Sonargraph User Manual

Version 15.0.0

Sonargraph User Manual: Version 15.0.0

Table of Contents

Chapter 1. Motivation for Code Quality

The main idea behind *Sonargraph* has always been to provide a tool that eases the creation and maintenance of high-quality software. Creating high-quality software is difficult: You need to know where the pain-points are and how to solve them.

For any serious project that must live longer than a couple of months, it is actually cheaper to spend part of your resources to keep your software constantly at a good level of quality than using all your time to create new features. Martin Fowler explains this very well in his article "Is High Quality Worth the Cost?"[1]. The bottom line is, that apart from the very early development stages, high-quality software is actually cheaper to develop, because it allows adding new features at almost constant speed, whereas it becomes more and more time consuming to add new features into a code base with low quality.

We at hello2morrow believe that a consistent architecture is a fundamental part of software quality. When we use the term "architecture", we think of it in terms of the IEEE 1471 standard:

> "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."

This chapter describes why architectural design as an activity is needed, why conformance checks need to be done automatically by a tool and how Sonargraph supports you as a developer and architect during these activities.

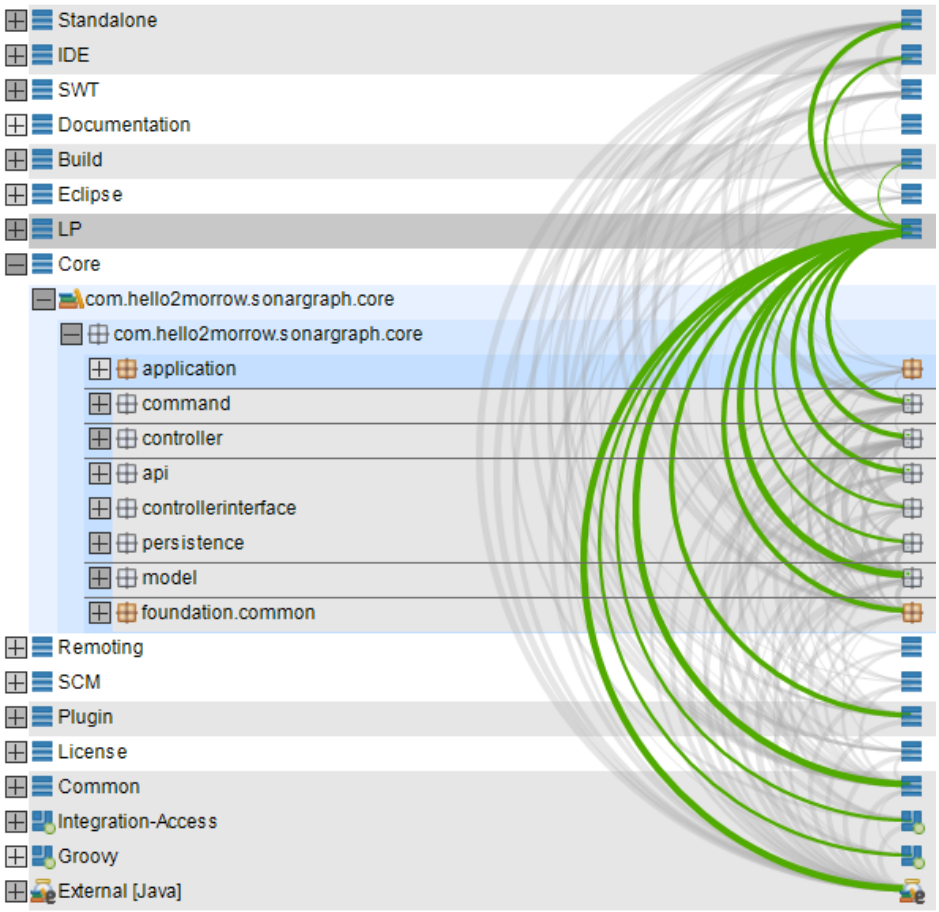


Figure 1.1. Visualizing Defined Architecture and Existing Dependencies in the Architectural View

[1] *"Is High Quality Worth the Cost?"* , https://martinfowler.com/articles/is-quality-worth-cost.html, 2019

The Need for Architecture

Martin Fowler wrote an excellent article "Is Design Dead?"[2] back in 2004. He argues that for anything serious, you cannot just code along and hope for the best, but need “planned design”:

> "If you want to build a doghouse, you can just get some wood together and get a rough shape. However if you want to build a skyscraper, you can't work that way - it'll just collapse before you even get half way up."
>
> - Martin Fowler

He is not arguing to design everything up front, but rather making the design activity part of the agile development process. As a consequence, the architecture evolves together with the code base and the knowledge of the team.

If you take the definition for architecture mentioned at the start of the chapter, then the top-level architecture should contain the main components and their dependencies. As with construction architectures for large buildings, no single diagram exists that contains all information for a large-scale software system. There is an upper limit of elements that our brain can process, especially if there are also interconnections of different types between elements. Thus, if the system grows beyond a certain size, abstractions are needed which can be thought of as maps at different scale. Simon Brown gives an example of this with the C4 model[3]. Where details are needed, additional diagrams can be created.

Of course, several viewpoints for software architecture exists as described by "The 4+1 ViewModel"[4]. This "static" architecture that describes the decomposition of a system in its parts is the foundation for the "dynamic" aspects like information flow: If there is no direct dependency between two components or between them and any of their commonly shared components, there cannot be any information flow between them.

Consistency of the diagrams now becomes a challenge: Higher-level abstractions must not be violated at the lower level: There must not be a secret tunnel at the detailed level where the higher level puts a clear barrier between components.

The last decades of agile software development have shown that it is impossible and impractical to do a big design upfront. Not only the requirements from the outside (read 'business') may change, but most certainly the developers' understanding of the domain improves over time and thus the architecture likely also needs to be adapted as a consequence. The effort needed to change a functionality and the effects this change causes for the rest of the system very much depend on the number of usages of this functionality.

Design for changeability therefore means to minimize coupling (i.e. the number of dependencies) between elements, because elements with low coupling can be more easily re-arranged. Especially bad for coupling are cyclic dependencies that may cause mental challenges in form of hen-and-egg problems and also make it harder to understand a system's structure if a large number of elements are involved. You want to avoid an entangled mess as shown in the following picture (where green arcs represent dependencies between Java packages and the direction of dependencies is counter-clockwise) that is often described as "big ball of mud":

[2] *"Is Design Dead?"* , https://martinfowler.com/articles/designDead.html, 2004
[3] *C4 Model* , https://c4model.com/
[4] *"Architectural Blueprints—The '4+1' ViewModel of Software Architecture"* by Philippe Kruchten, https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf, 1995

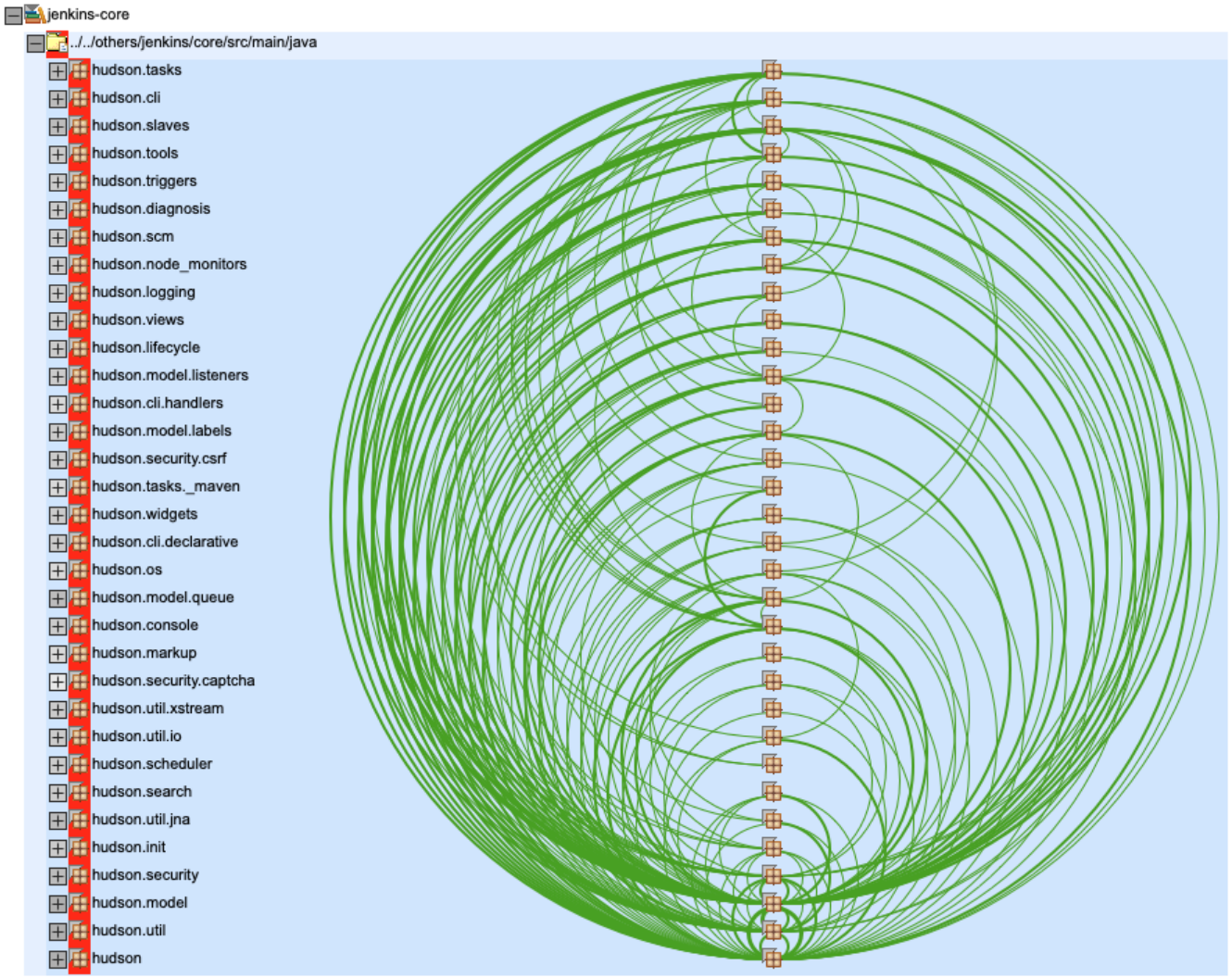


Figure 1.2. To Avoid: "Big Ball of Mud" Dependency Structure

As a consequence, we think that spending effort on a clean and consistent software architecture and controlling dependencies is essential to code quality as well as regularly cleaning up other code smells.

Our experiences match those of well-known experts. Here is an incomplete list of resources that we found affirmative to our thinking. They haven't lost any importance despite dating back a couple of years.

- 'Is High Quality Worth the Cost' by Martin Fowler, https://martinfowler.com/articles/is-quality-worth-cost.html, 2019
- 'Sustainable Software Architecture: Analyze and Reduce Technical Debt' by Dr Carola Lilienthal, dpunkt.verlag, 2020
- 'Is Design Dead' by Martin Fowler, https://martinfowler.com/articles/designDead.html, 2004
- 'Domain Driven Design' by Eric Evans, Addison-Wesley, 2004
- 'Large-Scale C++ Software Design' by John Lakos, Addison-Wesley, 1996
- 'The Pragmatic Programmer: From Journeyman to Master' by Andrew Hunt and David Thomas, Addison-Wesley, 1999
- 'Structured Design' by Edward Yourdon and Larry L. Constantine, Prentice-Hall, 1979
- 'Your Code as a Crime Scene' by Adam Tornhill, Pragmatic Programmers, 2015
- 'Applying UML And Patterns' by Craig Larman, Prentice Hall, 2000
- 'Refactoring to Patterns' by Joshua Kerievsky, Addison-Wesley, 2005
- 'Agile Software Development' by Robert C. Martin, Prentice Hall, 2003

Automating Checks

Manually tracking the evolution of the internal structure of a software system is not efficiently manageable. With thousands of classes and millions of dependencies this is impossible for any large system. You need a tool that automatically reports deviations in the implementation from your envisioned architecture. We think that the lack of proper tool support is the reason why so many projects suffer from software rot. There are many static analysis tools and linters that report errors and problems at source-file level, but most are missing out on the big picture: Tracking dependencies across source files and validating if the structure matches the envisioned design.

As a result, most systems contain a large number of cycle groups, elements are tightly coupled, no clear structure exists and if documentation exists, it is not up-to-date. New developers have a hard time to know how they should structure new functionality, where to place the new code and how it should interact with the existing code. Once you have lost control, the structural quality typically spirals downwards quickly and the software ends up in a "big ball of mud". Regular quality initiatives feel like Sisyphean tasks, because problems are created faster than they can be fixed.

Therefore, quality checks must be executed automatically by the Continuous Integration (CI) build, whenever new code is committed. Even better, quality checks should be executed while programming, so that those problems never get into the version control system.

Setting the Focus

Starting a new project with automated architecture checks in place is very practicable to maintain quality at a high level. It is way harder to sustainably improve quality for existing projects. Resources are scarce, and new features need to be implemented, so it is not feasible to simply stop all work and clean up everything first.

For any quality improvements, it is important to spend efforts where it supports the current work: This makes the positive effects visible, and the enthusiasm for code quality will stay and won't fade away quickly.

For this reason, Sonargraph offers to compare the current state of the system against a previously run analysis. This 'System Diff' identifies where quality was improved and worsened, making it ideal to support code and sprint reviews. Additionally, quality gates can be defined that ensure that quality trends in the right direction. A ranking algorithm highlights those issues where fixes provide high benefit, i.e. that are urgent and important: Issues that were added recently, that have a high impact on the system and where involved files have been changed recently.

The Sonargraph Product Family

Architects and developers are supported by Sonargraph to maintain and improve the quality of their software. Its focus is on architecture and dependencies, but it offers also a large number of metrics, duplicate code checks and additional rules that can be activated as needed, e.g. to detect unused code. Sonargraph is built upon the experiences that hello2morrow gained during the development and support of the predecessor products Sonargraph 7, SotoGraph and SotoArc. Sonargraph is lightweight and integrates smoothly with different IDEs, build and quality infrastructures (e.g. Eclipse, IntelliJ, SonarQube, Jenkins, Ant, Maven, ...).

Sonargraph consists of several products that help to ensure quality throughout the software development as shown in the following image:

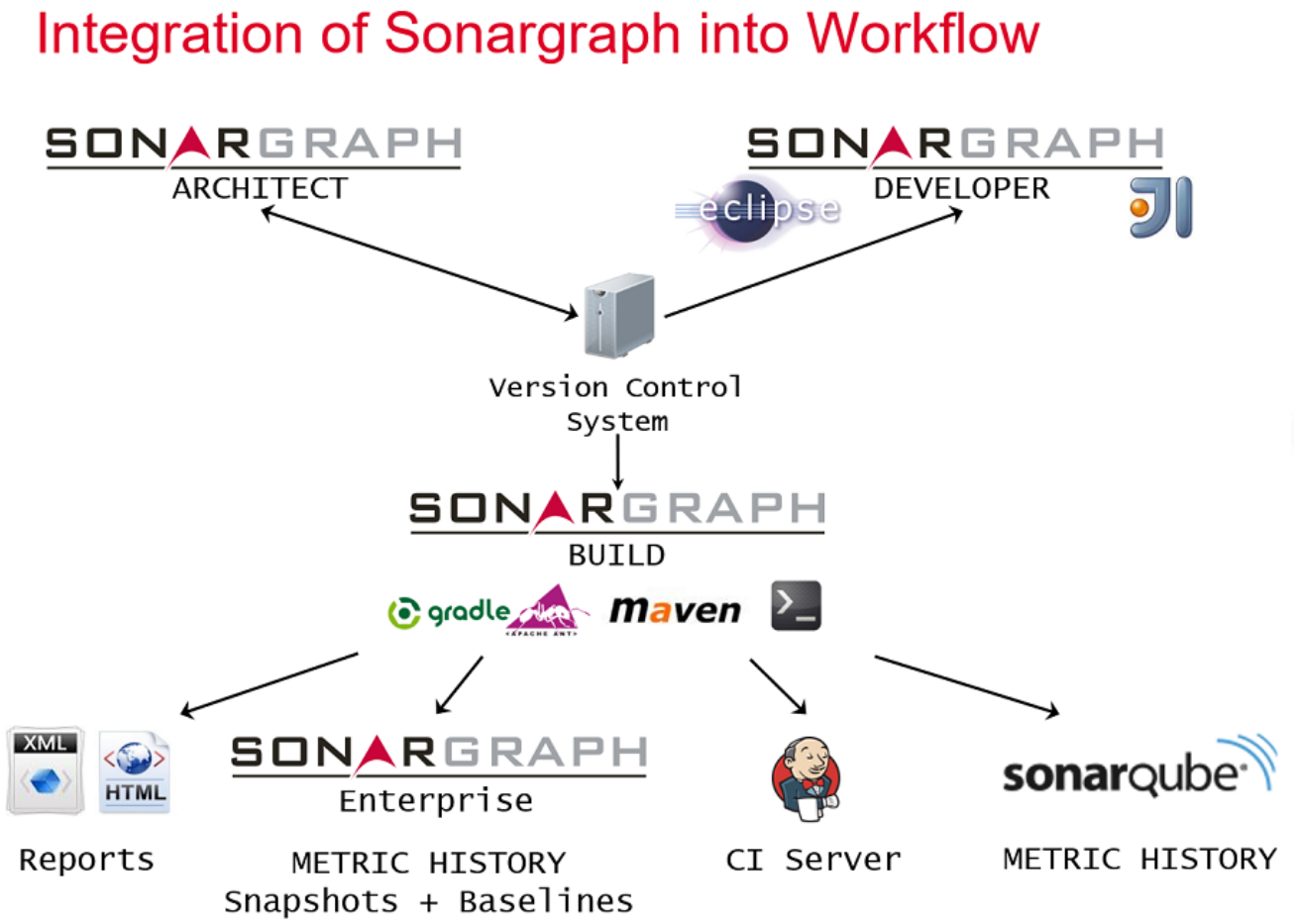


Figure 1.3. Sonargraph Products

- Sonargraph-Architect allows code exploration and definition of rules, i.e. architectures, metrics, anti-patterns, thresholds, tasks, refactorings. It offers additional analyzers. e.g. to detect code duplications and to provide custom metrics and issues.
- Sonargraph-Developer are integrations into IDEs that provide early feedback to developers. With a Developer license it is also possible to start the Sonargraph-Architect application and use its advanced visualization and exploration possibilities.
- Sonargraph-Enterprise is a web application that provides the history of metrics for multiple Sonargraph systems.
- Sonargraph-Build are integrations for various environments to run the quality checks on the continuous integration server.
- Further plugins exist that allow the integration of Sonargraph into SonarQube and Jenkins.

  We host an Open Source project on GitHub that provides easy access to all information contained in a Sonargraph XML report and can be used for custom post-processing: *https://github.com/sonargraph/sonargraph-integration-access*

Use Cases and Key Functionality

The following describes key functionality of Sonargraph and typical uses cases. This is just a summary, the rest of the user manual provides more details.

Architecture Definition

Sonargraph uses a Domain Specific Language (DSL) approach to describe the architecture. A system's architecture can consist of multiple architecture aspects which are checked in parallel. Alternatively, the architecture can be defined interactively. Architecture diagrams can be generated allowing to investigate connections between architecture artifacts.

Simulate Refactorings

Sonargraph allows the simulation of refactorings. For this, you can create multiple so-called virtual models. A virtual model is a space where the model from the parser(s) can be modified by refactorings and detected issues can be transformed into tasks or ignored (called resolutions). This allows the simulation of different approaches to change an existing structure. A virtual model can be based on another virtual model making it possible to reuse common refactorings and resolutions.

The 'cycle-breakup' analyzer proposes refactorings to find an efficient way to eliminate a cycle. It takes into account defined architectures and allows to interactively fine-tune the solution.

NOTE: A virtual model might affect metric values since the structure of the system can be changed with refactorings and issues can be transformed into tasks or ignored.

Hotspot Visualization

Sonargraph analyzes information from Source-Control Management (SCM), currently Git. The combination of issues and code changes and the visualization as software maps (a.ka. "Code Cities") allows the visual identification of hotspots.

Tracking Changes in Quality

If Sonargraph is used in existing projects there might be an overwhelming number of reported issues. The 'System Diff' analyzer allows focussing on changes, making it the ideal companion during reviews. Quality gates can be defined on the current system state or in comparison to a baseline, making it easy to follow the 'Boyscout Rule' and gradually improving the system's quality. The 'Issue Ranking' view recommends issues that are both urgent and important to fix.

Great Parser Model Detail, Little Memory Consumption

Dependencies are tracked down to method and field level offering more detailed exploration. Sonargraph has little memory consumption, as only the model coming from the different parsers is held in memory and all 'derived' structural elements (e.g. a layer) and their dependencies are calculated on demand.

Snapshots

The complete model of a system is stored in a compact binary format. This enables fast startup times (the last snapshot is used if available) without having to perform a full re-parse. Furthermore complete systems might be compared and historically analyzed - even passed around to enable reviews based on them - by directly loading the snapshot.

Fast Execution

Analyzers calculate metrics and analyze dependency structures (e.g. cycles) and content of source files (e.g. duplicated code). These analyzers run in parallel in a multi-threaded environment providing more speed while not blocking user interaction. Once an analyzer has finished, it`s results are available to the user.

Extensible Analysis

The user can extend Sonargraph's functionality by writing Groovy scripts accessing the model created by Sonargraph. These scripts can either simply act as custom queries finding artifacts with specific characteristics and/or to create issues pointing to potential problems in the system or create additional metrics.

Sonargraph also offers a plugin API to integrate external analyzers and to extend the parser model by custom elements. The currently existing plugins are 'Spotbugs' and 'PMD' for further file-local issues and 'Swagger' and 'Spring Microservices' to reveal web service dependencies.

Multiple Language Support

Sonargraph supports different languages depending only on the license without the need to have different installations. There is a unified approach (i.e. one user interface) to explore and monitor systems implemented in different languages. Systems have a module structure where each module can have a different language. A generic component approach is used for all supported languages - currently Java/Kotlin, C#, C/C++, Python.

Flexible Exploration of Dependency Structures

You are free to decide how to explore dependencies. Sonargraph offers a tree-like explorer, a graph viewer and a simple table-based viewer.

Automated Updates and Flexible User Interface

Automated updates and a flexible user interface (layout and customization) are provided as Sonargraph is built upon the Eclipse Rich Client Platform (RCP). Sonargraph-Build plugins for Maven and Gradle can also be configured to update automatically.

Exchangeable Quality Artifacts

The software system analysis comes with a multiple file approach. The software system is comprised of a main software system file, analyzer configurations, user defined scripts, different architecture aspects and so forth. The approach makes it easy to share valuable aspects of the analysis between software systems as well as to centralize common aspects in bigger companies.

Chapter 2. Getting Started

You are reading this, because you care about the quality of your code base and that's great!

Sonargraph identifies problematic areas and supports you to gradually improve your code base. Be aware that this is not an easy task, especially if no static code analysis checks have been executed for a long time on your project! It is very likely that there will be an overwhelming amount of issues that would take too long to be all fixed. But Sonargraph will steer you towards those issues where fixes provide the most benefit.

Don't Panic!

Not all issues will be easy to fix. Some, like huge cycle groups, might be really hard to solve.

Our advice is to treat "quality improvement" not as a short-term "sprint" but rather as necessary and integral part of software development that needs to be done continuously. The best you can do is to accept the current state of quality, look forward and gradually get rid of issues where code needs to be modified. If you cannot eliminate a big cycle group in one go, at least make sure that it does not get worse and free elements from it piece by piece.

The benefits will be great, because it not only improves the code base but these efforts will also make you a better programmer / architect, since you will be forced to think a lot about good solutions for the problems identified by Sonargraph.

"A Fool with a Tool is still a Fool"

This applies for Sonargraph, too. Programming and architecting skills need time, a lot of reading (see our recommendations at the end of the previous chapter) and deliberate practice.

Sonargraph is an excellent tool to tell you about the existing problems in your code. Finding good solutions is still your task!

This chapter is meant to be a quick reference on how to get started with Sonargraph. Links are provided to other chapters of the user manual, where you find more details.

Motivation and Key Concepts

In case you skipped Chapter 1, *Motivation for Code Quality*, we urge you to go back and read it. It provides convincing arguments about the usefulness of high-quality software, in case you need to convince someone else in your organization that these efforts are well spent. We also included a list of our favorite books that helped us write better software. As a next step, we recommend to get familiar with the key concepts used within Sonargraph by skimming Chapter 5, *Getting Familiar with the Sonargraph System Model*, so you know what we mean when we talk about "module", "root directory", "namespace", "component", "physical", "logical", "issue", etc.

Initial Configuration

Before you can analyze your code base, you need have a license. Check Chapter 3, *Licensing* for details on how to activate your license on Sonargraph.

If you want to analyse C/C++, C# or Python code, you probably need to configure some preferences, so that Sonargraph finds the code of the corresponding platforms on your machine. This is required to correctly resolve dependencies. Check chapters Section 4.7, “C/C++ Compiler Definitions”, Section 4.10, “C# Configuration”, and Section 4.11, “Python Configuration” respectively.

Help!

You will need some time to know your way around Sonargraph. We do our best to make the interactions as obvious as possible, but our intuition might differ from yours in some places. If you get stuck and don't know what to do next, simply press **F1** and some guidance will be provided in the context help with additional pointers to more detailed information.

In case that is not sufficient, please send us a feedback via the menu "Help" → "Send Feedback..." and we will get back to you as soon as possible for further support.

