algorithm to trigger the execution of the test. Before and after each `ILayoutProcessor` is executed the layout algorithm has to call methods in `TestController`. By these methods listeners are notified, if they are waiting for the processors execution. The interface `ILayoutProcessor` and the class `TestController` are located in `org.eclipse.elk.core.alg`.

List of Figures

List of Tables

List of Algorithms