Setup Sonargraph System and Define the Scope of Analysis

Having resolved all the initial tasks, it is now time to import your code to Sonargraph. We implemented several importers that should let you create a Sonargraph system based on code developed with Eclipse, Intellij IDEA or Visual Studio. Check Chapter 6, *Creating a System* for details. In case the automatic import is not possible, a manual setup is also supported. When the import has finished, hit "refresh" and let Sonargraph analyze your code. Afterwards, the Workspace view lists all modules and directories where code has been found. In case you want to exclude code from the analysis or simply ignore issues in certain areas, configure the workspace filters accordingly as described in Section 8.8, “Managing the Workspace”.

In case your workspace in your IDE has changed and new modules/projects have been created, you can also create new modules via several different importers as described in Chapter 7, *Adding Content to a System*.

Initial Assessment

If you are like us, you will be keen on seeing some dependencies now. For this, you can either click the Exploration view quick access tool item in the main tool bar or select any number of elements (e.g. modules, files...) from the "Navigation view", open the context menu via right-click and select "Show in Exploration View". The view shows the dependencies as green arcs as they have been derived from the code.

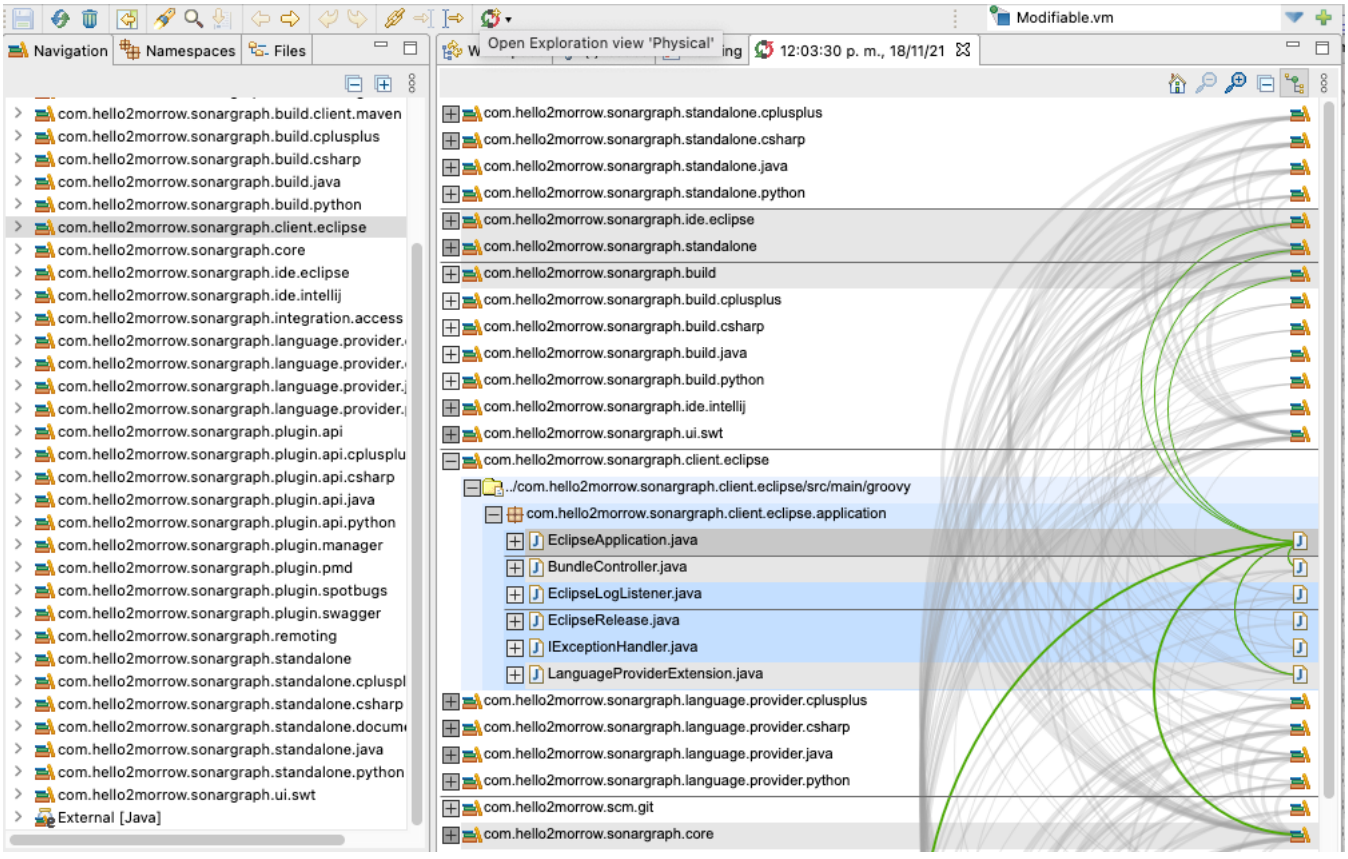


Figure 2.1. Exploration View

Selecting a node or dependency, the "Parser Dependencies View (Out)" view displays the details. We dedicated a whole chapter about how to explore the code and make the best use of the powerful functionalities: Section 8.11, “Exploring the System”

In case you started with static code analysis for an existing project, it is likely that the "Issues" view shows a huge list of problems. Of course, you can apply filters: Either by selecting elements in the tree view shown in the upper part or by selecting the issue types you are interested in. Section 9.2, “Examining Issues” describes all the possible interactions.

Not all issues should be treated equally. Some are more relevant to the future development than others and refactoring efforts should be focussed on them. We implemented an algorithm based on the "Eisenhower Method" to identify issues that are both important and urgent to fix. The suggested ranking can be examined in the "Ranking" view, the "Properties" view shows details of the individual parameters and how they contribute to the computed score. Check Section 9.2.1, “Identifying the Most Relevant Issues to Fix” for details.

Most likely cycle group issues will be among the most relevant issues. We have seen groups involving hundreds of elements, so their impact on the code base and the architecture is huge. Reducing the amount of code involved in cycles will have a very positive effect on the maintainability. How Sonargraph helps to investigate and to eliminate cycles is described in Section 8.10, “Analyzing Cycles” and Section 8.10.3, “Breaking Up Cycles”.

Duplicate code also has a negative impact, since it bloats the code base and makes bug fixing more difficult, because you need to know where the duplicates exist you need to repeat the fix at all occurrences. The inner workings and configuration options of the duplicate code analyzer are described in Section 8.13, “Detecting Duplicate Code”.

Of course, Sonargraph also computes a lot of metrics. The Metrics view as described in Section 8.15, “Examining Metrics Results” allows to search for outliers. This gets more convenient if you configure metric thresholds for those metrics that you find interesting. We prepared some thresholds for you, that can be imported as a quality model (check Section 6.4, “Quality Model” for details).

Define Meaningful Thresholds

Agreeing on "sensible" thresholds can be a matter of tough debates. Our advice is to not take them too serious. But, most of the times, you will find that the code is easier to understand after you applied a refactoring to eliminate the issue. In case it is not, you should talk to a colleague and maybe she will come up with a better refactoring proposal.

To identify hotspots, you can use treemaps as described in Section 9.2.2, “Identifying Issue Hotspots”. Simply looking at the code base from a different perspective can reveal suprising insights. Give it a try!

TIP

The best place for a Sonargraph system definition is next to the code base. If you haven't done it yet, share the system definition with the team and add it to your version control system. All information that makes up a Sonargraph system definition is contained in plain text files that are easy to read and to track their changes.

Define Architectures

One of the main ideas behind Sonargraph is to detect unwanted dependencies within the code base, so that the "big ball of mud" can be prevented. An architecture defines how parts of the system can reference each other. Sonargraph makes the architecture "actionable" by automatically verifying that the implementation matches the definition. Chapter 11, *Defining an Architecture* describes the rationale behind the implementation of the architecture as a Domain-Specific Language (DSL), and demonstrates the features using an example scenario.

The architecture DSL is tremendously powerful and allows to define complex structures with minimal effort. But it needs time to learn all constructs and how to combine them efficiently. That's why modeling the architecture interactively during system exploration was implemented in the "Architectural" view. It also allows defining refactorings while modeling, and it is fun to see how the system's structure is improving. Chapter 13, *Interactive Restructuring and Code Organization* describes details. (Note: No code is changed in this process, only tasks are created that need to be executed in your IDE.) The "Architectural" view is a sandbox. Once you are happy with the results, the architecture definition and tasks can be transferred and will then be actively checked.

TIP

The transfer creates a file containing the architecture defined with the DSL. If you have repetitive structures in your architecture, you should use DSL constructs to eliminate them, for example via "aspects" as described in Section 11.3, “Reusing Architecture Aspects”.

A lot of users like architecture representations as box-and-line diagrams. Since our architecture meta-model was derived from UML component diagrams, this is the implemented visualization that shows how the defined artifacts are interconnected. The hierarchical layout of elements follows the approach that is consistently implemented within Sonargraph: High-level elements with outgoing connections are above low-level elements with incoming connections. See Chapter 12, *Visualizing Architecture Aspects*.

Define Resolutions and Tasks

Knowing where the problems are is important. Sometimes, you decide you want to live with them, so Sonargraph lets create you "Ignore" definitions to move these problems out of focus. Sometimes, you find them important enough to be fixed, so you can define "Fix" definitions and recommend possible solutions. Using the Sonargraph IDE integrations for Eclipse or IntelliJ IDEA, those fix definitions will show up in the editor and help the developer to implement the solution.

Sonargraph also allows the simulation of refactorings: You can evaluate the effects of "move", "rename" and "delete" refactorings before they are implemented. The Sonargraph IDE integrations make it dead-easy to execute "move" and "rename" refactorings by delegating these to the IDE's builtin refactoring functionalities. The functionality of the integrations is described in Chapter 20, *IDE Integration*.

NOTE

Sonargraph task definitions should have a short life span. Otherwise there is the risk that tasks and the underlying code base get out of sync.

Continuously Check the Quality

The more frequent the quality is checked, the faster is the feedback about new problems and the easier it is to fix them. That's why SonargraphBuild can be integrated into your Continuous Integration (CI). It is up to you how to react on the results, either let the build fail or only send an email out to the developers. We have a dedicated user manual for this product that details all the configuration options: *https://eclipse.hello2morrow.com/doc/build/content/index.html*

Incremental Quality Improvements

Big-bang approaches rarely work. We propose that you accept the current state, move forward and ensure that the quality improves over time. Sonargraph lets you focus on changes and highlights added, worsened, improved and removed issues in the "System Diff" view. You create a baseline that the system's quality will be compared against, as described in Chapter 14, *Examining Changes*. We recommend to define goals that you want to achieve, e.g. during the next sprint or until the end of the next release. Sonargraph lets you define those as quality gates for metrics and issues (see Chapter 15, *Defining Quality Gates*) and checks them automatically.

Once you have completed the above steps and got familiar with Sonargraph's features, it's time for a recap. You should think about when and how you want to use Sonargraph to check that you are still on track. Baselines can support reviews for features and releases and ensure that no additional problems are introduced.

Apart from ensuring that no new issues are introduced, we recommend to regularly look at the ranked issues and select some of them to be fixed. It is also worth to check for hotspots using treemaps, so that areas for larger refactorings can be identified.

Using Special Checks

You can extend Sonargraph by writing additional checks and compute futher metrics via Groovy scripts. A number of predefined scripts exist that can be imported from the built-in quality models. They check for "dead code", compute metrics like "Depth

of Inheritance", identify code that has the most impact on coupling ("ACD Top Scorer") and detect patterns like "Singleton" to name a few examples. Chapter 16, *Extending the Static Analysis* provides more details.

Stay Up-To-Date

No software is perfect and Sonargraph is no exception. We heavily use assertions to check for internal consistency. Sonargraph will let you know if one of them fails and we kindly ask you to send us the error report. Bug fixes have high priority for us and we frequently release updates that the application will offer you to install at startup.

We regularly publish blog articles at *https://blog.hello2morrow.com/* to illustrate the benefits that you get by using certain features. Our web site also contains a number of videos that show Sonargraph in action (*https://www.hello2morrow.com/videos*).

In case you have ideas for additional functionality, please send them to us via "Help" → "Send Feedback..." and we will be happy to integrate them in our backlog.

Chapter 3. Licensing

When you start *Sonargraph* you will be asked for an activation code or a license file. For additional licensing and pricing information please contact <sales@hello2morrow.com> or <support@hello2morrow.com> and check our *web site* .

3.1. Getting an Activation Code or a License

When you have purchased a *Sonargraph* license, an activation code or a license file will be delivered to you.

There might be a program for free *Sonargraph* licenses which are time-limited and/or size-limited. Please register on our website and check the available programs.

In order to replace a valid license by a new one, choose "Help" → "Manage License..." from the user menu in the GUI-based product. *Sonargraph* licenses are bound to a named user. The usage by a different user is a violation of the license agreement.

3.2. Activation Code Based Licensing

Activation code based licensing activates *Sonargraph* licenses via Internet or a local license server by requesting a so-called ticket. Every activation code is customer specific and represents a pool of *Sonargraph* user licenses as purchased and licensed to the specific customer. Activation code based licensing technically requires that *Sonargraph* has Internet access or that a local license server is reachable. There are two types of activation code based licenses available:

1. Flexible User License (if you bought Sonargraph before version 9.0 you have flexible user licenses)

2. Floating License (new with Sonargraph 9.0)

Flexible user licenses support a feature that allows customer-driven transfer of a *Sonargraph* user license to another user after some amount of time. This works like this:

- When an activation code based license is requested, *Sonargraph* automatically requests a license ticket from the hello2morrow license server. This ticket expires after some time, for example after 30 days. During these 30 days, the use of the *Sonargraph* installation that requested the ticket is licensed (by the user who ran *Sonargraph* when the license ticket was requested). *Sonargraph* can be used during this period without any access to the Internet.

- After the ticket of a *Sonargraph* installation has expired (in our example scenario, this happens on the 31st day after the ticket has been requested), one of two things typically happen:

  1. The same *Sonargraph* installation is started again. *Sonargraph* then notices that the license ticket has expired and lets the user know about it by presenting a dialog to manually request a new ticket from the hello2morrow license server, for the same activation code or a different one if desired. The new ticket again is valid for the same time period. You can toggle the feature at ' Help → Renew License Ticket Automatically ' to have *Sonargraph* silently perform license ticket requests using the current activation code, without further user interaction.

  2. Alternatively, the user of the installation might not continue to work with *Sonargraph*; then the license is now, after the expiration of the ticket in the *Sonargraph* installation, available to some other user. The hello2morrow license server will supply a license ticket to the next user that requests one for the given activation code.

Note that the number of license tickets that can be supplied by the license server for some activation code might be more than one. For example, a company might license *Sonargraph* for 20 users. The same activation code can be used by all of them, but as soon as the 21st license ticket is requested for this activation code, this request will be denied. A new request for a ticket will only be fulfilled after one of the already supplied tickets has expired, so that at any one moment, at most 20 non-expired license tickets exist for the activation code.

It is not required that the same user requests a replacement of an expired license ticket; any user that knows the activation code can request one of the free tickets. This mechanism reduces the effort needed for license management in a changing user group.

However, in order to avoid any misuse we strongly encourage you to restrict the information about your activation code to those persons who are supposed to use *Sonargraph*.

If you have any suspicion about misuse please inform <support@hello2morrow.com> immediately. We can promptly deactivate an activation code so that any further misuse is stopped and provide a new activation code to you.

Floating licenses bind a ticket to an instance of Sonargraph while it is running. As soon as Sonargraph is terminated the license can be used by another user.

Most of our customers are using our Internet based license server, so there is no need for you to operate your own license server as long as the machines running Sonargraph have access to the Internet. If this is not the case or you want to avoid being dependent on the availability of hello2morrow's web-based license server you can request the usage of a local license server by contacting us via <sales@hello2morrow.com> or <support@hello2morrow.com>. Once your request has been approved, you can download hello2morrow's local license server and run it on your premises. If you have a *flexible user license* it is also possible to run Sonargraph with file based licenses.

3.3. Proxy Settings

If you use hello2morrow's Internet servers and Activation code based licensing, you need Internet access. If your network configuration does not allow direct Internet access, but provides access through an HTTP proxy instead, you can specify the host name and port of the proxy server. If the proxy server access is password protected, you can supply a user name and a password in order to authenticate.

For the GUI-based product, the proxy settings can be changed via "Preferences..." → "Proxy Settings" .

Check the user manual of Sonargraph-Build for proxy configuration options of the build server integrations.

3.4. License Server Settings

I you use your own license server you need to configure the access to it. You must specify the host name and port of the license server.

For the GUI-based product, the proxy settings can be changed via "Preferences..." → "License Server Settings" .

Chapter 4. Initial Configuration

This chapter summarizes what is needed for *Sonargraph* to run, how the update mechanism works and the necessary configuration before you can start creating *software systems* .

Related topics:

- Chapter 3, *Licensing*
- Appendix B, *Tutorial - Java*

4.1. Installation and Updates

Sonargraph is built upon the Eclipse Rich Client Platform (RCP) framework. The following prerequisites must be fulfilled:

- Microsoft™ Windows™ , Mac OS-X or Linux® operating system.
- 2048 MB RAM (Win32: 1400 MB)

Sonargraph leverages the advantages of the Eclipse Rich Client Platform update mechanism, thus, it will automatically connect to the hello2morrow update site and check for new versions at startup.

On Windows, Sonargraph stores application specific data (e.g. state files for the undo/redo history) in the directory %APPDATA%\hello2morrow\Sonargraph. If you notice slow performance during edit operations and you cannot exclude this directory from the virus scanner, create a script that reconfigures the environment variable "APPDATA" and then starts Sonargraph.

4.2. Help

The documentation for *Sonargraph* (i.e. this document) is also integrated into the product and available via the main menu entry "Help" → "Help Contents..." or by pressing the **Ctrl+F1** shortcut. It also provides a search functionality.

Dynamic / context-sensitive help is available within the application via the shortcut **F1** .

If there is no answer to your question available, contact us via the built-in feedback functionality, which can be found at "Help" → "Send Feedback..." or by sending an email to `<support@hello2morrow.com>` .

4.3. Editor Preferences

For architecture files and scripts you can set editor preferences. In the "Preferences..." menu, you find the possibility to change the editor preferences:

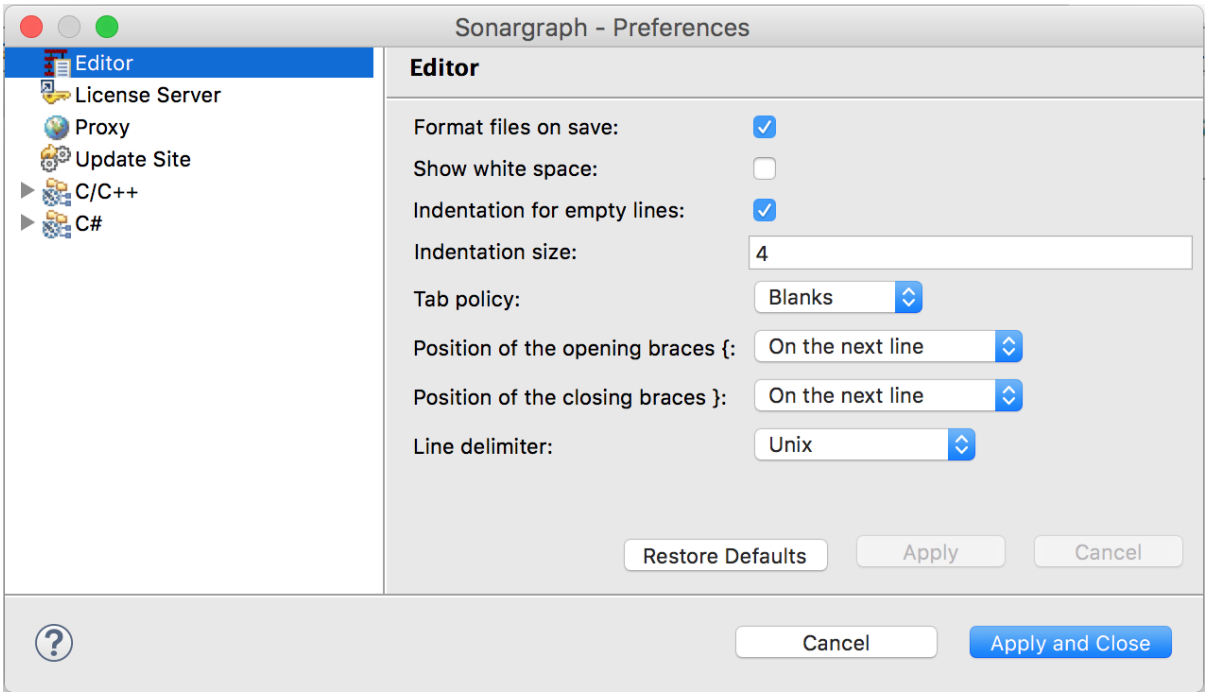


Figure 4.1. Editor Preferences

- **Format files on save** If set architecture and script files are formatted when saved, otherwise not.
- **Show white space** If set white space characters are shown with special characters, otherwise not.
- **Indentation for empty lines** If set empty lines will be automatically indented while being formatted, otherwise empty lines will stay empty.
- **Indentation size** Set the indentation size (only relevant for tab policy "Blanks").
- **Tab policy** Choose between "Blanks" to use blanks for indentation, and "Tabs" to use tabs for indentation.
- **Position of opening braces** Choose between "On the same line" to put opening braces on the same line, and "On the next line" to put opening braces to the next line.
- **Position of closing braces** Choose between "On the same line" to put closing braces on the same line, and "On the next line" to put closing braces to the next line.
- **Line delimiter** Choose between "Windows" which will end lines with CR and LF, and "Unix" which will end lines with a LF.

4.4. License Server Preferences

In the "Preferences..." menu, you find the possibility to change the license server preferences:

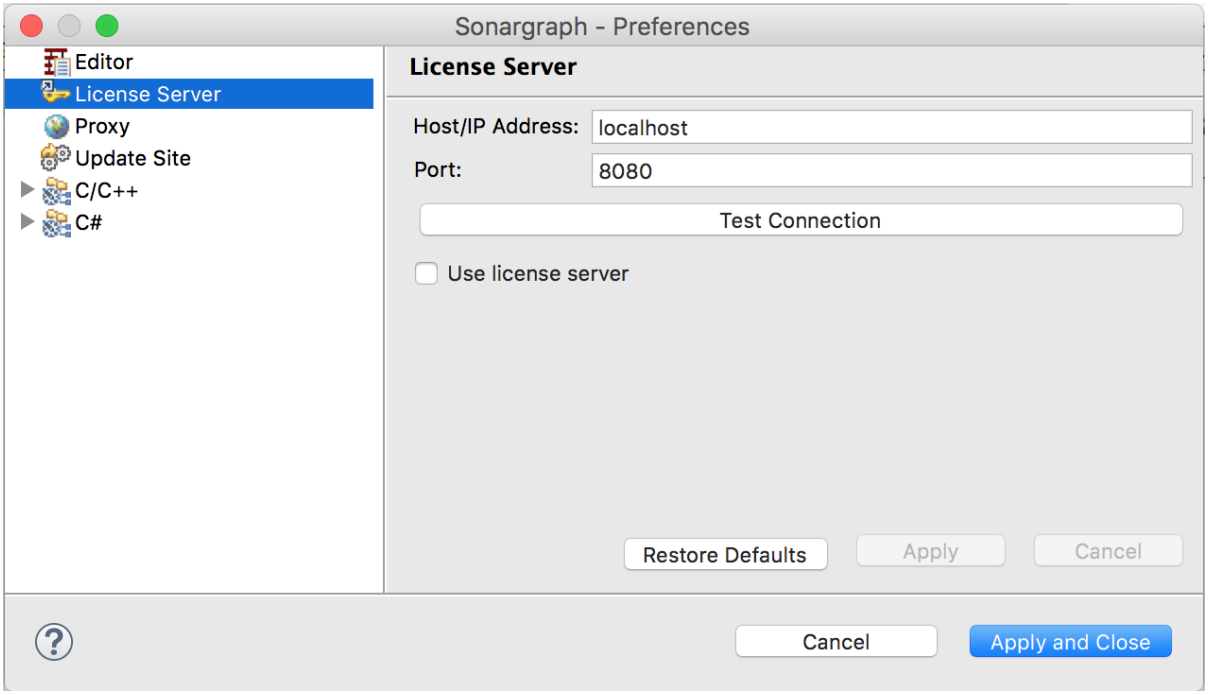


Figure 4.2. License Server Preferences

4.5. Proxy Preferences

In the "Preferences..." menu, you find the possibility to change the proxy preferences:

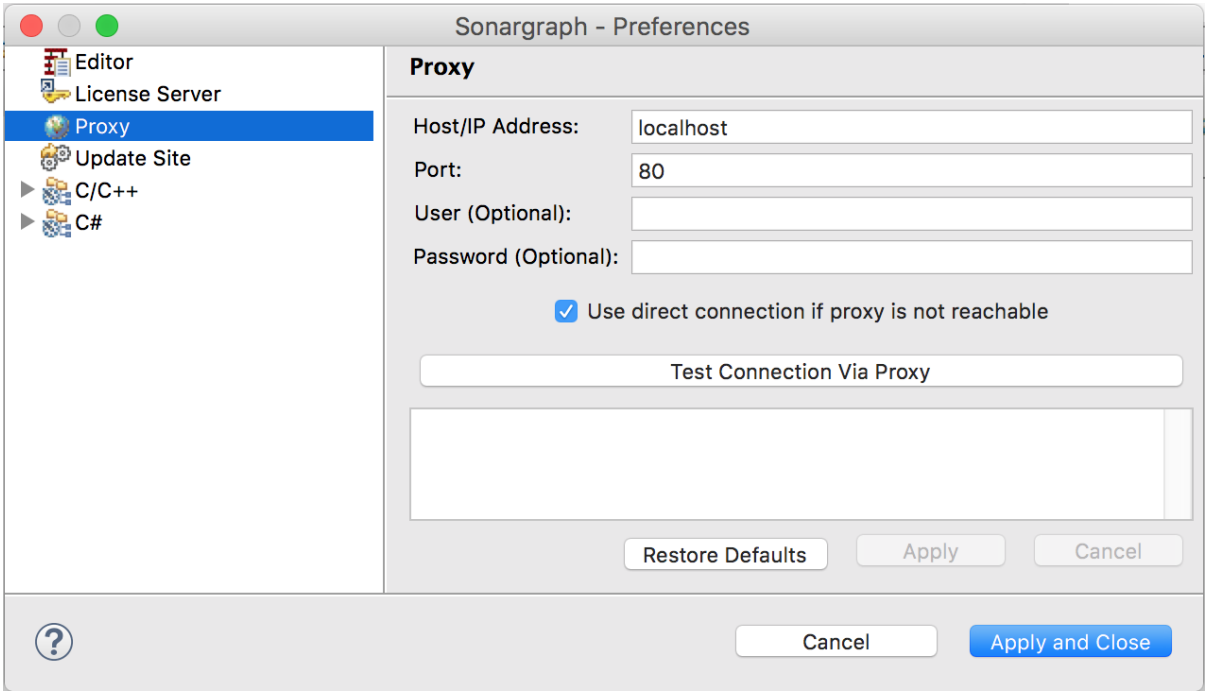


Figure 4.3. Proxy Preferences

4.6. Update Site Preferences

In the "Preferences..." menu, you find the possibility to change the update site preferences:

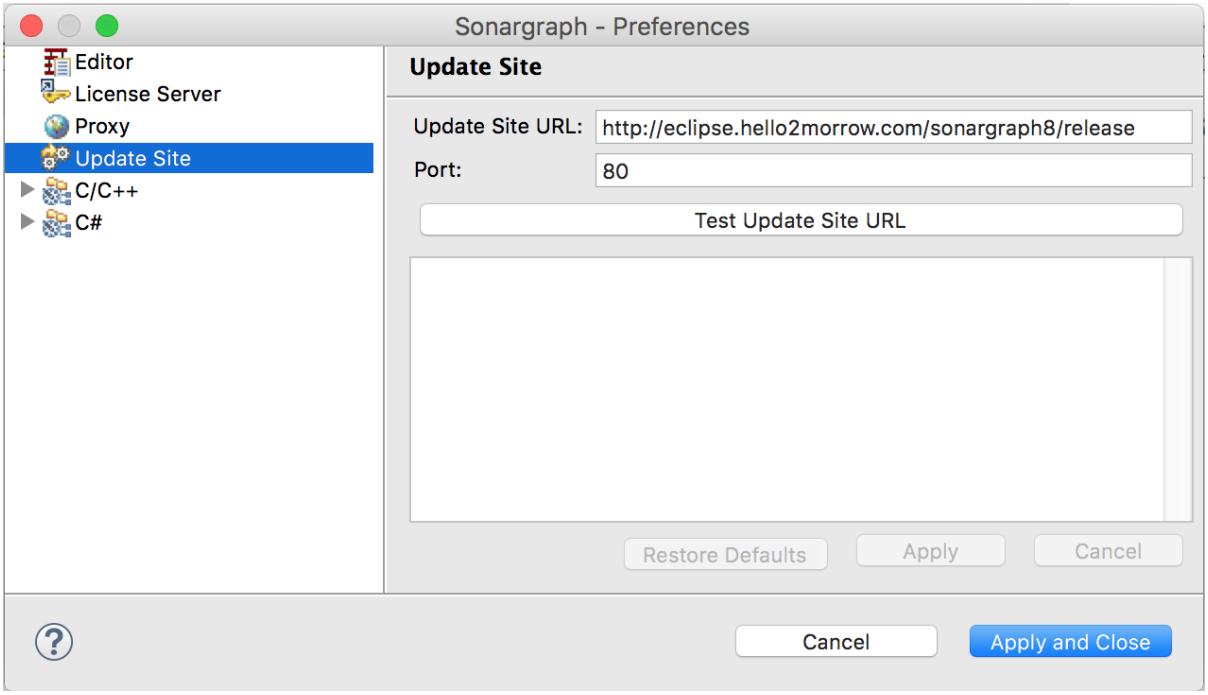


Figure 4.4. Update Site Preferences

- **Update Site URL** Use this update site to check for new releases of Sonargraph Standalone. Change this if you want to operate a local mirror of the official hello2morrow Sonargraph update site.

  Port The port number of the update site.

If a proxy is configured in Section 4.5, “Proxy Preferences” it will be used while connecting to the update site.

4.7. C/C++ Compiler Definitions

Sonargraph uses internally the *Edison Design Group (EDG) C/C++ Front End* to parse C/C++ sources. In order to emulate the behavior of your C/C++ compiler, Sonargraph needs a compiler definition. A compiler definition contains the location of the directories containing the system include files, a list of predefined macros and other options for the EDG parser defining language features and compatibility levels. You will not be able to successfully parse a software system without a proper compiler definition for your compiler. One compiler definition has to be set as the "active" definition, which will be used by default for opened *software systems* containing C/C++ modules.

Sonargraph comes with pre-defined compiler definitions that are activated by default depending on the platform *Sonargraph* is running on:

- "CLang" for Mac OS-X.
- "GnuCpp" for GNU C++ compiler on Unix based systems (Linux, Unix).
- "VisualCpp_x_y_z" for Windows based systems that have Microsoft Visual Studio Compiler installed. (x = version, y=architecture, z=processor, e.g. VisualCpp_12.0_x86_amd64). These definitions will not be automatically generated anymore because from Visual Studio 2019 on it is not possible anymore to query the registry for the installation location of Visual Studio. You have to tell Sonargraph where Visual Studio is installed. You can even register different Visual Studio versions with Sonargraph. To register an installation use the "Visual Studio Installations" preference page under the C/C++ preference

page group. To get there just select "Preferences" from the "Windows" menu. You then add the root directory of each Visual Studio installation you would like to use with Sonargraph. The root directory must have a sub-directory "VC".

If you are using a different compiler the easiest way to create a new compiler definition is to use the wizard under the "File/New/Configuration..." menu. If you have used our old product Sotograph before the wizard offers you to import a Sotograph compiler definition into Sonargraph. If you do not have a Sotograph compiler definition file you can ignore this step.

In the "Preferences..." menu, you can manage and modify existing compiler definitions or create new ones based on existing compiler definitions.

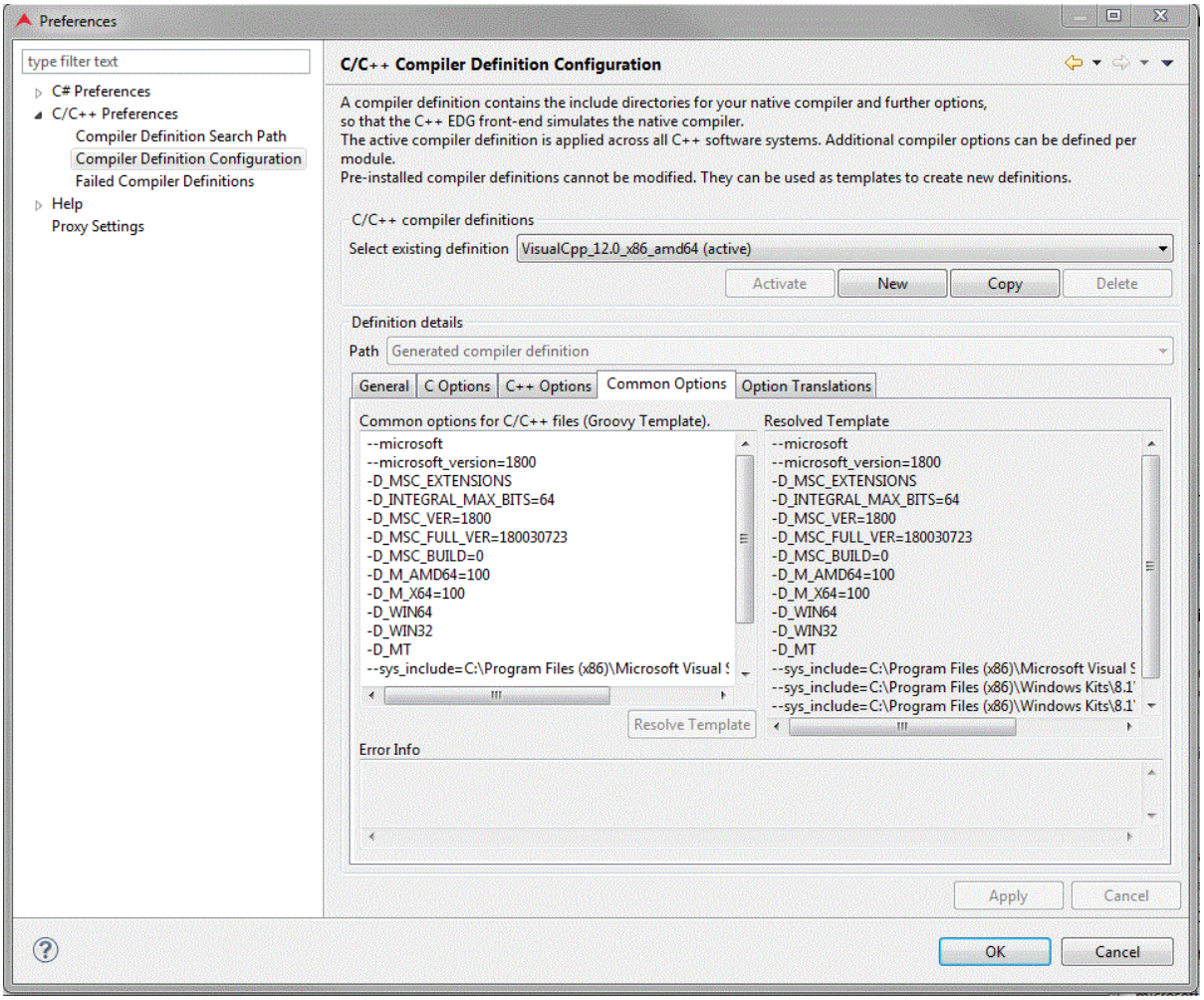


Figure 4.5. C++ Compiler Definition

The translation tab allows to define how options retrieved from imports need to be handled: For C++ modules created based on imports (e.g Makefile or Visual Studio 2010 project files (.vcxproj)), only macro (-D) and include (-I) preprocessor options will be applied. Use the translation functionality if any additional options of the imported project are required for parsing or the EDG parser uses a different value than your standard compiler.

For certain compilers it is possible to dynamically retrieve predefined macros and the include search path. To do that compiler definitions can be based on Groovy templates that invoke the compiler to query those settings. This is of course not possible for all compilers. Therefore we also have created a compiler definition wizard that will collect the information about the compiler to be emulated from you. You can invoke this wizard from the "File" → "New" → "Configuration..." menu. The wizard also supports the import of compiler definitions from Sotograph. (Previous tool from hello2morrow)

NOTE

You need to "activate" a compiler definition to use it for parsing. Just selecting a definition is not enough.

NOTE

Replacing the active compiler definition or modifying its content will force a reparse of the currently loaded *software system* as soon as the compiler definition is activated or the changes are applied.

By default, compiler definitions are stored in the *Sonargraph* home directory. These definitions are not intended to be shared. If you want to share compiler definitions across team members, it is recommended to specify a separate directory in the search path that contains these shared definitions. See Section 4.8, “Search Path Configuration” .

4.8. Search Path Configuration

Similar to a Java classpath, C++ compiler definitions are looked-up using search paths. The search paths contain at least one entry, which is per default located within the *Sonargraph* user-home directory. Further directories can be added to the search path that allow to share configurations between users, i.e. if those directories belong to a network drive. Those directories are searched if the configuration file is not found in the installation-specific directory.

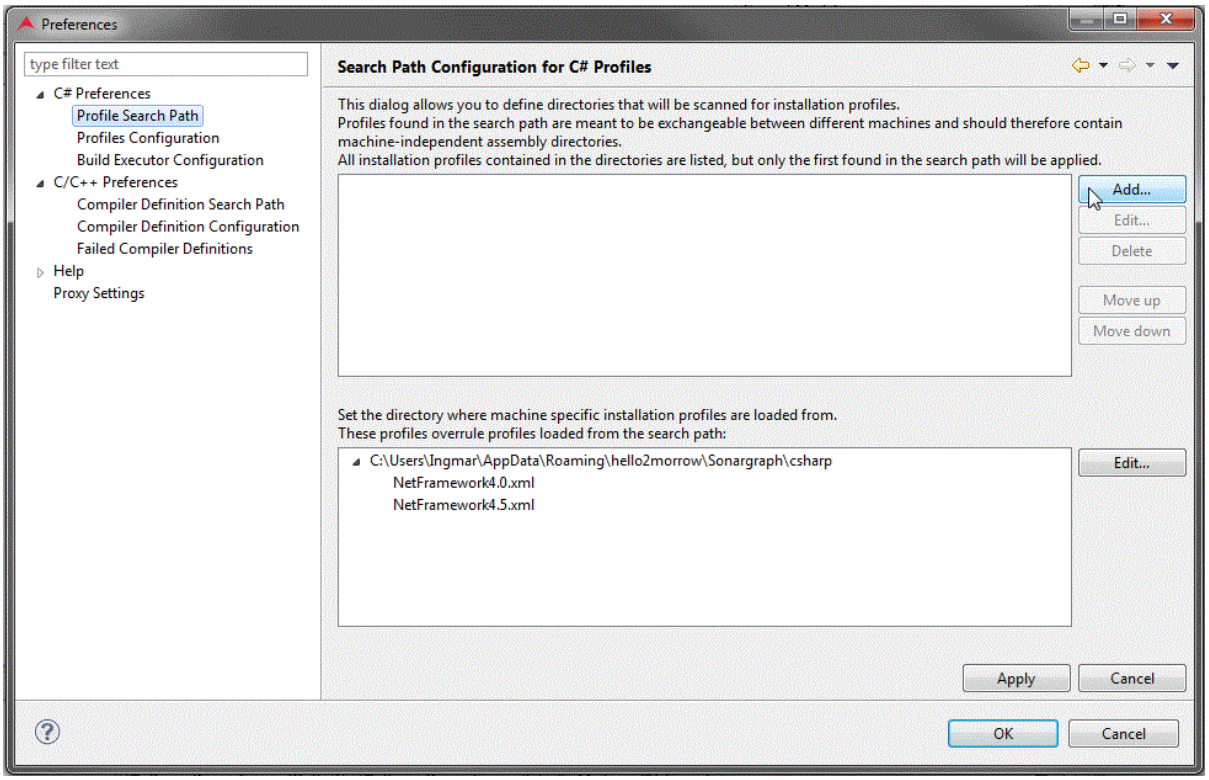


Figure 4.6. Search Path Configuration

4.9. C/C++ Parser Daemon Configurations

Sonargraph uses background daemon processes to speed up the parsing process. The daemons only run during refresh and need between 250 MB and 350 MB of memory. Per default the number of daemons is configured to 8. You can change that via setting in the C/C++ preferences to a value between 1 and 16. You should lower the default when your machine is short of memory. You can use a higher value than 8 if you have a high-end CPU and lots of memory. You can also increase the daemon stack size from the default of 1MB. This might be necessary if you have crashing parser daemons. Maximum stack size is 128 MB, which would significantly increase the memory needed per daemon.

Please note that fewer daemons are started if the number of files to parse is smaller than the configured number of daemons.

4.10. C# Configuration

Our C# parser is based on the open source Roslyn project, which is also the basis for Microsoft's official C# compiler. So we are using your local .Net configuration. You only need to select the modules you want to analyze and make sure that the solution can be built locally using your favored IDE.

4.11. Python Configuration

Sonargraph supports Python version 3 and higher. To enable the support Sonargraph must know the location of the executable for the Python interpreter. You can configure that in the "Python Preferences" "section of the Sonargraph preferences dialog. We also assume that you would use virtual environments for managing project specific dependencies. In that case you should configure the Python interpreter of your virtual environment in the setting dialog brought up by "System/Configure". In any case Sonargraph will ensure that your interpreter supports at least Python 3.

Since Python is a dynamic language many dependencies will not be detectable by Sonargraph - everything is an object and typing information is rarely available. Nevertheless the model will still contain the most relevant dependencies (e.g. object creation, inheritance, function calls, member access etc.) so that the result is good enough to analyze dependencies and enforce architectural constraints. Please be sure to read the section about Sonargraph's Python model in the next chapter.

To analyze a Python system with Sonargraph you must execute the following steps:

- Create a new software system by using "File / New / New System..."
- Add a Python module by selecting "File / New / Module / New Python Module...". Usually Python systems only contain a single module.
- Add the root directory for your Python project by right clicking on the module you created in the previous step and select "New Root Directory...". If you have more than one source root directory you can add several.
- If you project uses a virtual environment please configure the Python 3 interpreter of this virtual environment via the "System / Configure..." dialog.
- Save your newly created system.
- Start the parser by clicking on the "refresh" icon (top left icon in the tool bar). The first parser run will always take longer since we have to parse all the directly and indirectly imported files from the Python library.
- Now you should have a model and you can browse dependencies, metrics and anything else that is contained in the model.

Chapter 5. Getting Familiar with the Sonargraph System Model

The *software system* is the scope of analysis in *Sonargraph* . This chapter describes the model used by *Sonargraph* to represent a *software system* based on your code components and elements in order to fulfill different goals regarding the analysis.

5.1. Physical File Structure

The *Sonargraph software system* is physically represented in the file system by a directory <System-name>.sonargraph that contains a file named *system.sonargraph* :

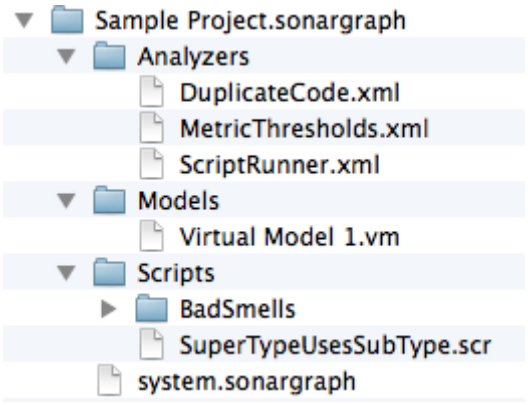


Figure 5.1. Physical File Structure

- system.sonargraph contains all information necessary to parse the code, i.e. the workspace information about modules, directories, etc. See Section 8.8, “Managing the Workspace” and Chapter 6, *Creating a System* .
- Analyzers sub-directory contains configuration for code duplication, metric thresholds and which of the Groovy scripts are executed automatically.
- Models sub-directory contains the *virtual model* files, i.e. the information about resolutions (todo, ignore, fix) for detected issues.
- Scripts sub-directory contains the Groovy scripts that allow custom queries.

Analyzer files and scripts are part of the Sonargraph quality model. See Section 6.4, “Quality Model”

5.2. Language Independent Model

The language independent domain model of the system is depicted in the following diagram. Domain models for specific languages are detailed in subsequent sections. Referenced types that cannot be located in the workspace are put under the "External" node. External elements are not part of the metrics calculations.

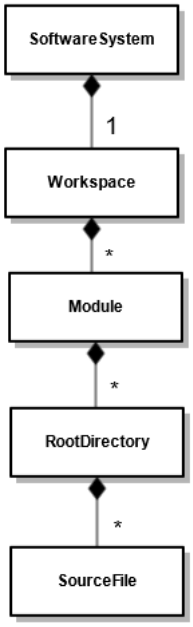


Figure 5.2. System Domain Model

5.3. Language Specific Models

The language specific models are built around the central idea of a *component* as defined by John Lakos in "Large Scale C++ Software Design": “A component is the smallest unit of physical design.”

They represent specializations of the language independent model elements. Those specializations depend, of course, on the elements of the language.

5.3.1. Java/Kotlin Model

Sonargraph parses the Java/Kotlin byte code (i.e. the .class files) for the static analysis. For a basic analysis, it is sufficient to specify the directories where the compiled byte code can be found. For a more advanced analysis like the detection of duplicate code blocks and the direct navigation to references in the source code, the source root directories are required (recommended). If the source file is available for a found type (class, interface, ...) the compilation unit is created underneath the corresponding source root directory. If no source can be found the compilation unit is created under the corresponding directory where the byte code was found. The following diagram shows the domain model for Java.

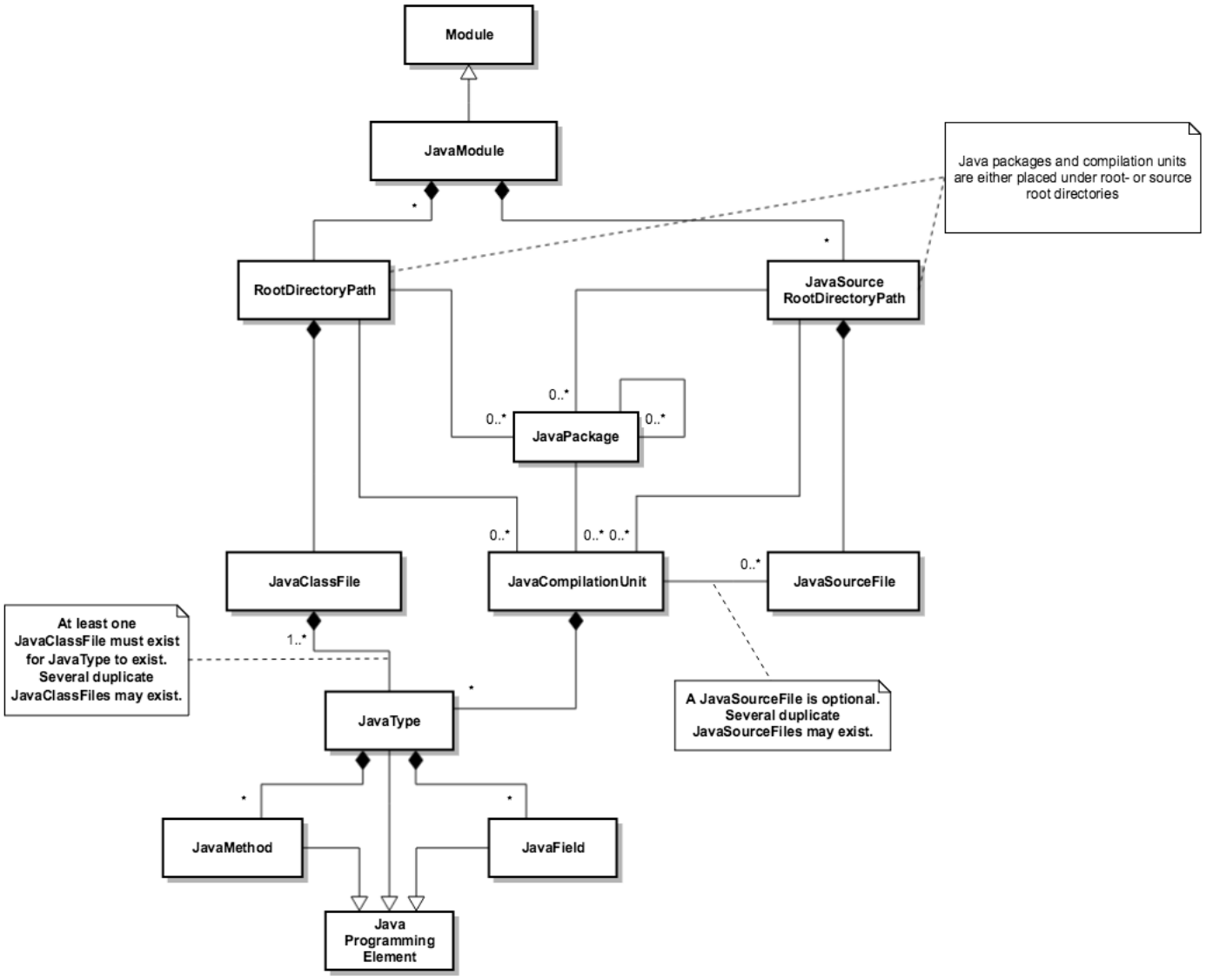


Figure 5.3. Java Domain Model

For inner classes and anonymous inner classes the correct nesting of Java compilation units is applied to types and methods respectively. This is not shown in the diagram for simplicity reasons.

All classes found in the byte code of the specified workspace are part of the system. Classes that are referenced by these classes but cannot be found in the given root directories are not part of the workspace and appear in the "External" node.

5.3.2. Kotlin Specific Issues

We added support for the JVM version of Kotlin to Sonargraph. There are, however, some issues with the Kotlin support that cannot be solved easily due to the way Sonargraph analyzes the code. The biggest issue comes from inline functions and methods. Since Sonargraph is relying mostly on byte code to analyze dependencies you will not see the dependencies at the location where the code is inlined. In most cases this is not really a serious problem, but you should be aware of this problem. The easiest way to avoid the problem is limit using inline functions in your code. Most of the time the potential performance gain can be neglected anyway.

5.3.3. C++ Model

Sonargraph uses the Edison Design Group (EDG) C++ Front End for parsing C/C++ code. The EDG parser must be configured appropriately in order to simulate your native C++ compiler. The basic domain model for C++ is shown in the figure below.

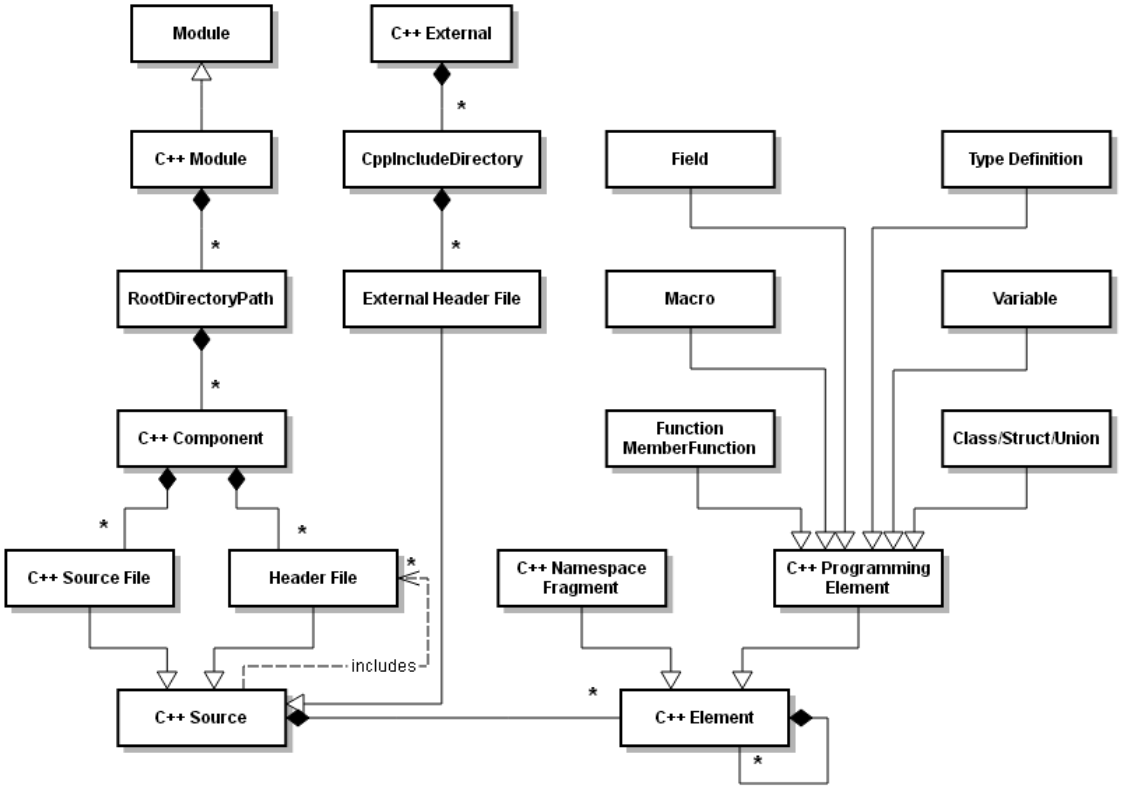


Figure 5.4. C++ Domain Model

An important difference to the model of other languages is the fact that C and C++ are using header files to declare items and source files to implement them. Associated header and source files form a logical unit that is called a component in Sonargraph. In other languages like Java components are always represented by single source files. Sonargraph is able to determine the components automatically by looking for declares relationships. If a function is declared in header "function.h" and implemented in a source file "function.cpp" Sonargraph will automatically combine the two into a component called "function". The component is anchored in the directory of the source file.

It is possible for a component to have more than one header file, if the elements implemented in a source file are declared in more than one header file. It is also possible for a component to have more than one source file, if the elements declared in a header file are implemented in more than one source file. It is nevertheless good practice to avoid those situations.

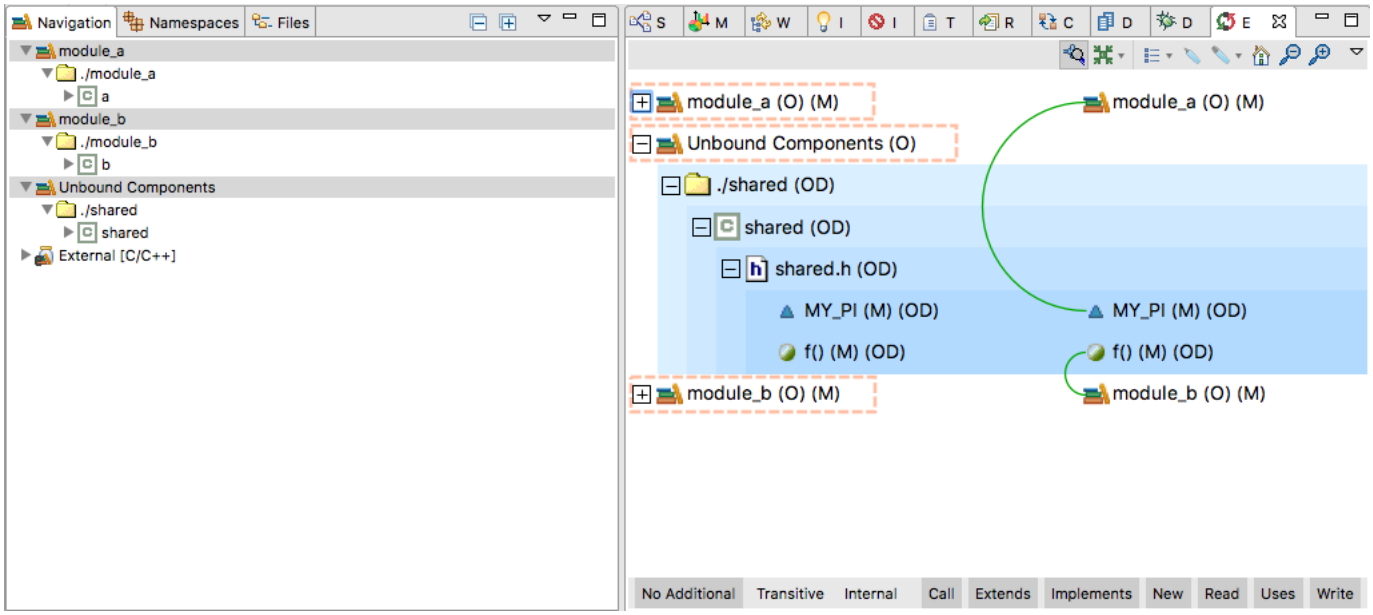
Sometimes it can happen that the automatic creation of components creates overly large components containing unrelated header and source files. That is usually caused by cross declaration, e.g. a global variable is declared in several unrelated header files. If you come over a component that contains unrelated source files you can always analyze the situation by opening the "Component Construction View". To open this view right click on a component in the navigation view and select this view from the context menu. The view will show a graphical representation of all the declares relationships within a component. Using that view it should be easy to find the rogue declarations that cause unrelated files to end up in a single component. You fix the problem by removing the rogue declarations from their header files. Instead you should include the correct header file before using the declared entity.

Sonargraph will attach a warning issue to components that contain more than one header file so that you can easily find components that might be containing unrelated source files. If after inspection you come to the conclusion that the source files

in a component are properly related you can ignore the corresponding issue in the issues view (by right clicking on the issue and selecting "Ignore" from the context menu). Ignoring the issue will hide it from the issues view and also will suppress the warning marker that was attached to the component.

Sometimes it is also possible that a component only contains a single header file, e.g. when a class has only inline members. In that case there are situations when it will be impossible for Sonargraph to determine where to anchor such components. To solve this problem Sonargraph will create an artificial module called "Unbound Components" and anchor the component there. The user can then right click on such components and select "Assign to module..." from the context menu. After saving the current system state that decision will be persisted. As soon as the last unbound component has been assigned to a module the artificial module will disappear.

In the example below for example the component "shared" could belong to "module_a" or to "module_b". Only the user is able to resolve that.



5.3.4. C# Model

Sonargraph parses the C# source files and relies on the existence of all referenced assemblies. Sonargraph offers C# profiles to specify the directories where assemblies are located. Types found in these referenced assemblies are put under the "External" node.

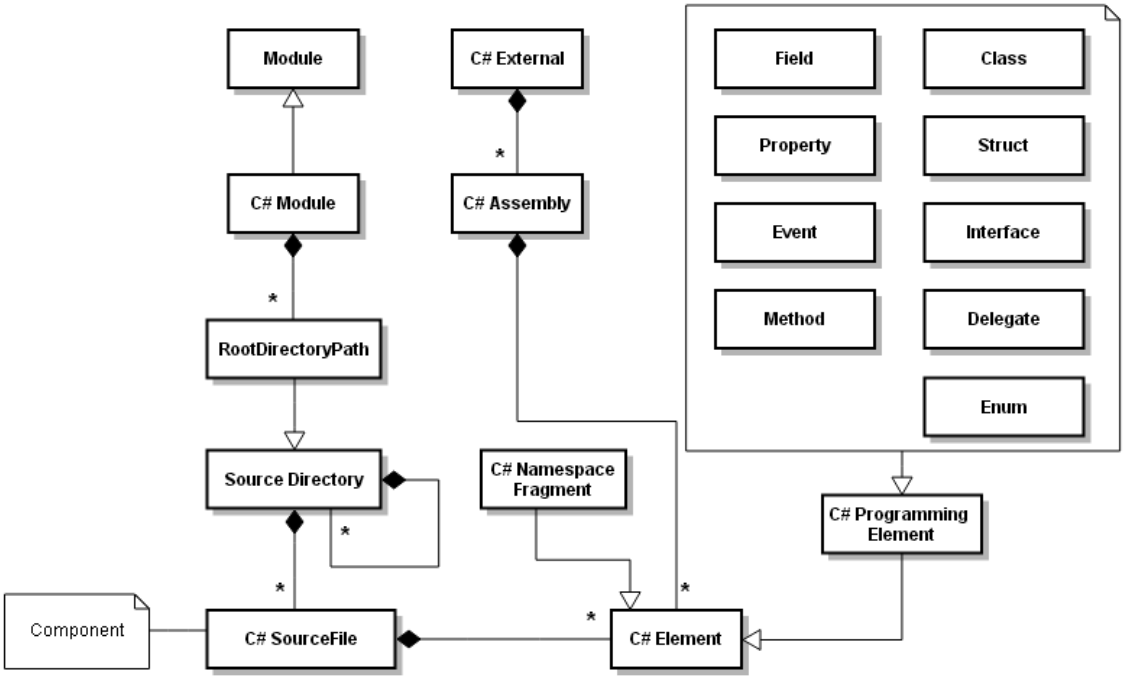


Figure 5.5. C# Domain Model

5.3.5. Python Model

Sonargraph parses the Python source files of your project and all directly and indirectly included files from the Python library and other third party libraries used by your software system. Since namespaces in Python work quite differently compared to the other languages supported by Sonargraph we decided that Python modules are considered as namespaces/packages in the logical model. So in the namespace view your Python modules (i.e. source files) will show up as packages.

For the cycle analyzer that means that Python modules play a double role as "components" and as packages at the same time. So package cycles can actually contain single Python modules.

When it comes to analyzing dependencies with respect to calling a method of a class that can only be resolved if the class of the receiver is known at compile time, which usually is only true for calls on "self" and if type hints are available.

5.4. Logical Models

Besides the model that comes from each language-specific parsing process, *Sonargraph* offers two more models that contain system-based and module-based logical elements which are calculated based on the physical model. These elements are basically *logical namespaces* and logical programming elements and their calculation is explained with more detail below.

Logical Namespaces

To better understand the concept of Logical Namespaces, it is necessary first to take a look at a couple of examples of physical namespaces:

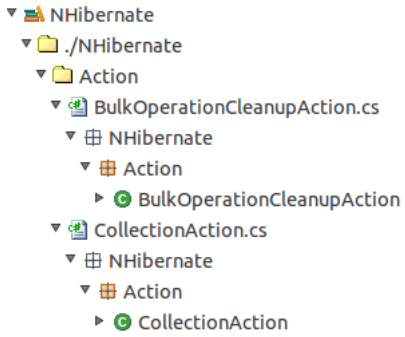


Figure 5.6. Physical Namespaces

In the image, two source files are displayed, BulkOperationCleanupAction.cs and CollectionAction.cs. The C# parser detects that below each one of them we have the namespace NHibernate.Action; on the physical level they are both independent and have no relation. On the logical level on the other hand, the content will look like this:

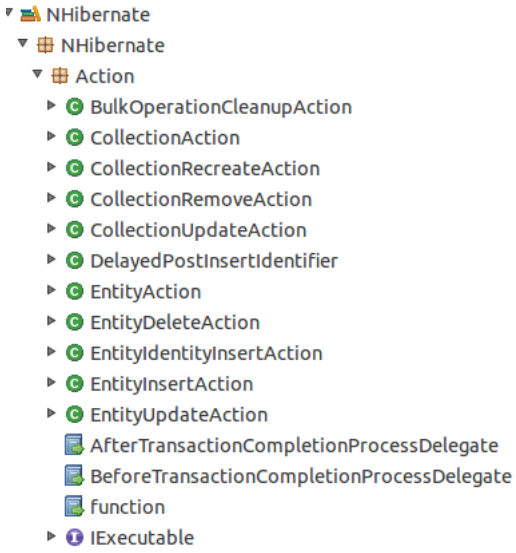


Figure 5.7. Logical Namespaces

As it can be inferred from the images, *Sonargraph* maps all physical namespaces that have the same name into a single *logical namespace*. This mapping can be system-based or module-based, see Section 5.4.1, “System-Based Logical Model” and Section 5.4.2, “Module-Based Logical Model” for more information.

Logical Programming Elements

Logical Programming Elements construction from Programming Elements is not as simple as *logical namespaces* construction and it is language-specific.

- Java: Logical Programming Elements are mapped 1 on 1 to Programming Elements.
- C/C++: When programming C or C++, there are declarations/definitions for Programming Elements such as classes, structs, unions, routines, variables and namespaces. In this case, the declaration(s) and definition(s) are mapped into a single Logical Programming Element. All other Programming Elements that do not follow the declaration/definition approach will be mapped 1 on 1 to Logical Programming Elements.
- C#: Logical Programming Elements are mapped 1 on 1 to Programming Elements except for partial types; in their case, all partial types that contribute to the same definition are mapped into a single Logical Programming Element.

The construction of Logical Programming Elements can be system-based or module-based, see Section 5.4.1, “System-Based Logical Model” and Section 5.4.2, “Module-Based Logical Model” for more information.

5.4.1. System-Based Logical Model

After parsing the source files from any language, *Sonargraph* creates a system-based logical model based on the parser model which correspond to following diagram:

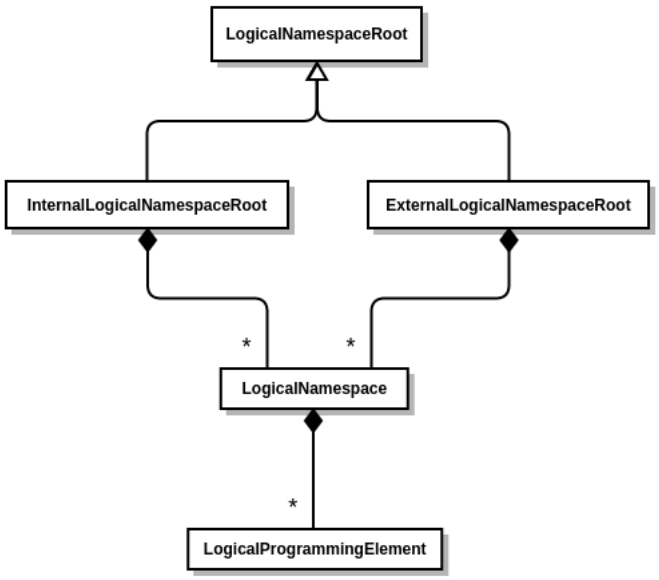


Figure 5.8. System-Based Logical Model

The system-based logical model is constructed in a way that the mapping from physical elements to logical elements occurs in the internal and external scopes separately meaning that the following conditions will be met:

- Given a physical element "abc" inside a module of the user code and a physical element "abc" inside the external elements, there will be a logical element "abc" belonging to the InternalLogicalNamespaceRoot and another logical element "abc" belonging to the ExternalLogicalNamespaceRoot in the system-based logical model.
- Given a physical element "abc" inside a module X of the user code and a physical element "abc" inside the module Y also in the user code, there will be a single logical element "abc" belonging to the InternalLogicalNamespaceRoot in the system-based logical model.

5.4.2. Module-Based Logical Model

After parsing the source files from any language, *Sonargraph* creates a module-based logical model based on the parser model which correspond to following diagram:

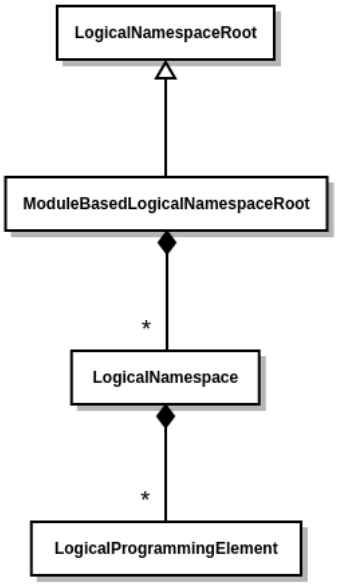


Figure 5.9. Module-based Logical Model

The module-based logical model is constructed in a way that the mapping from physical elements to logical elements occurs inside each module and in the external scope separately meaning that the following conditions will be met:

- Given a physical element "abc" inside a module X of the user code and a physical element "abc" inside the module Y also in the user code, there will be a ModuleBasedLogicalNamespaceRoot X containing a Logical Programming Element "abc" and another ModuleBasedLogicalNamespaceRoot containing also a Logical Programming Element "abc".
- Given a physical element "abc" inside a module X of the user code and a physical element "abc" inside the external scope, there will be a ModuleBasedLogicalNamespaceRoot X containing a logical element "abc" and another logical element "abc" belonging to the ExternalLogicalNamespaceRoot in the module-based logical model.

Chapter 6. Creating a System

Basic working units in the *Sonargraph* workspace are called *modules*. A *system* consists of one or several modules representing the *components* that your product is made up of. Each *module* contains one or several root directories pointing out to the source code or the executable artifacts.

At the menu "File" → "New" → "System" *Sonargraph* provides different wizards to easily create *software systems* . You can either create an empty system and manually add modules to it or use one of the language based wizards.

If you need to have modules from different languages in the same system you can add those of the second language later, regardless of the type of system you have created. See Chapter 7, *Adding Content to a System*

All wizards contain a page where you can specify the system's name, a short description for it and the local directory where you want to create the system. Optionally, you can use a predefined quality model for the new system. See Section 6.4, “Quality Model”

To create an empty system to which you can add modules later select "File" → "New" → "System" → "New System...". You will be asked for a system name and a storage directory for the Sonargraph system folder. See Chapter 7, *Adding Content to a System* for how to add modules to your system.

TIP

It is always smart to store the Sonargraph folder at the root of your project because its content needs to be added to your version control system. This folder does not contain any binary files, all content of the Sonargraph system definition is contained in plain text files, making it easy to track changes.

6.1. Creating a Java System

Sonargraph offers different methods to create Java systems:

- **System based on Java Eclipse workspace:** See Section 7.1.1, “Importing Java Modules Using an Eclipse Workspace”
- **System based on Gradle:** See Section 7.1.2, “Import Modules using the Sonargraph Gradle Plugin”
- **System based on Maven:** See Section 7.1.3, “Import Modules using the Sonargraph Maven Plugin”
- **System based on Bazel:** See Section 7.1.4, “Importing Java Modules Using a Bazel Workspace”
- **System based on Build Unit(s):** See Section 7.1.5, “Import Modules Using the Build Unit(s) Importer”

NOTE

If you plan to use our Eclipse or IntelliJ plugins, place the Sonargraph system in a directory that is parallel to your modules and not part of any of your Eclipse or IntelliJ modules. Otherwise executing Sonargraph refactorings might easily corrupt the system's information, if the Sonargraph files are not excluded from modifications during refactoring execution.

6.2. Creating a C# System

You can import directly from a Visual Studio solution file. After that you can add additional modules from the same solution file.

6.3. Creating C/C++ Systems

Creating a C/C++ system is a bit more complex than creating a system for other languages. First we need to select or create a compiler definition. Then we need to define the required include directories for each module as well as the macro definitions required for conditional compilation. Sometimes it is also necessary to exclude certain compilation units from modules. The "Create New C/C++ System..." wizard gives you maximum flexibility to specify all that. But if you use CMake or Visual Studio you can also import the system more conveniently.

The first page of each C/C++ system creation wizard will allow you to select an existing compiler definition or create a new one. If you decide to create a new compiler definition the next wizard pages will guide you through this process step by step.

If you use the "Create New C/C++ System..." wizard please make sure to select the root directory of your system as the storage location for the Sonargraph folder. Only source files located directly or indirectly under this directory can be added to the system. The wizard will scan all files under this directory for "#include" statements and will try to locate the referenced include files. The scanner does *NOT* consider conditional compilation, so you might see lots of irrelevant unresolved include references that you can ignore safely. By adding additional include folders you can make sure that all relevant include references can be resolved.

While we provide many different choices for C/C++ project setup I recommend to use the ccspy wizard (see below) for project setup. The documentation for ccspy can be found here: https://github.com/sonargraph/ccspy. The basic idea is that ccspy works as an intermediary between your build tool (e.g. make) and your compiler. Each time a file is compiled ccspy will record the used compiler options in a file in the ccspy target directory, which will then be used by Sonargraph to analyze your project.

Here are the other wizards to create new C/C++ systems:

- **System based on C/C++ CMake JSON command file:** Allows to create a system out of a generated compile command JSON file.

  Name your new system and choose a directory to store it. In the next wizard page you need to choose the location of your JSON command file. To generate such a file you need to run cmake with -DCMAKE_EXPORT_COMPILE_COMMANDS=ON.

  The next wizard page presents the root directories found in the JSON file and allows you to fine tune those directories and sub-directories you want marked as root directories or excluded in the resulting system. You need to mark at least one root directory:

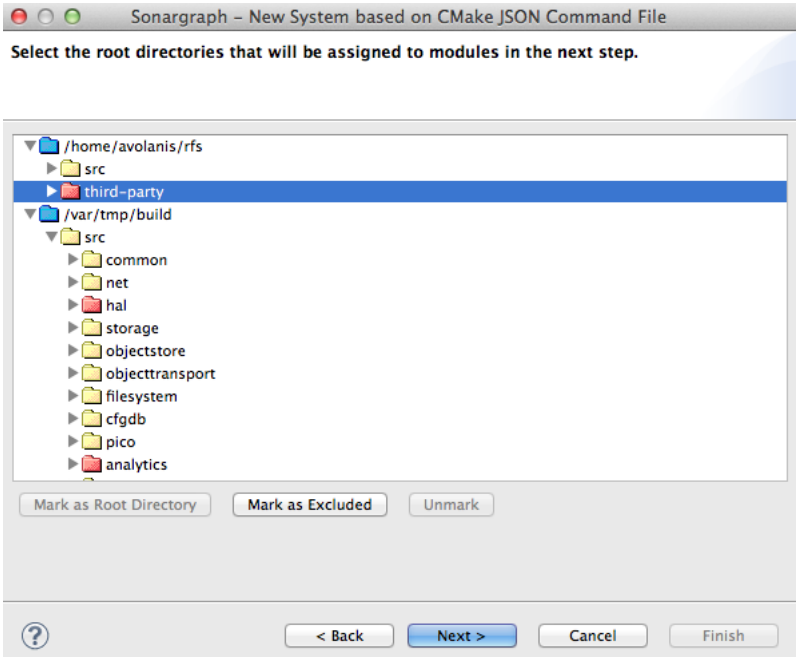


Figure 6.1. Marking root directories from JSON file

The final page of the wizard allows to give a name to each one of the modules that will be created out of the root directories marked in the previous step. Sonargraph will try to guess a module name out of the root folder name. You are able to change that name if it should not fit.

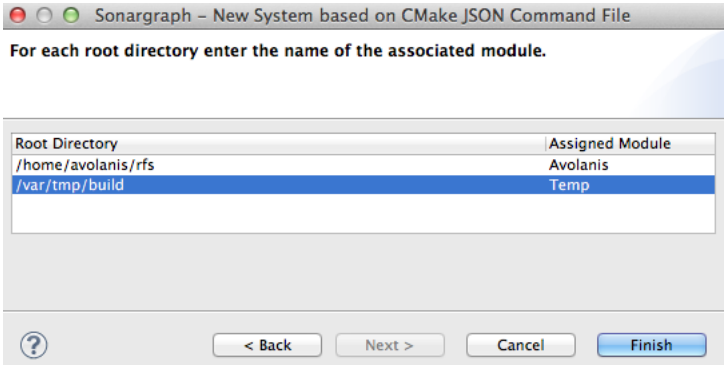


Figure 6.2. Naming modules for root directories from JSON file

- **System based on ccspy target directory:** Works pretty identical to the cmake JSON importer (see above), except that you have to specify the ccspy target directory. Make sure to build your system before you analyze it to ensure the most up-to-date input.
- **System based on C/C++ Visual Studio 2010 (or newer) Solution file:** See Section 7.2.1, “Importing C++ Modules from Visual Studio Files”

Most wizards are similar whether you create a new system or add modules to an existing system.

6.4. Quality Model

Sonargraph defines a "Quality Model" as a group of settings and files aimed to help you getting started with your code analysis.

The components of the quality model are displayed in the Files view. See Section 8.7, “Managing the System Files”

When creating a new system you can optionally use one the pre-defined quality models that ship with *Sonargraph* . The default quality model suggested depends on the type of system you want to create: If you are creating a system manually, you get the Core quality model suggested, which contains language-independent settings and scripts. If you are creating a new software system using one of the language-based wizards, you will get a quality model customized to the corresponding programming language.

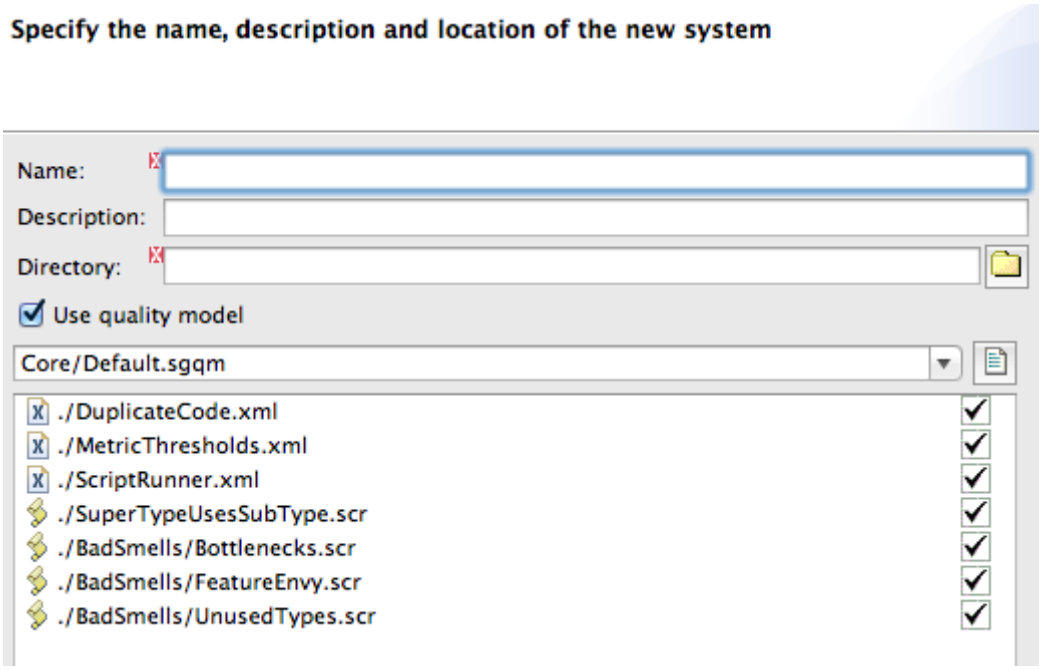


Figure 6.3. New System with Quality Model

You can include or exclude quality model elements as you see fit for each project.

6.4.1. Importing a Quality Model

You can import an external quality model file, generated with a different installation of *Sonargraph* into the current *software system* via the menu "File" → "Import Quality Model" .

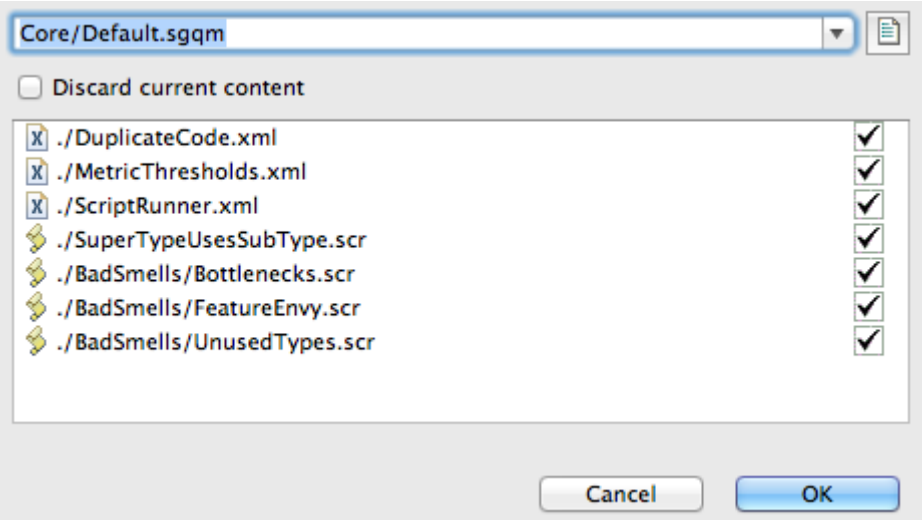


Figure 6.4. Import Quality Model

Check "Discard current content" if you want to delete all the configurations and scripts currently loaded and start afresh with the imported quality model elements.

NOTE

If don't discard your current content, quality model elements with equal names will still be overridden by the incoming elements!

6.4.2. Exporting a Quality Model

To export the currently used quality model select "File" → "Export Quality Model" :

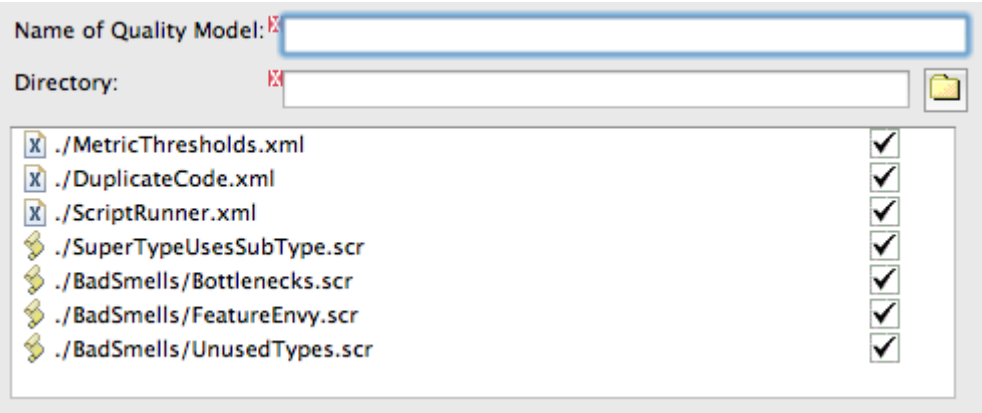


Figure 6.5. Export Quality Model

Select the quality model elements to be included in the resulting file with *.sgqm* extension.

Chapter 7. Adding Content to a System

Sonargraph supports both the manual creation of programming language specific *modules* and the usage of external sources like Eclipse or IntelliJ workspaces, Visual Studio solution or project files to setup the workspace automatically.

The following sections describe the different ways you can add content to a *software system*.

7.1. Creating or Importing a Java Module

There are several ways to add Java modules to a *software system*.

7.1.1. Importing Java Modules Using an Eclipse Workspace

You can import Eclipse projects as modules into an existing *Sonargraph* project or while creating a new system.

To import Eclipse projects as modules directly into an already existing *Sonargraph* project use "File" → "New" → "Module" → "Java Modules from Eclipse Workspace" .

Select the location of the Eclipse workspace you want to import projects from. You can choose those projects and root directory paths that should be imported and those that should not. The imported Eclipse projects become modules in the *Sonargraph* workspace.

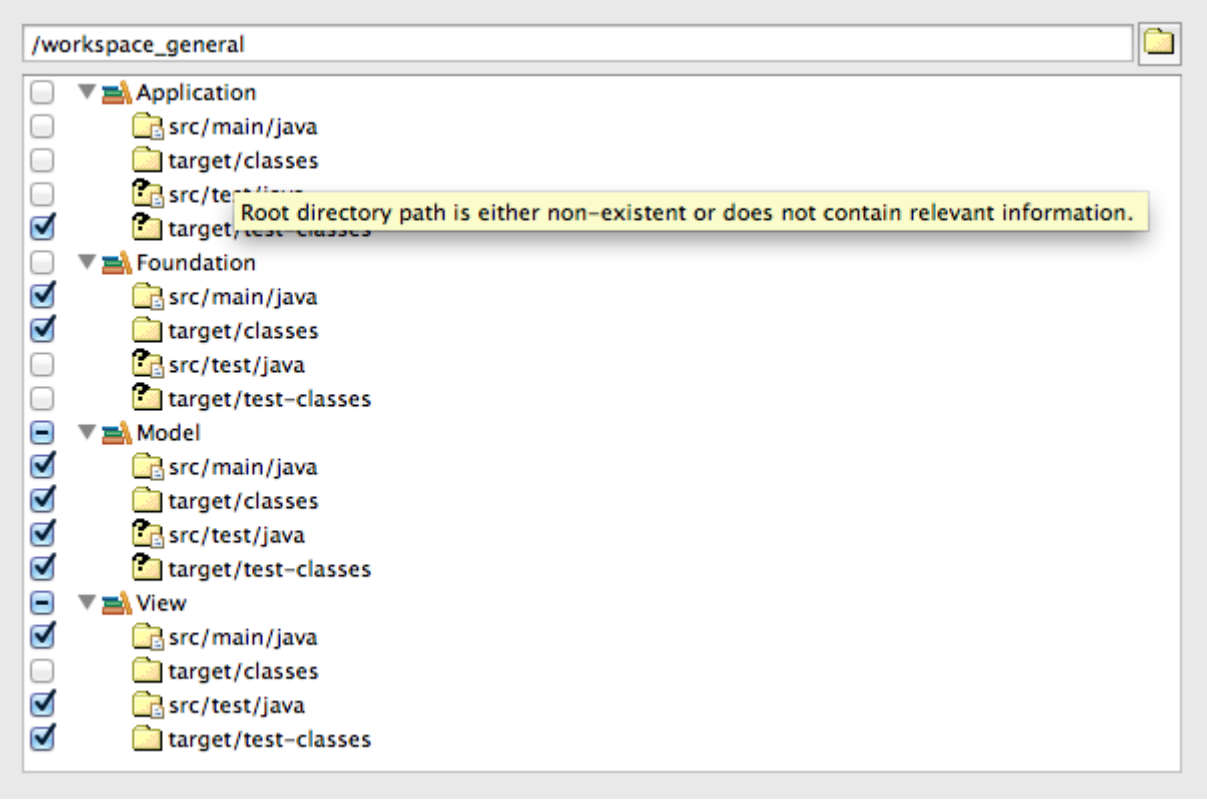


Figure 7.1. Importing Java Modules Using an Eclipse workspace

Sonargraph will let you know about content that is already in the *software system* , empty or irrelevant directory paths and dependencies between modules.

7.1.2. Import Modules using the Sonargraph Gradle Plugin

To create or update a Java system based on a Gradle build you have to use the Sonargraph Gradle Plugin invoking the task 'sonargraphCreateOrUpdateSystem'. The plugin makes use of Sonargraph Build.

NOTE: No license is required to use that task!

NOTE: It is not necessary to download Sonargraph Build manually since this can be performed automatically!

For details see the Gradle documentation in the Sonargraph Build manual: *Sonargraph Gradle plugin task 'sonargraphCreateOrUpdateSystem'* .

7.1.3. Import Modules using the Sonargraph Maven Plugin

To create or update a Java system based on a Maven build you have to use the Sonargraph Maven Plugin invoking the goal 'create-or-update-system'. The plugin makes use of Sonargraph Build.

NOTE: No license is required to use that goal!

NOTE: It is not necessary to download Sonargraph Build manually since this can be performed automatically!

For details see the Maven documentation in the Sonargraph Build manual: *Sonargraph Maven plugin goal 'create-or-update-system'* .

7.1.4. Importing Java Modules Using a Bazel Workspace

You can import a Bazel workspace as a single module, or multiple modules (per Bazel build file, or per Bazel rule), into an existing *Sonargraph* system or while creating a new system.

NOTE

For the *Sonargraph* Bazel import to work, a 'bazelisk' or 'bazel' executable must be found either in Bazel's workspace root directory, or on *Sonargraph's* path.

Required Bazel version is 2.0.0 minimum.

Supported 'bazel rules' are 'java_binary', 'java_library', and 'java_test', others may be added in the future.

To create a new *Sonargraph* system from a Bazel workspace as a single module, or multiple modules, use "File" → "New" → "System" → "New Java System Based On Bazel Workspace" .

To import a Bazel workspace as module(s) directly into an already existing *Sonargraph* system use "File" → "New" → "Module" → "New Java Module(s) Based On Bazel Build Files" .

To import a single Bazel build file as a module directly into an already existing *Sonargraph* system use "File" → "New" → "Module" → "New Java Module(s) Based On Bazel Workspace" .

Select the location of the Bazel workspace you want to import modules from. Decide first if you want to import the whole workspace as a single module, or as multiple modules. For multiple modules it is possible to use either the bazel output jars, or the directories as *Sonargraph* root paths.

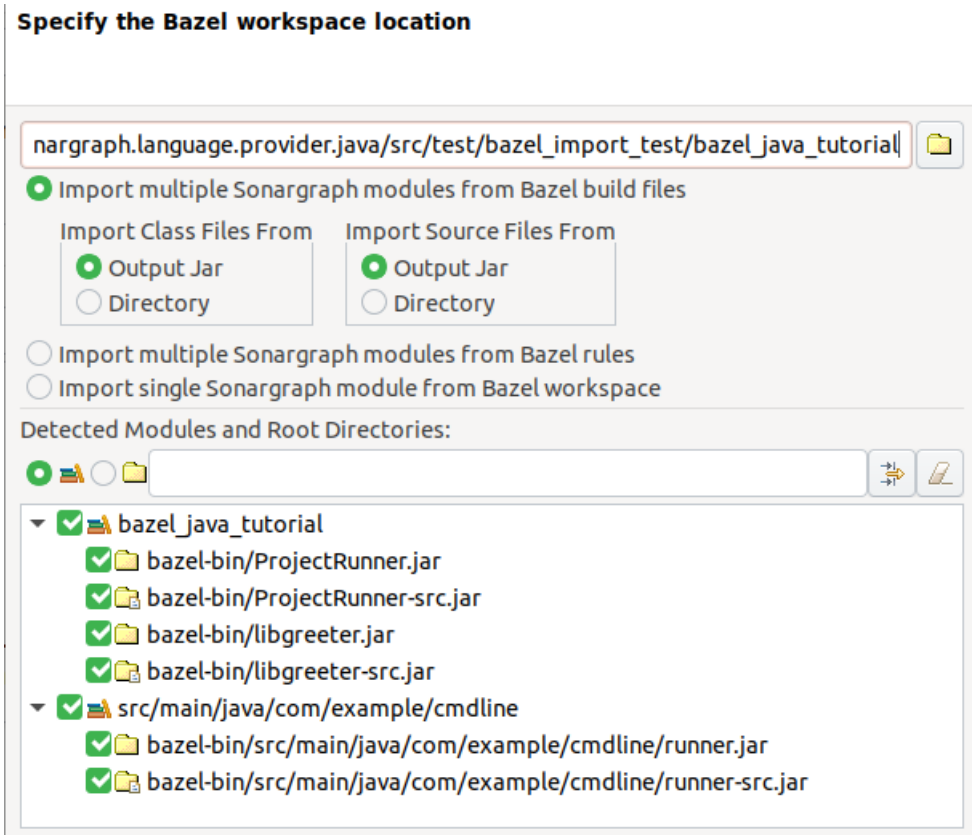


Figure 7.2. Importing Java Modules Using a Bazel workspace

Sonargraph will let you know about content that is already in the *software system* , empty or irrelevant directory paths and dependencies between modules.

7.1.5. Import Modules Using the Build Unit(s) Importer

You can import Java modules based on build units into an existing *Sonargraph* system or while creating a new system.

Note: This importer can only work when you have previously build your system (hence the name). Java class and source files are analyzed to obtain class and source root information. A build unit contains Java class and corresponding source roots and corresponds to a module.

In a multi-module system setup normally the modules share common root directory structures (or definitions). The following root directories could be used in such a multi module system:

- ./src/main/java (Java production code)
- ./src/test/java (Java test code)
- ./target/generated (generated Java code)
- ./target/classes (compiled Java production code)
- ./target/classes-test (compiled Java test code)

The main idea of the build unit importer is to collect at least 1 root directory containing compiled Java code (i.e. byte code) and at least 1 root directory containing corresponding Java source code. The common root directory of both would compromise the module root directory. So lets say the directory 'Common' contains './src/main/java' and './target/classes' containing byte code and source code the build unit import would recognize 'Common' as a build unit (i.e. module) with 2 root directories.

The build unit importer uses 2 important terms:

- *Root Segment*

  Begins with '/' followed by full directory name or a part of it. It has only 1 '/'! It is used to recognize *Root Definitions*.

  In order to recognize './src/main/java' from the above example it would be necessary to define 3 root segments: '/src', '/main' and '/java'.

- *Root Definition*

  './src/main/java' from the above example would be a root definition. It consists of 1 or more '/' and full directory name pairs.

In general the following 3 steps need to be performed:

- Select the directory containing the build units you want to detect. The wizard will detect *Class/Source Root Definitions* based on predefined *Root Segments*. The *Root Segments* can be modified to obtain better *Class/Source Root Definition* matches. This will trigger the re-detection of *Class/Source Root Definitions*. In *Unassigned Class/Source Roots* you will see all roots that are currently not assigned to any module candidate.
- Adjust the *Root Segments* and/or *Class/Source Root Definitions* (via context menu entries 'Add...', 'Edit...' and 'Delete') and rerun the import candidate detection until you obtain the desired Sonargraph import candidates (i.e. modules and class/source root directories).
- Tweak the obtained Sonargraph import candidates by including/excluding selected entries.

Some things to keep in mind:

- The wizard detects only the root directories that are not already contained in the Sonargraph workspace.
- Modules containing only source roots are not checked by default, since Sonargraph needs the class files too.
- Module candidates can be renamed via the context menu 'Edit...' entry.
- To reset the wizard to it's initial state simply open the 'Select Directory' dialog and close it with 'Cancel'.

- You can either modify *Root Segments*, *Root Definitions* or both to obtain the desired Sonargraph import candidates.
- In the *Root Segments* and *Root Definitions* viewers you can use a search widget to find entries. *Ctrl+Shift+F* on Windows and Linux and *Cmd+Shift+F* on Mac.

7.1.6. Creating a Java Module Manually

Select "File" → "New" → "Module" → "Java Module" . Alternatively, the context menu in the Navigation and Workspace views can be used.

Sonargraph relies on the Java byte code for its static code analysis. For the ability to show dependencies in the source code, the source directories must be provided as well. Source Root Directories and Class Root Directories can be added individually using the corresponding context menu entries.

Alternatively, a dialog is available via the context menu "Manage Java Source/Class Root Directories/Archives..." that allows the automatic detection of Source and Class Root Directories. The detected directories can be assigned to Java Modules via drag and drop.

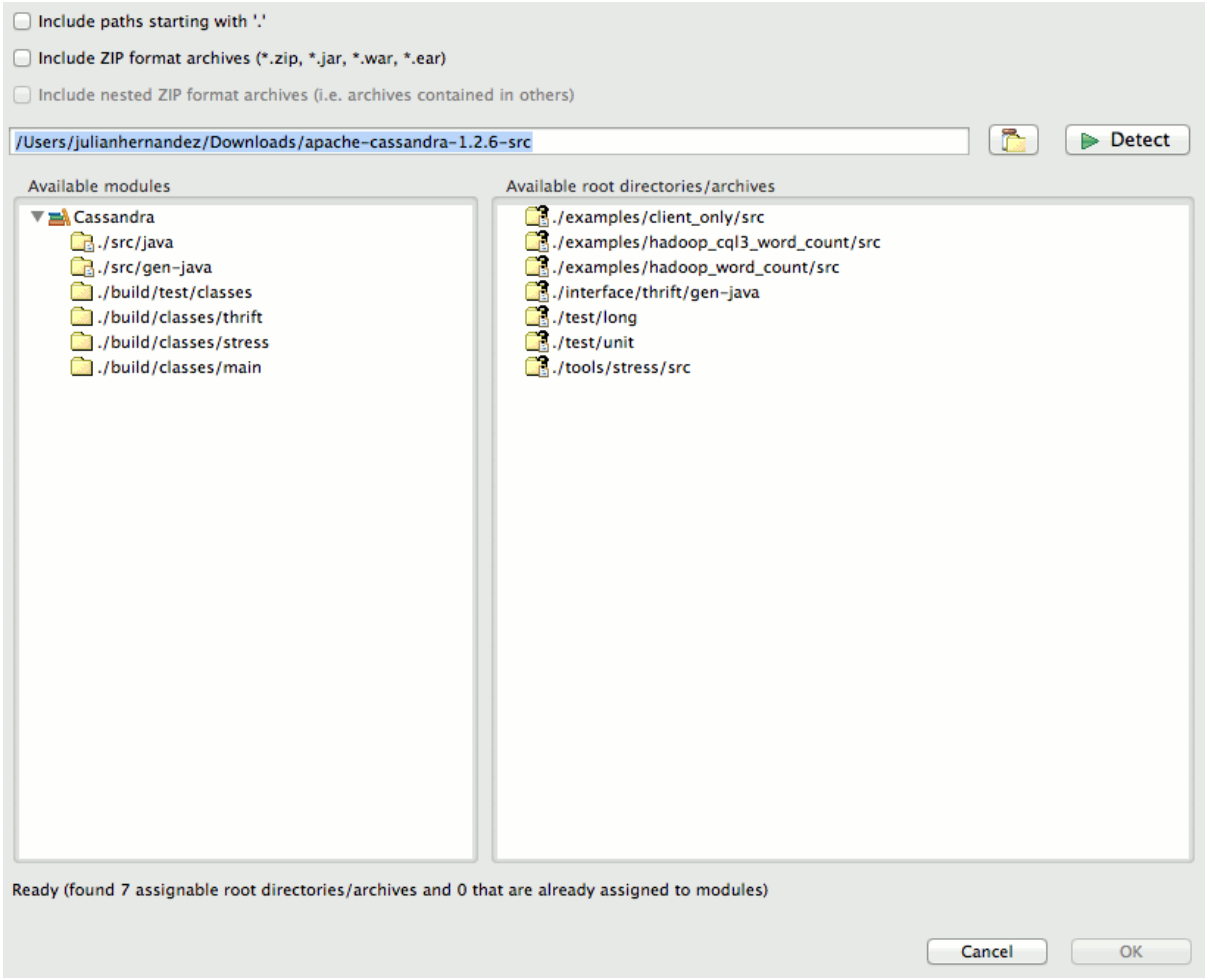


Figure 7.3. Manage Root Directory Path

7.2. Creating or Importing a C++ Module

After a *software system* has been created, there are several ways to set up C++ modules: Import from a Visual Studio 2010 Project file (.vcxproj), import via Makefile command capturing files or manual module creation.

7.2.1. Importing C++ Modules from Visual Studio Files

Via the menu entry "New" → "Module" → "C/C++ Module from Visual Studio Project file" a C++ module can be created based on a .vcxproj file. Select the project file and the required configuration. The same approach applies for creating a system based on a Visual Studio Solution file (.sln) as shown in the following screenshot:

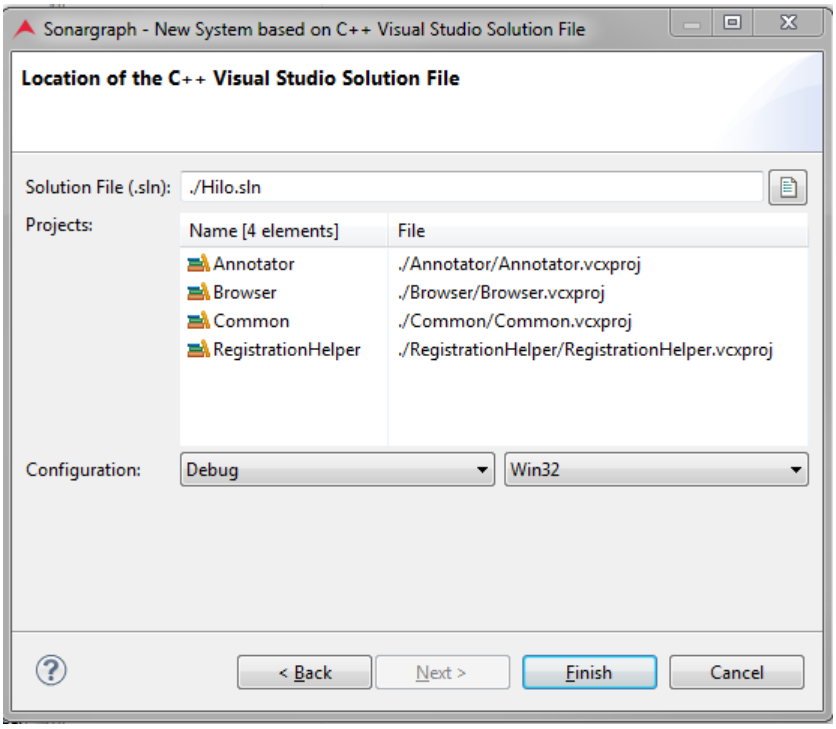


Figure 7.4. Create C/C++ System Based on Solution File Import

7.2.2. Importing C++ Modules Via CMake or CCSpy

Select "New" → "Module" → "C/C++ Modules based on Command File / CCSpy Directory]"

You will have to do a complete rebuild of your system using make (with ccspy as your compiler) or cmake. Then the wizard will guide you through the remaining steps needed to add modules to your system.

7.2.3. Creating a C++ Module Manually

Via the menu "New" → "Module" → "New C/C++ Module(s)" plain C++ modules can be created. The wizard will guide you through the process and will allow you to select root directories and assign them to new modules. You can also specify extra include directories and macro definitions for conditional compilation.

NOTE

The Sonargraph folder must be stored in the root directory of your system. You can only add modules that are located under this root directory.

7.2.4. C/C++ Module Configuration

For configuration of additional compiler options, select the menu entry "System" → "Configure C/C++ Module(s)..." . This dialog allows to configure options using Groovy templates. You can define system wide options and/or module specific options. System wide options will be applied to all modules. You can also define options that are not specific to any compiler by selecting "Any Compiler" from the compiler definition drop down list. The effective options for a module are the system wide options for "Any Compiler", then the system wide options for the active compiler, then the module specific options for any compiler followed by the module specific options of the active compiler.

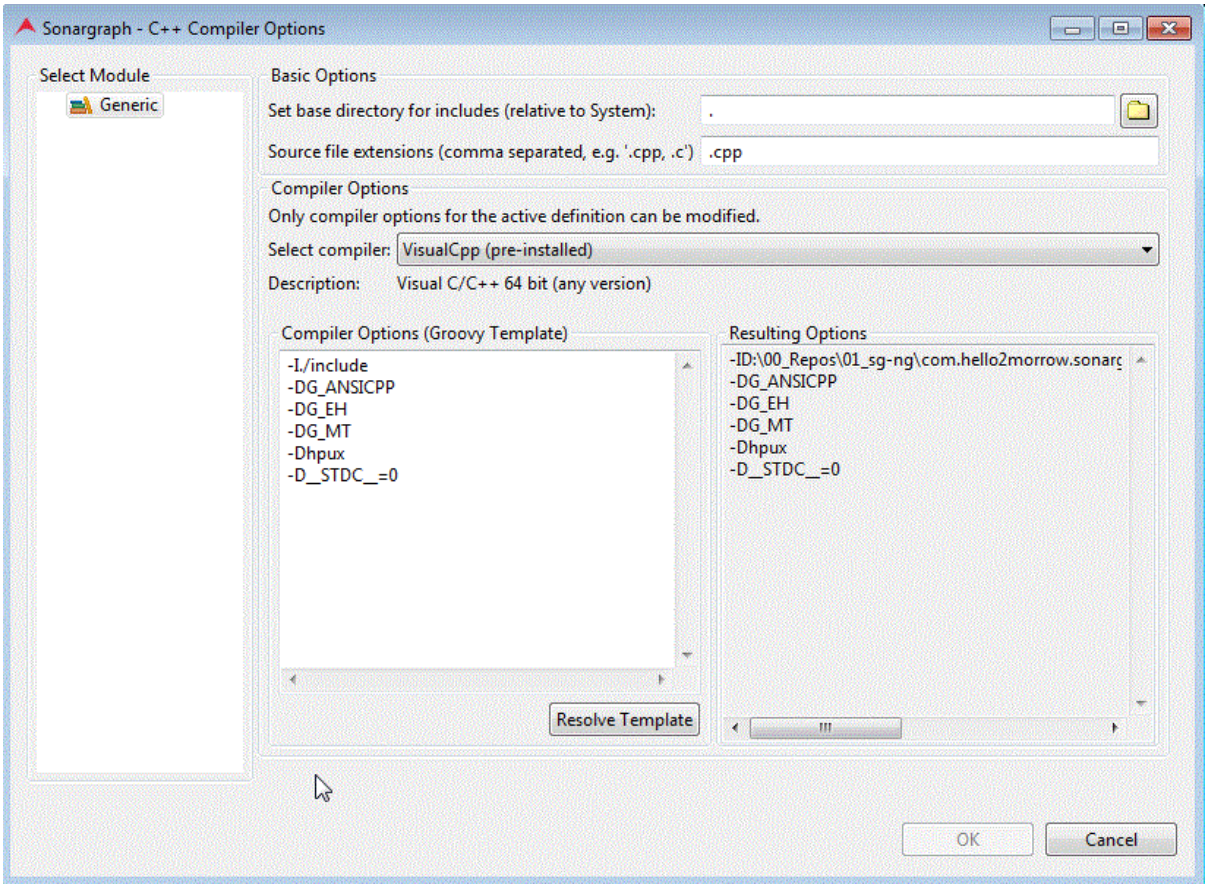


Figure 7.5. C/C++ Module Configuration

TIP

The compiler options can be verified via the menu "System" → "Execute C/C++ Preprocessor". This is usually seven times faster than a full refresh. Problems of the preprocessor are reported in the "C/C++ Parser Log" view.

7.3. Creating or Importing a C# Module

After a *software system* has been created, you can always change the selection of analyzed modules from your solution file.

7.3.1. Importing C# Modules Using a Visual Studio Solution File

You can import C# modules from a Visual Studio solution file either when creating a new system using the corresponding wizard or selecting "File" → "New" → "Module" → "Update C# module selection".

Select the location of the solution file (.sln) to have *Sonargraph* offer you a list of analyzable modules.

Chapter 8. Interacting with a System

This chapter describes how the views of *Sonargraph* can be used to interact and explore a system and conduct basic use cases like examining duplicates and cyclic dependencies. More advanced functionality like adding custom metrics or defining an architecture are explained in their own chapters.

8.1. User Interface Components

This chapter describes different graphical components of the main application window and additional frequently used components.

8.1.1. Menu Bar

Contains menu entries that allow the execution of system-wide actions or commands that are applicable for the current selection; the meaning of the categories is the following:

- **File:** Contains commands for creating, loading and saving systems as well as exporting or importing Quality Models (See Section 6.4, “Quality Model”), creating Microsoft Excel, HTML and XML reports.
- **Edit:** Contains commands for undoing and redoing the last performed operations, editing and deleting components and perform system-wide searches.
- **System:** Contains commands that allow re-reading the current *system* from the disk (and re-checking it), perform system configuration and changing language specific module settings such as where source and compiled files are being searched for.

  Additional entries allow creation of fix, ignore or TODO issues.
- **Window:** Provides access to the different views of *Sonargraph* and the preference pages to modify installation-wide settings.
- **Help:** This menu provides access to the online and dynamic help, allows management of license information, allows to send feedback to the *Sonargraph* developers and provides general information about the installation.

8.1.2. Tool Bar

Figure 8.1. Tool Bar

Allows to access the most common operations. It is always visible regardless of the active view. It offers the following actions:

- **Refresh** : If there is currently no representation of the system in memory it performs a full parse. If the model exists, it performs a "delta refresh" to update the in-memory content with the latest state from the disk. The synchronization of the model with the disk content is normally not done automatically on startup, because this can take a considerable amount of time. However, it can be specified that on opening the software system, a synchronization should be performed automatically by checking the menu item "System" → "Refresh On Open" .
- **Clear** : Drops the memory representation of the system under consideration. After performing this action a full parse of the system is required to resume with the analysis.
- **Navigate Backward/Forward** : Allows to navigate backward and forward in the history of recently performed actions across the application.
- **Manage Virtual Models** Modifiable.vm : Allows to change the current *virtual model* or create a new one. (See Section 9.1, “Using Virtual Models for Resolutions”)

8.1.3. Notifications Bar

Figure 8.2. Notifications Bar

The notification area is located in the bottom area. It informs about the current operation being performed with both a text feedback and a progress bar. On right side, you find notifications about different situations going on in the application that may be of interest to you such as proximity of license or support expiration date and proximity to reaching the limit of available elements per the active license. Specifically:

- The icon indicates that there is at least one information notification available.
- The icon indicates that there is at least one warning notification available.
- The icon indicates that there is at least one error notification available.
- The icon indicates that there are no notifications available at this time.

To bring up notifications just click once on the icon.

8.1.4. Tables

Tables of views that potentially display huge amount of data like the Issues view can be filtered. To bring up the text filter as shown in the screenshot below, use the key combination **Ctrl+Shift+f** . A row containing the text in any table cell will be shown. Pressing **Return** activates the filter, pressing **Escape** clears it and displays again all items. A yellow background indicates if any elements are filtered. Most tables can also be sorted by clicking on their column headers.

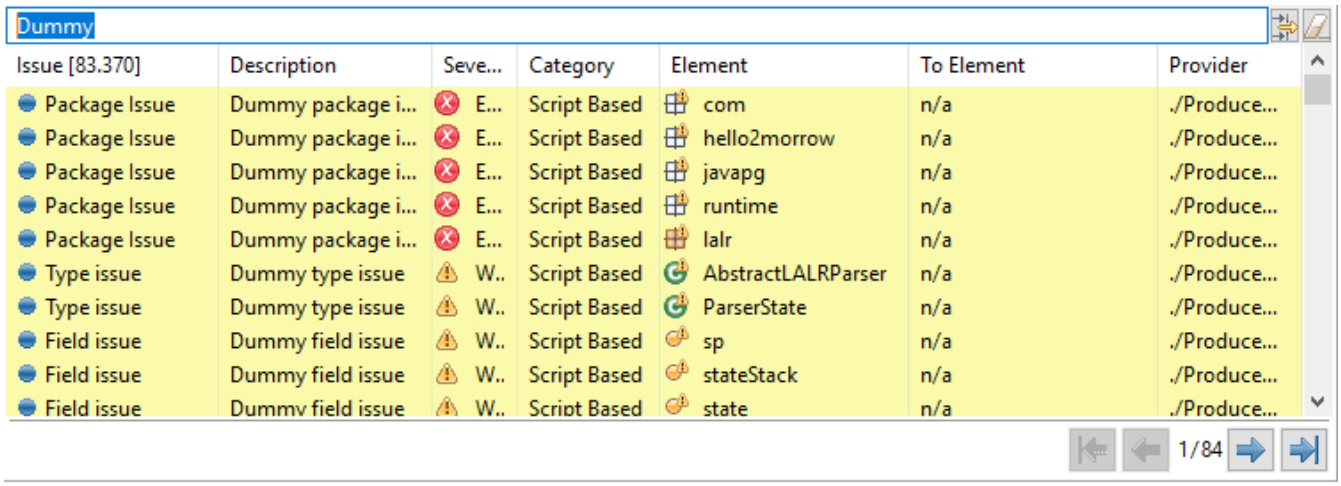


Figure 8.3. Table with activated text filter

8.2. Common Interaction Patterns

The following interaction patterns (called gestures) are common across the *Sonargraph* application:

- **Single clicking** on an element normally means to select it; holding the control key (command key on Mac) while clicking normally aggregates the selection elements, and holding the SHIFT key normally selects all elements between the current one and the last selected one.
- **Double clicking** not only selects the element but, if possible, shows it in a view that is best suited to inspect it or edit it. It is important to note that a double click gesture will show the selected element in another view only if at least one of the following two conditions is fulfilled:
  - Element is associated with a single source file: Elements like C/C++, Java and C# source files, types, structs, methods, and functions among others fulfill this conditions. For other elements like namespaces, packages or directories it is not possible to associate them with a single source file.
  - There is a single possibility of navigation: If the selected element only offers one view to navigate to, then the double-click gesture will show the element in that view, otherwise, it will not meet this condition.
- **Right clicking** on elements normally presents a context menu with element-specific actions. Some of the most common interaction patterns available with right-click are showing elements in different views, exporting tables to Excel and exporting graphics as images to the file system among others.
- **Drag and drop** is used in several different contexts in *Sonargraph* to perform different operations: filters can be re-organized in the Workspace view. Nodes can be re-arranged for better appreciation in the Graph and Cycle views while holding the SHIFT key pressed. Dragging and dropping the mouse cursor while holding the SHIFT key and the primary modifier key of the platform (CTRL on Windows/Linux and CMD on Mac) pressed in the Workspace Dependencies view allows to specify a new dependency between two nodes (see Section 8.8.2, “Managing Module Dependencies”).

8.2.1. Special Graphic Elements Decorations

Across the *Sonargraph* is common to find two decorations:

- * : A star behind the name of an element means that the description of such element has been changed locally but not yet saved to disk: The in-memory representation and the disk representation of the system are not identical.
- ! : An exclamation mark behind the name of an element generally means that this element needs your attention or requires you to take some action on it.

8.3. Sonargraph Workbench

The default workbench of *Sonargraph* is divided into 4 regions. However, as *Sonargraph* is built upon Eclipse's Rich Client Platform you can always re-arrange views as you like.

The following image shows these regions and the subsequent sections explain each one of them:

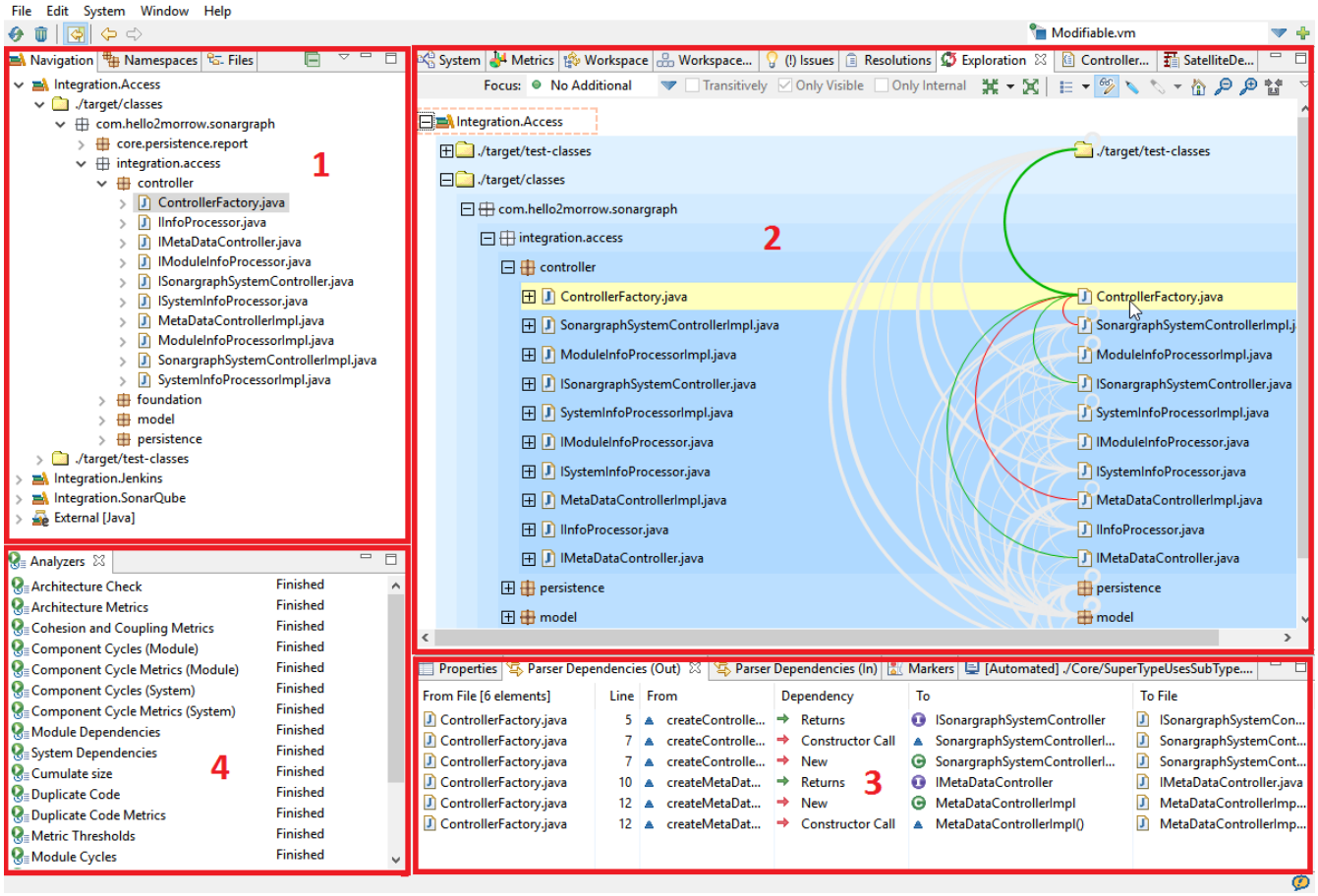


Figure 8.4. Sonargraph Workbench

1. **Master Views:** Located at the upper left hand side of the workbench, provide control over the system structure and the files that make it up. All Master views offer the following operations:

   - **Collapse All** : Collapses the whole tree of elements.

   - **Link** : Selecting it specifies if the selection in the current Master view should be synchronized as far as possible with the selection in the currently selected Slave view.

   The Navigation and Namespaces views offer a "View Menu" option which can be used to specify whether the elements of the tree are to be displayed in a flat mode or in the hierarchy induced by their dot-separated full paths. Exclusive to the "View Menu" option of the Namespaces view is the possibility to choose between system-based or module-based representations.

2. **Slave views:** Located at the upper right-hand side of the workbench provide ways to manage and explore the *components* of the system under consideration. The slave views have the capability of responding to selection from the master views.

3. **Auxiliary views:** Located at the lower right-hand side of the workbench, provide support to some of the slave views to expand their system exploring capabilities.

4. **Information views:** Located at the lower left hand side of the workbench, provide information about the status of the analyzers running over the system model.

8.3.1. Auxiliary Views

The Sonargraph workbench offers several auxiliary views. Those views react on selection in other views and display information related to the selected elements.

Parser Dependencies Views

There are 2 parser dependencies views:

- Parser Dependencies (In)
- Parser Dependencies (Out)

Those views react to selection of the following views:

- Source View
- Dependencies View
- Graph View (if based on parser dependencies) including the Cycle View
- Exploration and Architectural View

The parser dependencies views use a pin mechanism which ties them to a specific supported view. If more than 1 supported view is open the user can explicitly tie them to 1 of the supported views. That helps to not loose track of what is currently being inspected. The pin button's icon and tool tip show information about the view that is pinned to the corresponding parser dependencies view.

- *Not pinned to any view*  : The corresponding parser dependencies view is not pinned to any view.
- *Pinned to focused view*  : The corresponding parser dependencies view is pinned to the currently focused view.
- *Pinned to another view that is not focused*  : Pressing the pin button will pin the corresponding parser dependencies view to the currently focused view.

NOTE: When selecting a dependency or an arc representing a dependency in 1 of the supported views both parser dependencies views will show the same content to ease the usage. When selecting a node with dependencies in 1 of the supported views the Parser Dependencies (In) view will show the incoming dependencies and the Parser Dependencies (Out) view the outgoing dependencies.

Properties View

This view reacts to selection of all views except the help views and shows the properties of the selected element if there are any.

Markers View

This views shows error/warnings of the following supported views:

- Source View
- Script View
- Architecture (DSL) File View

Temporal Coupling View

This view shows the temporal coupling of a selected source file. Temporal coupling occurs when several files are committed together into the version control system (VCS). The view lists all files that the selected file has been committed together with over the last 5 years. The number in the weight column represents the number of shared commits in that time frame.

This can be quite useful when you are working on legacy systems and want to understand dependencies between source files. A shared commit indicates some kind of semantic connections between files.

8.4. Getting a Quick Impression

In version 12 we release a reworked "System" view. There are boxes for different quality aspects, like "Structure", "Complexity" and "Code Organization". Each box contains a few key metrics, with the option to expand the box via the "+"-icon to see more details. A couple of new metrics like "Entanglement (%)" have been introduced, so that it is now easy to see how much of the code base is affected by a certain type of issue, e.g. "cycle groups". These percentages also indicate the probability that a developer is affected by the problem if she looks at any line of code. These values are easier to interpret than a rather abstract numeric metric value like "Average Component Dependency".

If a baseline report has been applied, metric trends are shown. This makes it very easy to spot where the quality of the code base needs more attention.

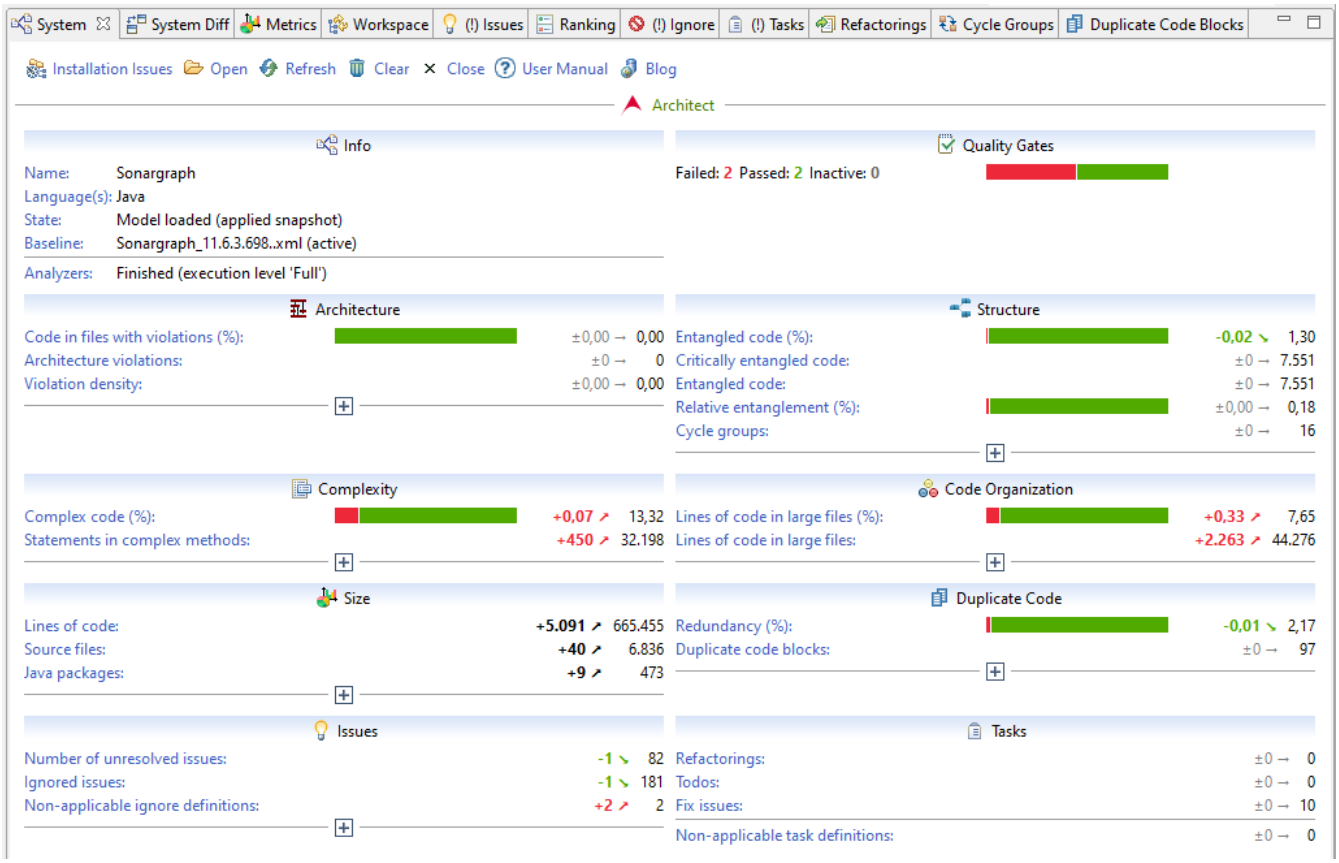


Figure 8.5. System View

8.5. Navigating through the System Components

The Navigation view presents the directory -or archive- structure of the source and/or binary files of the loaded *system* as it has been determined from the workspace defined for the modules of the system.

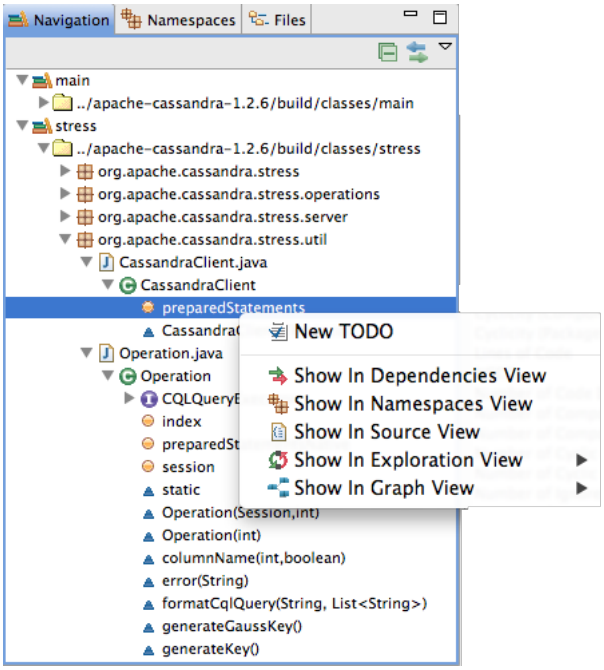


Figure 8.6. Navigation View

The context menu interaction gives you options to inspect elements in suitable slave views or perform element specific actions such as creating a TODO task (see Section 9.4, “Defining Fix and TODO Tasks”).

8.6. Exploring the System Namespaces

In order to be able to see and explore the logical models calculated by *Sonargraph* (See Section 5.4, “Logical Models”), users can rely on the Master view called Namespaces view.

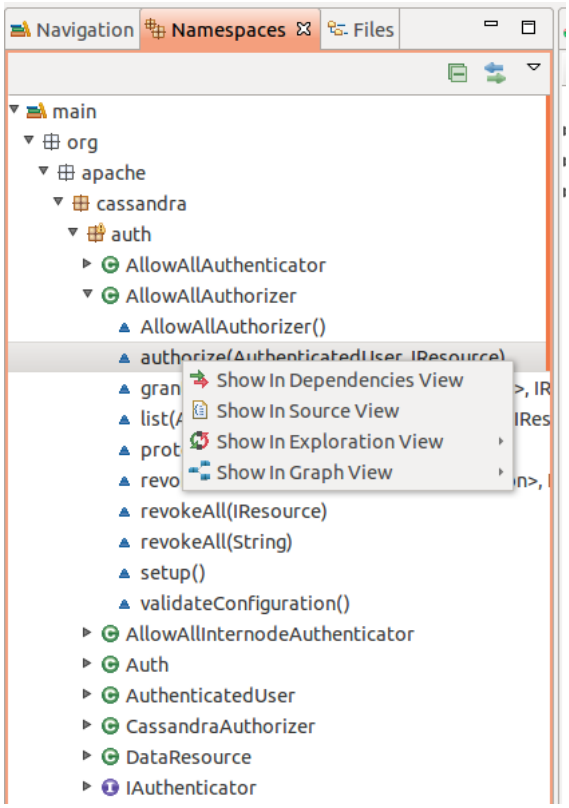


Figure 8.7. Namespaces View

As shown in figure “Namespaces View” , the logical elements that appear in Namespaces view also offer interactions for exploration and source code visualization when it is the case.

This single view provides access to both system-based and module-based logical models. To choose which logical model you want to see, use the view menu:

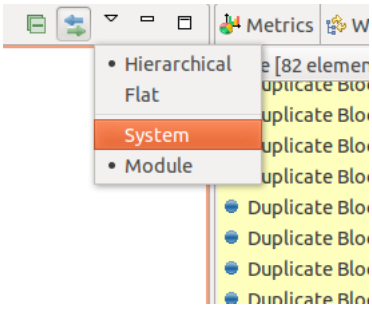


Figure 8.8. Logical Model Selection

Besides choosing which logical model to see, the Namespaces view also offers the possibility to change the *logical namespaces* presentation from flat to hierarchical and vice versa.

8.7. Managing the System Files

The Files view represents the structure of the files that make up the current *software system* .

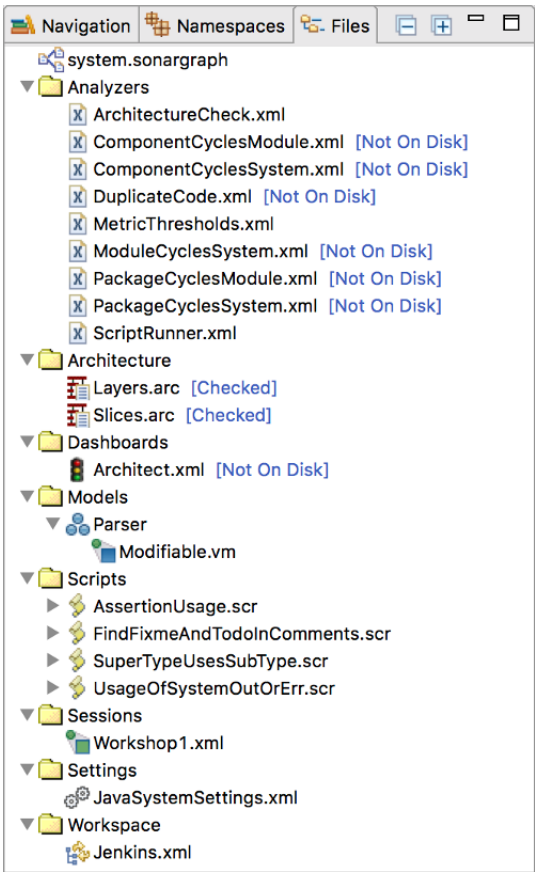


Figure 8.9. Files View

Those files are:

- **System File:** Named as "system.sonargraph", represents the current *software system*.
- **Analyzers:** Contains available configuration files of analyzers. A double click opens the corresponding configuration page. Alternatively the configuration pages are reachable via "System" → "Configure...".
- **Architectural Views:** Contains architectural view models.
- **Architecture:** Contains architecture files. A new architecture file can be created using the context menu of the "Architecture" folder. Existing architecture files can be added/removed from the architecture check also via their context menu.
- **Dashboards:** Currently, the file underneath is not modifiable and the content shown in the System view is fixed. In the future, the content displayed in the System view will be configurable.
- **Models:** Contains *virtual models* of the current *software system* (see Section 9.1, “Using Virtual Models for Resolutions”). These files only get modified when altering the set of resolutions and/or refactorings.
- **Plugins:** Contains plugin configuration files.
- **Scripts:** Contains scripts that can be executed for the current *system*. Those scripts have been added by either using a quality model (see Section 6.4, “Quality Model”) or they have been created manually (see Chapter 16, *Extending the Static Analysis*).
- **Settings:** Contains language specific settings.

- **Workspace:** Contains workspace profiles.

The files presented in the Files view get a star symbol (*) when they are modified as explained in Section 8.2.1, “Special Graphic Elements Decorations”

8.8. Managing the Workspace

The Workspace is a key concept in order to be able to set up and manage correctly a *Sonargraph software system* . Depending on the workspace definition, *Sonargraph* will be able to detect the source files (and class files when applicable) that will be used as input for the parsing process and generation of the domain models.

8.8.1. Definition of Filters, Modules and Root Directories

The Sonargraph workspace consists of the following elements that can be managed via the Workspace view:

- *File Filter*: Can be used to completely exclude files from being added to the model. The matching is based on the relative path of Sonargraph input files. This works in contrast to the Production Code Filter which includes the elements in the model, but explicitly marks them as excluded. As an example lets assume you do not want "dontLookAtMe.cpp" to be parsed and added to the model. That can be achieved by adding the following exclude pattern: "**/dontLookAtMe.cpp". The filter matches against the value of the property 'Identifying Path' shown in the Properties view for the selected element.

- *Production Code Filter*: Is used in order to exclude test code from the analysis. The filter is *component* based and processes all internal components. External components for which all incoming dependencies come from excluded internal components are also marked as excluded. Outgoing dependencies from non-excluded internal components to internal excluded components are marked with the issue 'Dependency to Excluded Internal Component'. This might indicate a problem since non-test code should not reference test code. The filter matches against the value of the property 'Workspace Filter Name' shown in the Properties view for the selected element.

- *Issue Filter*: Is used in order to exclude portions of the code to no longer generate analysis issues (e.g. cycles, threshold violations, duplicate code block issues, ...). This is useful in case you have legacy or generated code that you are not able to adapt or don't want to adapt. The filter is also *component* based and processes all internal non-excluded components. The filter matches against the value of the property 'Workspace Filter Name' shown in the Properties view for the selected element.

  NOTE: Parser issues and architecture violations cannot be filtered.

- *Module*: Is the top-most element and the root container for all the user-defined elements of a *Sonargraph software system* . Modules are equivalent to Eclipse projects, Maven modules, Visual Studio projects or IntelliJ projects and they contain at least one Root Directory Path.

- *Root Directory Path*: Corresponds to a location on the user's file system and is the top-most directory where the search for source files (and class files when applicable) will take place.

- *External*: Is the root container for elements that are used from within the user code but do not belong to any of the modules of the *software system* .

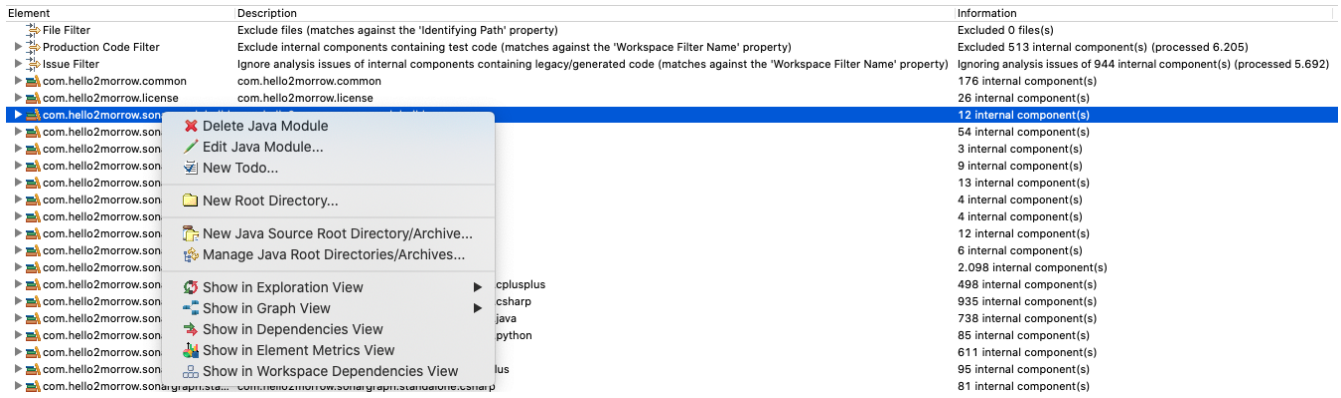


Figure 8.10. Workspace View

What you need to know about the Workspace Filter: The workspace filter works best when used with source file based analysis (i.e. C,C++ and C#). When analyzing Java .class and .java files are parsed. So special care must be taken to exclude .java files and the corresponding .class files. Since using the Production Code Filter has the added benefit of detecting unwanted dependencies from test to production code you should prefer the usage of the Production Code Filter.

What you need to know about the Production Code and Issue Filter:

- The filters use the 'Workspace Filter Name' of the components to produce matches with include and exclude patterns. The 'Workspace Filter Name' can be found in the *Properties* view when selecting a component. The name has the following structure: [Module]/[Root Directory]/[Physical Path]/[Component name without extension]. The name 'Events/src/com/app/events/Event' would refer to the component 'Event' in the directory 'com/app/events' in the root directory 'src' in the module 'Events'.

- The patterns support the following wildcards: ?=any character, *=any sequence between dots or slashes, **=any sequence. Both filters have a built-in '**' include pattern. So it might be enough to add exclude patterns. If needed you can define your own include patterns (disabling the built-in one). The include pattern(s) define which components pass and the exclude pattern(s) subtract from that set.

- With the Search dialog you can check which programming elements have been excluded and which ones are ignoring issues. To bring it up select "Edit" → "Search..." .

As seen in the previous image, the Workspace view offers interactions to create, edit and delete the workspace elements if necessary.

8.8.2. Managing Module Dependencies

Dependencies between modules that have either been defined manually or that have been generated automatically during the import/synchronization with external project files are visualized in the Workspace Dependencies View. It is important to note that if the Workspace Dependencies were calculated by the *software system* as a result of the synchronization process, it is not possible to modify them, nor delete or add more dependencies.

When Workspace Dependencies can be added manually, a dependency between modules might be created by pressing SHIFT and the primary modifier key of the platform (CTRL on Windows/Linux and CMD on Mac) and dragging a line with pressed left mouse button from source to target module.

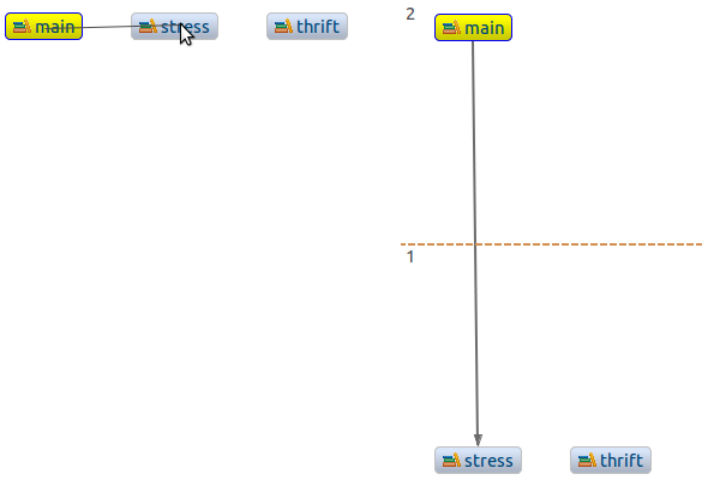


Figure 8.11. Defining a Manual Workspace Dependency

Similarly, if Workspace Dependencies are manually defined, they can also be deleted via context-menu or double-click interactions.

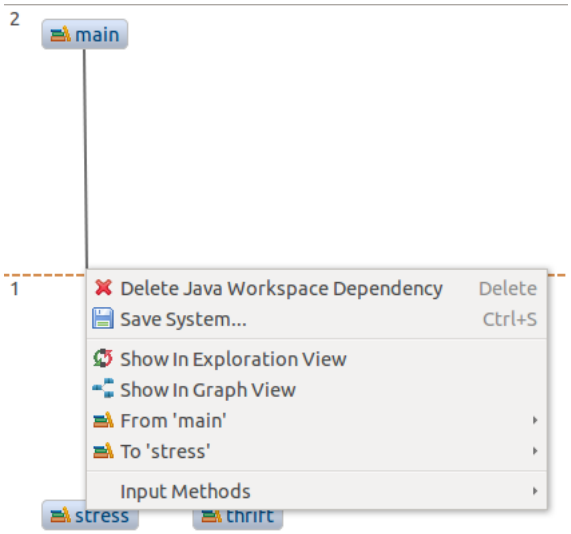


Figure 8.12. Defining a Manual Workspace Dependency

8.8.3. Creating Workspace Profiles for Build Environments

Workspace profiles help to solve the following problem for Java systems: If a workspace has been set up using for example an Eclipse workspace import, these root directories likely do not exist on the build server but only on a developer's machine. (Integration of Sonargraph-Build on the build server is described in more detail in the user manual of Sonargraph-Build.) In order to run the same checks with *Sonargraph* on the build server, a workspace profile defines transformation of root directories. Currently this applies only to Java class root directories. The transformation is done using an arbitrary number of profile patterns that consist of regular matchers and replacement expressions. The profile name can then be applied in the Sonargraph-Build configuration.

Each profile pattern consists of three parts:

1. **Module name matcher**: Regular expression matching module names. Only if this pattern matches, the module's root path will be applicable for transformation by this profile pattern.

2. **Root path matcher**: Regular expression matching against the identifying path of roots of the matched module.

3. **Root path replacement**: This pattern defines the new path that will be used to create a new root directory for this module and replace the existing path. Capturing groups that are used in the module name and root path matchers are accessible.

The two matchers are logically combined, so the capturing groups' indices of the root path matchers do not necessarily start at 1, but depend on the number of capturing groups in the module name matcher. Alternatively, named capturing groups can be used, as illustrated in the following example:

Let's assume that every module of the system has the same layout and has a root path with the identifying path "./target/classes", but you need to map that path to "./target/<module_name>-0.0.1-SNAPSHOT.jar". The three parts that make up the profile pattern can be defined as shown in the following screenshot. A detailed explanation is given below the screenshot.

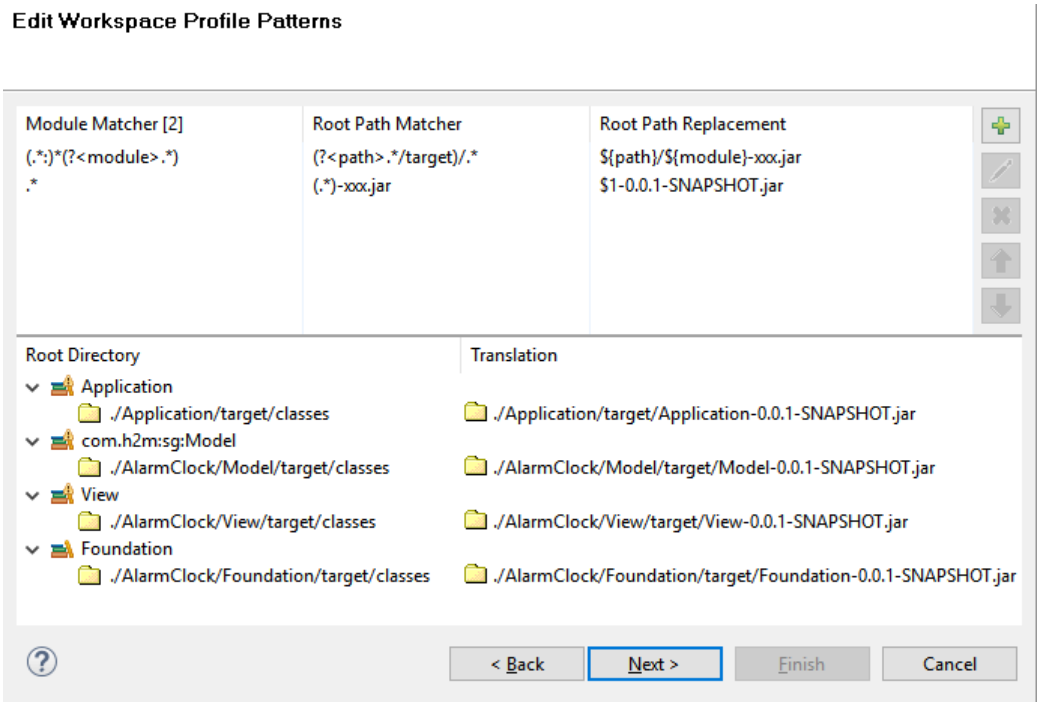


Figure 8.13. Creating Workspace Profile

1. **Module name matcher**: "(.*:)*(?<module>.*)" - This regular expression matches all module names and keeps the module name in a named capturing group "module" that allows re-using the module's name for the JAR file. If the module has been created by a Maven import and the name matches the schema groupId:artifactId, the groupId will be omitted by the first optional capturing group.

2. **Root path matcher**: "(?<path>.*/target)/.*" - This regular expression matches against the identifying path of roots of the matched module. The part that needs to be re-used is made available via another named capturing group "path".

3. **Root path replacement**: "${path}/${module}-xxx.jar" - This pattern defines the new root path that replaces the match. The named capturing groups are used to insert the part of the original path that needs to be re-used and also the module name.

The next pattern replaces the "xxx" string with the correct version using a "standard" unnamed capturing group. It is a matter of taste if you want to split the transformation into several profile patterns or do it in one step.

NOTE

All root directories must be mapped! If profile patterns result in the same mapping for different root directories of the same module, only one directory will be created. Otherwise the same rules apply as for the standard software system workspace: It is not possible that the same root directory is used by different modules.

TIP

More info about the regular expression capabilities can be found in the JavaDoc of *java.util.regex.Pattern* and its section about *capturing groups*.

Related topics:

- Chapter 19, *Build Server Integration*

8.9. Analyzer Execution Level

Analyzer execution levels have been introduced in *Sonargraph* 9.6.0. Depending on the currently set level not all analyzers are executed. Depending on the goal of a session this results in a smoother user interaction. The user can select one of four levels (Full, Advanced, Basic, Minimal) by System → Analyzer Execution Level .

The list of Analyzers to be run for each level depends on licensed features and languages, and can be shown by System → Analyzer Execution Level → Description... .

Note

This setting is stored between sessions.

Note

If the level is not set to 'Full' not all possible issues and metrics are available.

The Analyzers, their Analyzer Execution Level, and their current state are show in Analyzers View.

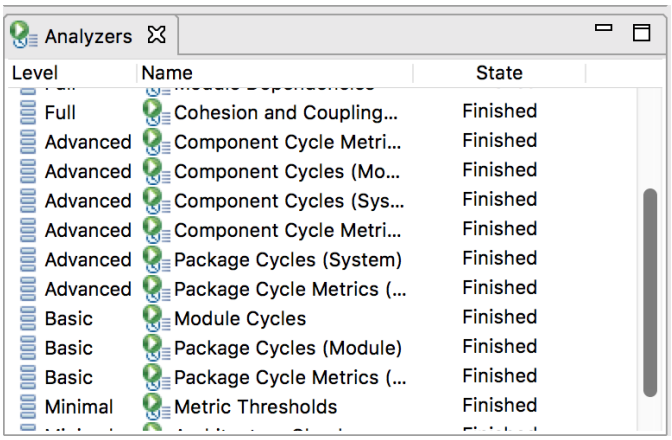


Figure 8.14. Analyzers View

Related topics:

- The Analyzer Execution Level may also be set in *Sonargraph* Eclipse Plugin Section 20.1.4, “Setting Analyzer Execution Level” .
- The Analyzer Execution Level may also be set in *Sonargraph* IntelliJ Plugin Section 20.2.1, “Assigning a System” .

8.10. Analyzing Cycles

The cycles analysis capability of *Sonargraph* is leveraged by the Cycle Groups, Cycle and Exploration views. The first one is used to list cycle groups and the *components* involved in those and the other two allow to inspect in detail those cycle groups.

8.10.1. Revising Cycle Groups

Cycle Groups are containers for elements participating in a cycle. This view gathers those cycle groups found during system analysis by the Cycle analyzer and groups them according to scope (system or module) and element level (e.g. component, namespace, package, ...).

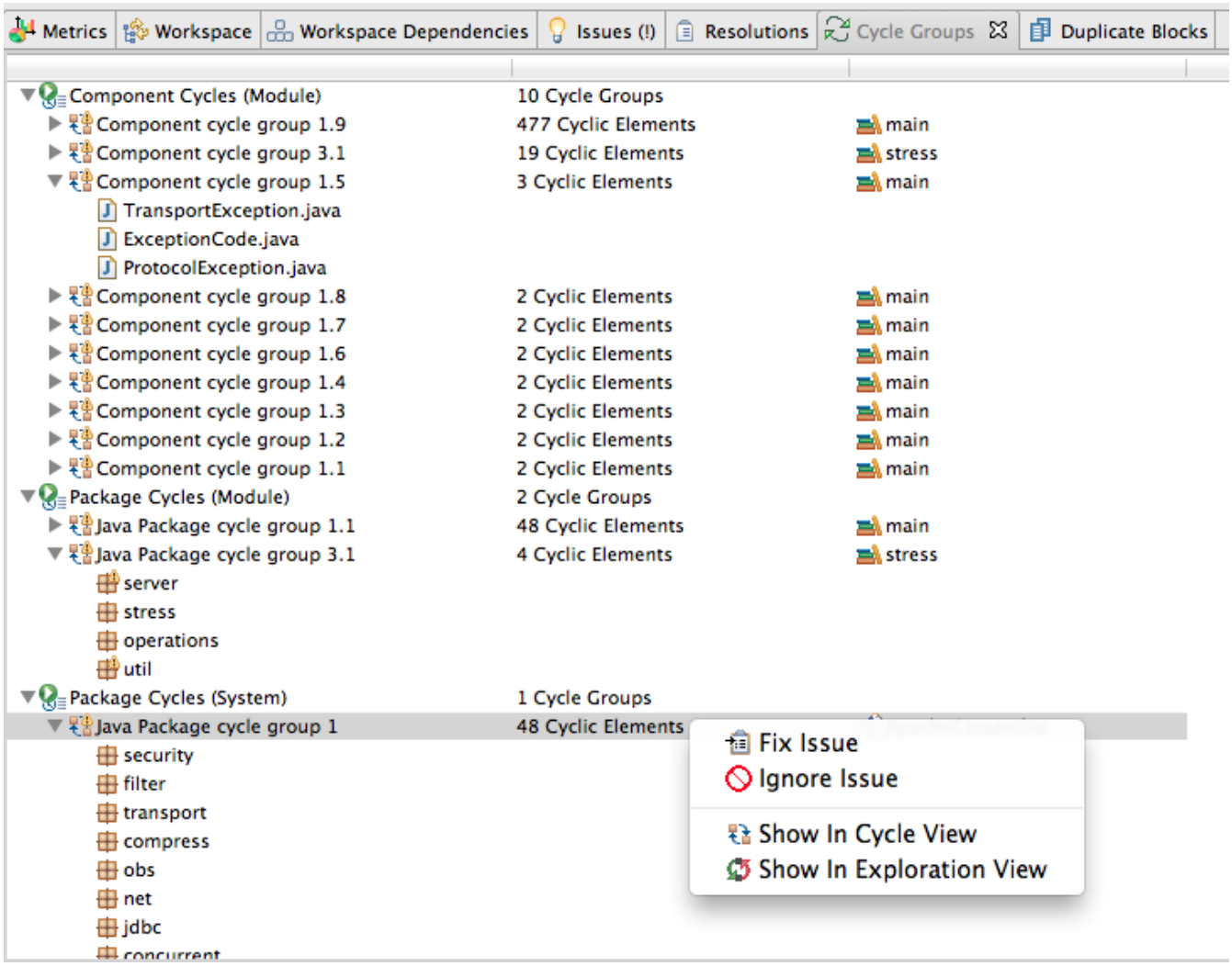


Figure 8.15. Cycle Groups View

The first column references, in different tree based levels, the category, a unique cycle group identifier and the elements participating in the cycle. The second column informs about the number of elements involved in the cycle group and finally, the third column shows the corresponding module name (for module based cycles) of the cycle or the system name.

Sonargraph considers cycles as issues, as they greatly contribute to the structural erosion of the code base. Thus, using right-click on a cycle group you can define resolutions in order for it to be addressed by the team (see Chapter 9, *Handling Detected Issues*).

The context menu for a cycle group also offers options to visualize the cyclic elements in specialized views such as the Exploration view (see Section 8.11, “Exploring the System”) and the Cycle View (see Section 8.10.2, “Inspecting Cyclic Elements”).

8.10.2. Inspecting Cyclic Elements

When choosing to examine a cycle group in the Cycle view, it allows to inspect in detail the dependencies between the cyclic elements belonging to the selected group. Selecting a node or a dependency allows to use the support of the Parser Dependencies (In and Out) auxiliary views to point out to where in the code base the associated dependencies are being generated. Context menu is also enabled for nodes and dependencies.

The nodes are colored depending on their parent. The parent coloring source can be changed via the toolbar and information about the different parents is also provided by clicking on the palette-icon in the toolbar. Generally, the more colors are shown, the more entangled is your structure.

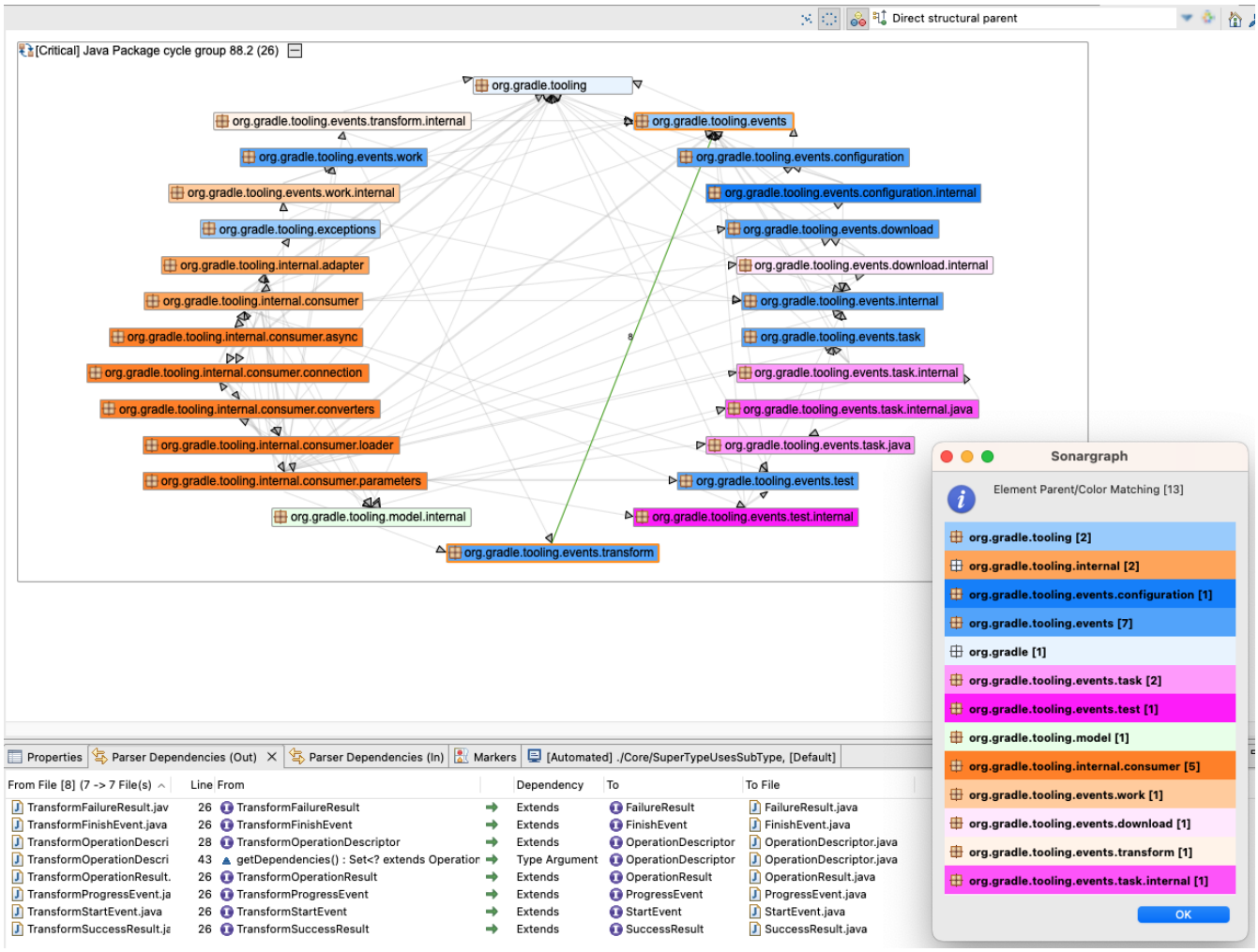


Figure 8.16. Cycle View

Via context menu the cycle group can also be displayed in the Exploration view as shown in the following screenshot. This has the advantage that not only the elements participating in the cycle group are visible, but also their parents and the children that are the end points of the involved dependencies.

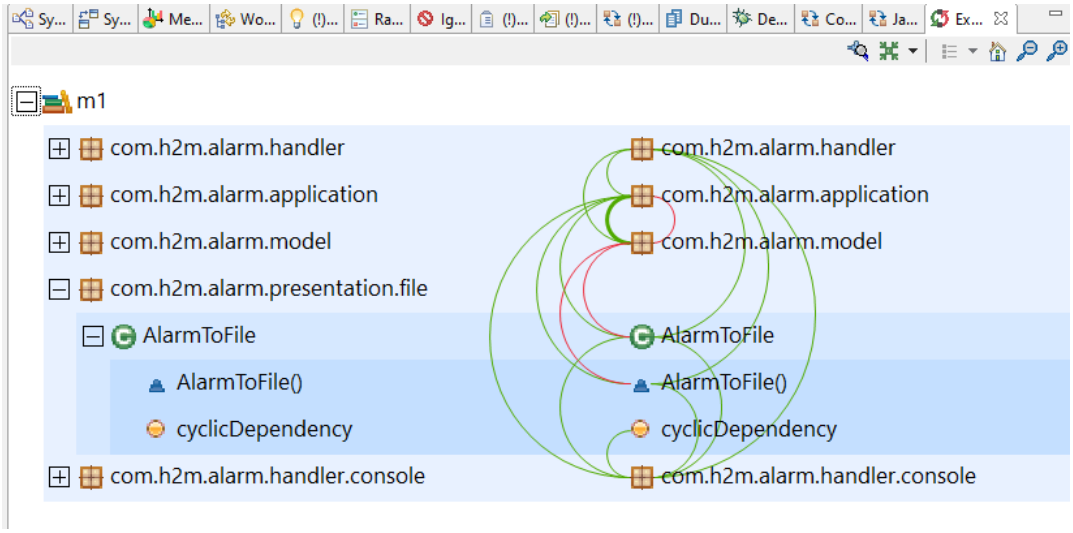


Figure 8.17. Cycle Group shown in Exploration View

8.10.3. Breaking Up Cycles

The Cycle Breakup view can be opened via the context menu of the Cycle view. The context menu must be requested without a selection (i.e. right click on the background).

Pressing "Compute" calculates a breakup set of edges to completely remove the given cycle. The algorithm used was presented in 'Combinatorial Algorithms for Feedback Problems in Directed Graphs' written by Camil Demetrescu and Irene Finocchi. The authors summarize the algorithm as follows:

Roughly speaking, our algorithm tries to find a compromise between two (somewhat opposite) approaches, i.e., removing light arcs, that is, arcs with small weight, and removing arcs belonging to a large number of cycles. Indeed, light arcs are convenient to be deleted as they contribute to breaking cycles, yet increasing the weight of the feedback set only to a limited extent. On the other hand, if a heavy arc belongs to a large number of cycles, it may be convenient to choose it instead of a numerous set of light arcs.

The "Breakup" table shows edges from top to bottom representing the removal order and the effect on cyclicity and number of cyclic nodes.

Dragging edges to the "Remove" table instructs the algorithm that these should be explicitly removed without considering the number of parser dependencies. Drag them back to the "Breakup" table to remove this configuration. Edges to be explicitly removed can also be dragged from the corresponding cycle view into the "Remove" table. If more edges are contained in the "Remove" table than are necessary to break up the cycle, the breakup set is over-defined. Those unnecessary edges are indicated by a grey background of the 'from' and 'to' element.

Pressing "Remove Violations" moves all violating edges to the "Remove" table.

Dragging edges to the "Keep If Possible" table instructs the algorithm that these should be kept if possible. If no more edges are left to remove even those that should be kept are considered. Drag them back to the "Breakup" table to remove this configuration. If the algorithm needs to consider edges as removal candidates that should be kept, the edges are analyzed from bottom to top (i.e. the topmost edge is the last to be considered). This order can be changed in the "Keep If Possible" table by dragging edges up or down. Edges that should be kept but need to be removed are highlighted in both tables with a yellow background of the 'from' and 'to' element.

Changing the set of edges to be explicitly removed or kept requires a re-computation. This is indicated by a grey background color in the "Breakup" table and an (again) enabled "Compute" button.

If the "Breakup" table contains the correct edges that should be removed, a "delete" refactoring can be defined by multi-selecting the entries and open the context menu via right mouse click. For details, see Section 10.1, “Creating Delete Refactorings”

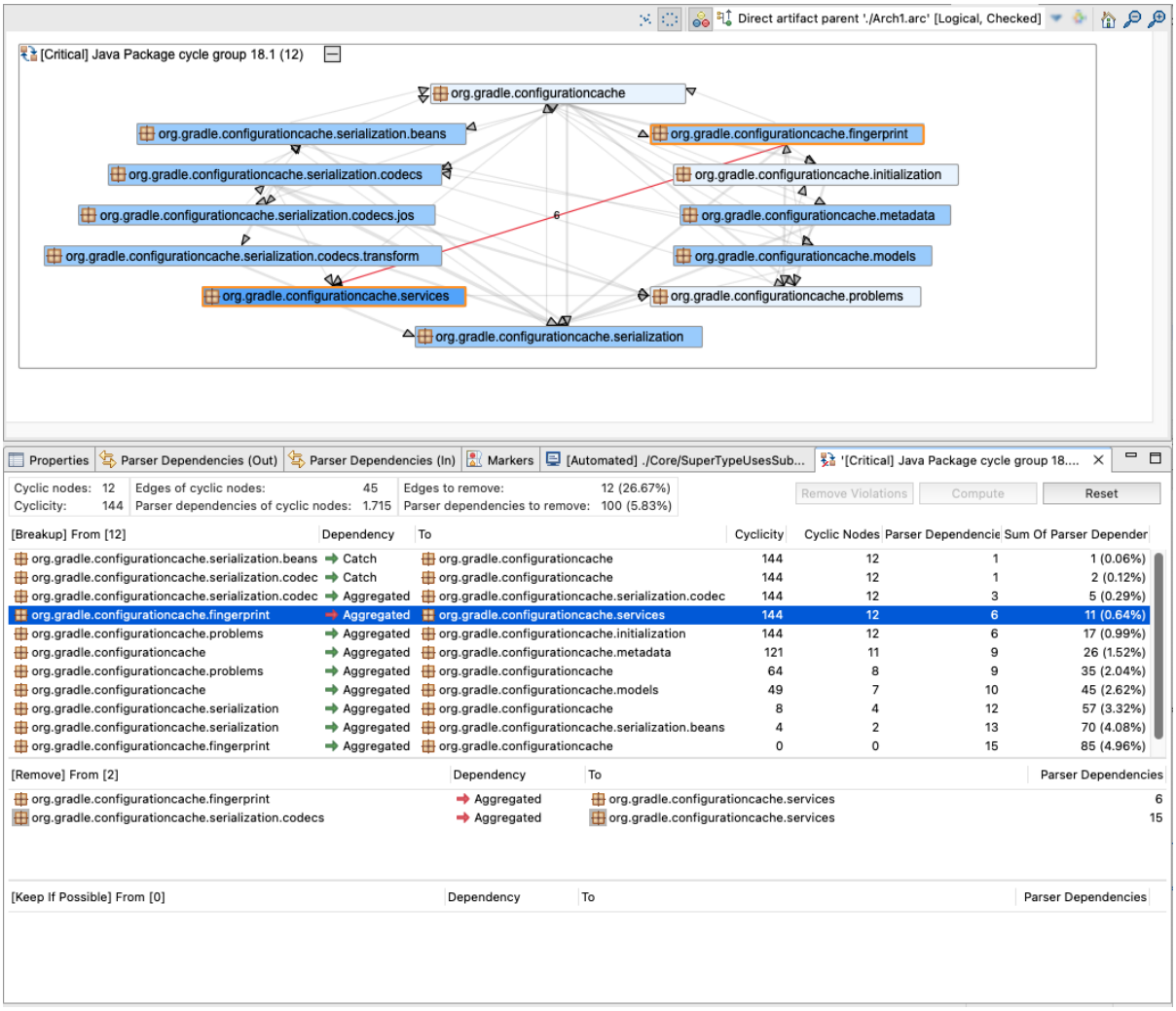


Figure 8.18. Cycle Breakup View

Highlighting Added Cyclic Elements

If a baseline has been created and activated (see Chapter 14, *Examining Changes*) added cyclic elements are highlighted in the Cycle view with a blue plus sign as shown in the screenshot below. These elements and their incoming/outgoing dependencies are usually a good starting point for refactorings.

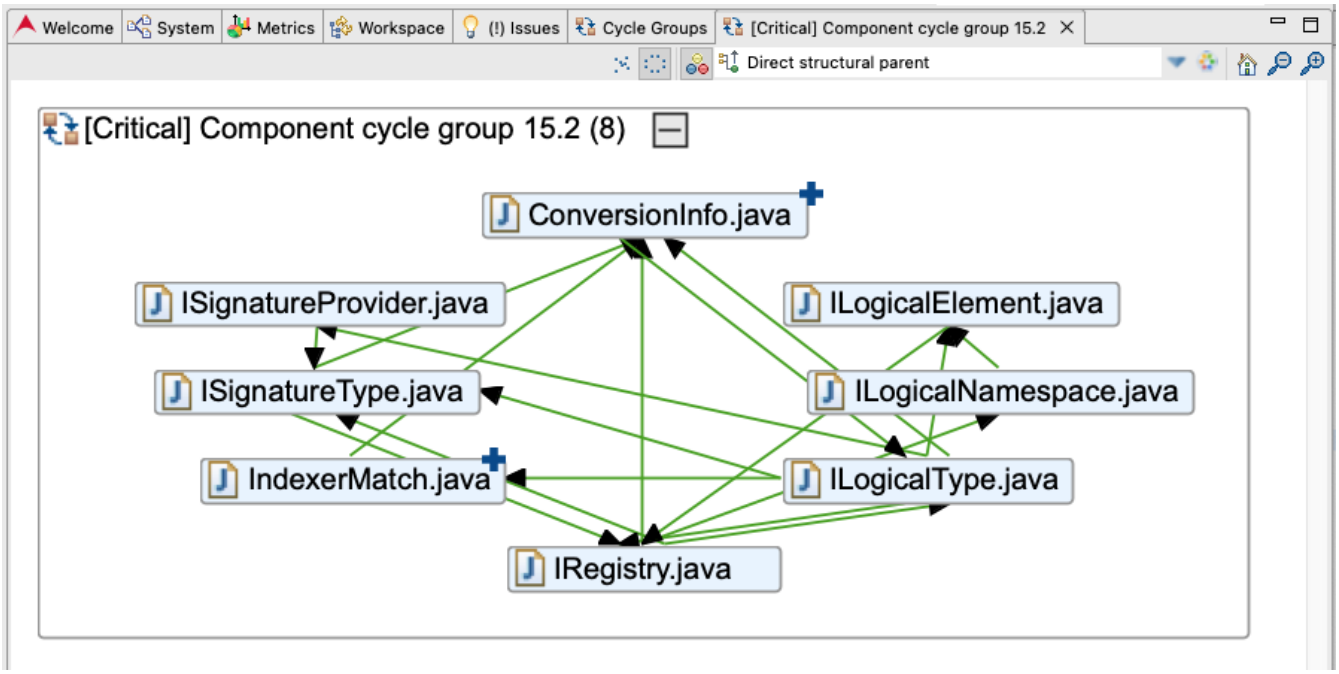


Figure 8.19. Highlighted Added Cyclic Elements

8.11. Exploring the System

Sonargraph offers a set of views such as Exploration, Graph and Dependencies views to allow users to explore the dependencies between elements. The following sections describe these views in detail and how to use them efficiently.

8.11.1. Exploration View

The Exploration view shows a tree-like structure of the *software system*. The elements contained in the software system are represented as nodes that can be expanded/collapsed if they contain other elements. Dependencies between the elements are represented as arcs, that have to be read counter clockwise. Left-hand side dependencies are downward and right-hand side dependencies are upward. The number of upward dependencies is minimized to show the construction order of the software system.

The Exploration view performs ad-hoc level and cycle analysis for elements underneath the same parent, shows cyclic elements with a red background and visualizes the levels by showing horizontal grey lines.

The Exploration view aims to show a very compact presentation even for very big software systems. If there is limited space vertically and a lot of interconnected nodes to be shown the arcs are squeezed to occupy less space.

The Exploration view offers focus operations to reduce the amount of visible elements/dependencies allowing the user to "focus" on specific aspects.

The Exploration view supports forward/backward navigation recalling different visibility states via the main tool bar's yellow back and forth arrows (). Use that mechanism if you want to go backward or forward to a specific state of the Exploration view. The 'Home' tool bar button () resets the Exploration view to it's initial state and clears all navigation states. The different visibility states include focus, expand/collapse and selection state of the view.

Structure Modes

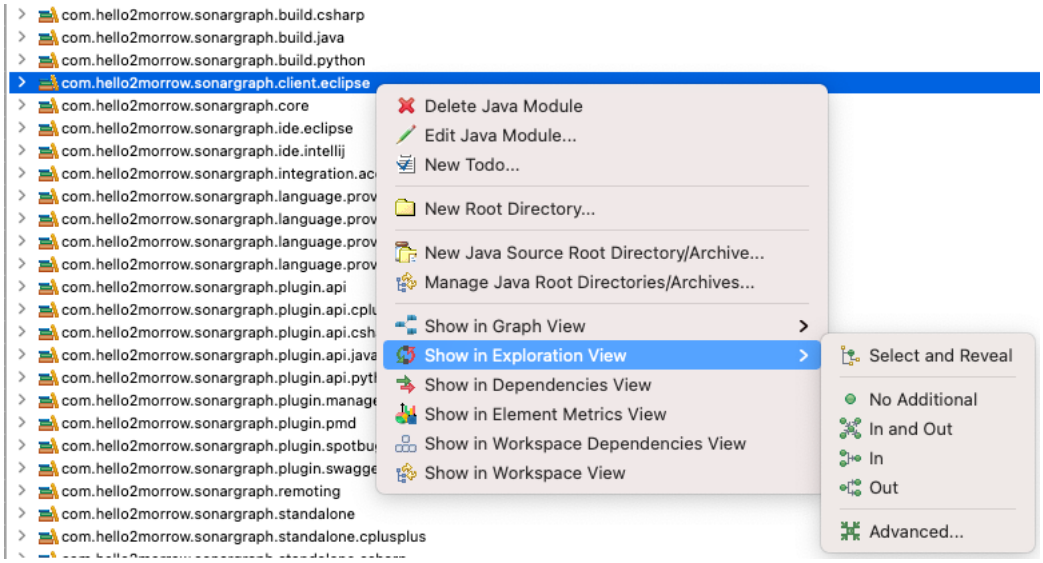
The Exploration view supports 4 different *structure modes*:

- *Physical*: Shows the physical structure as shown in the Navigation view including root directories.
- *Physical w/o Root Directories*: Shows the physical structure as shown in the Navigation view excluding root directories for a more compact presentation.
- *Logical System Scope*: Shows the logical structure as shown in the Namespaces view using system scope.
- *Logical Module Scope*: Shows the logical structure as shown in the Namespaces view using module scope.

Creating an Exploration View

Once you have a successfully parsed the software system the 2 most straightforward ways to create an Exploration view are:

- Use the quick access tool item in the main tool bar showing the Exploration view icon: The quick access tool item offers a drop-down box where you can choose 1 of the 4 structure modes.
- Select any number of elements in the Navigation or Namespaces view and use the context menu entry 'Show In Exploration View'.



- Select and Reveal : Create an Exploration view without focus, selecting the input elements and revealing the first input element.
- No Additional : Focus only the input elements. Only the input elements and the dependencies between them are going to be part of the displayed content.
- In and Out : Focus the input elements and the ones directly connected with incoming/outgoing dependencies. The input elements plus the ones which they depend on and the ones that directly depend on them will be part of the displayed content along with all the dependencies involved.
- In : Focus the input elements and the ones directly connected with incoming dependencies. The selected elements plus the ones that directly depend on them will be part of the displayed content along with the dependencies between all of them.
- Out : Focus the input elements and the ones directly connected with outgoing dependencies. The selected elements plus the ones which they depend on will be part of the displayed content along with the dependencies between all of them.
- Advanced... : This will open an advanced Exploration view creation dialog.

In general an Exploration view can be created via the context menu by selecting different elements throughout the user interface, some of them are:

- A checked architecture DSL file creating an Exploration view including the corresponding artifacts.
- A cycle group creating an Exploration view including only the corresponding dependencies (i.e. using a focus on the dependencies).
- Elements or dependencies from script result previews.
- Issues.

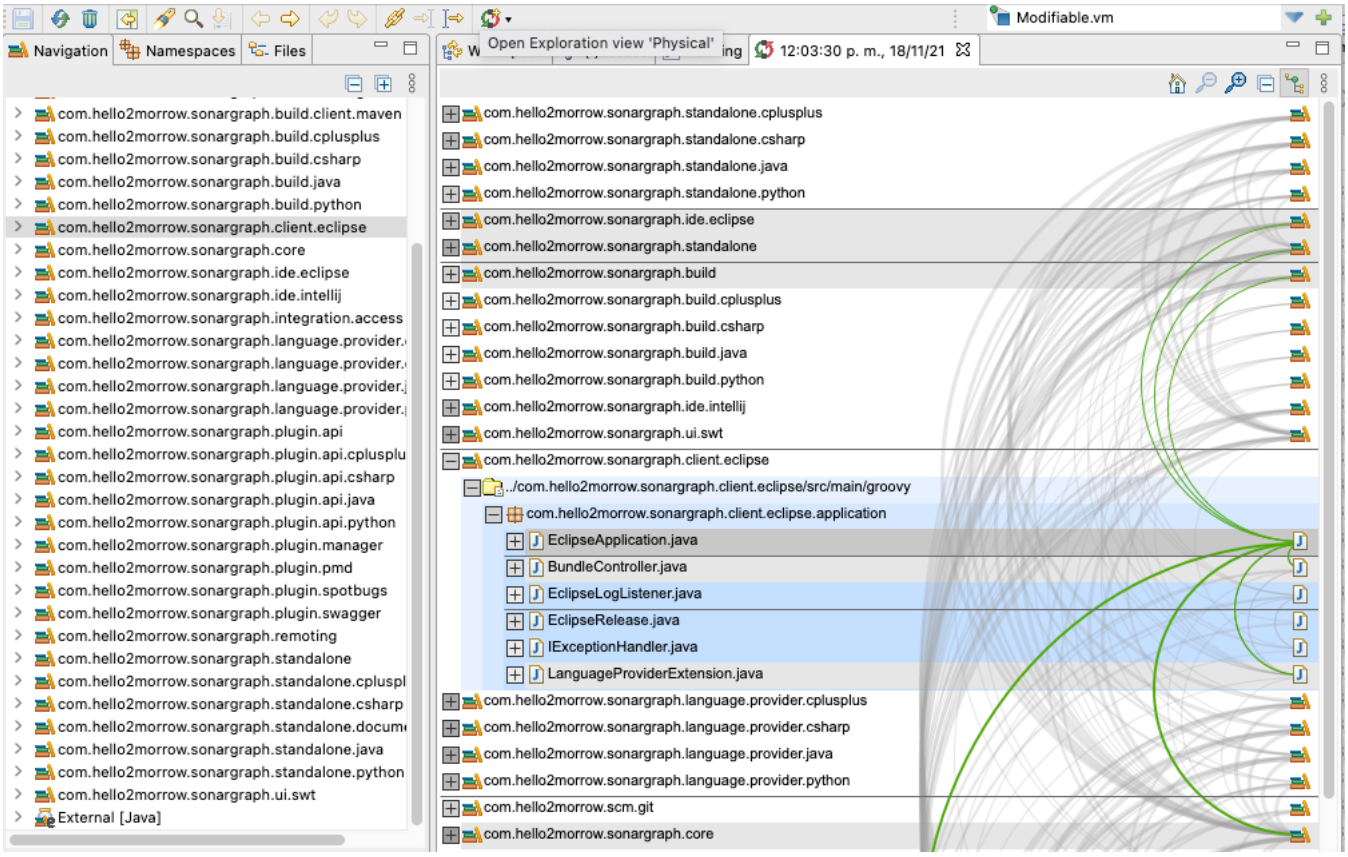


Figure 8.20. Example of Created Exploration View without Focus

Questions that can be answered using the Exploration view are:

- What are the dependencies between some layers, subsystems, packages or types?
- What is the reason for some unexpected dependency?
- How is it possible to decouple a given pair of package trees?
- Where is package X or file Y located in the package tree?

The exploration view provides a number of means supporting you to:

- Get an overview of your *software system* on a high abstraction level.
- Drill-down to answer specific questions.
- Zoom in and out of the Sonargraph model tree by expanding and collapsing element nodes.
- Use the focus operations so that irrelevant information is hidden.

8.11.1.1. Presentation Modes, Levelization, Semantics of Icons and Decorators

The Exploration view offers 3 presentation modes that affect the presentation of recursive elements (e.g. package, namespace, directory):

- Mixed: Empty elements without siblings are compacted.
- Hierarchical: All elements are shown.

- Flat: Only the elements containing elements of other types are shown.

The Exploration view uses the following strategies to visualize different aspects:

- The Exploration view shows some horizontal grey lines. These are the level lines. Non-cyclic elements on the same level are not depending on each other. Cyclic elements of a cycle group are on the same level. Those elements have a red shade rectangle background. Different red shades are used for different cycle groups (relative to their parent) as well as a cycle index property is shown in the Properties view. Nodes belonging to the same cycle group show the same cycle group index. To differentiate non-cyclic and cyclic elements on the same level an additional red line is shown dividing the level.
- If an element has children, it will show a collapse/expand figure (+/-). If the background of this figure is darker it has more children.
- If not all child elements of an element are shown due to a focus the element is marked with a grey triangle. If not all outgoing dependencies of an element are shown the element is marked with a light grey triangle. If both is true the triangle will show one half in grey and one half in light grey.
- Selected elements are highlighted with a grey background. Dependent and using elements have a light grey background.
- Different blue shaded boxes are used to better distinguish the nesting structure.

The following Exploration view shows some of the mentioned visual elements. The 'controller' package has been selected and focused with 'In and Out' dependencies:

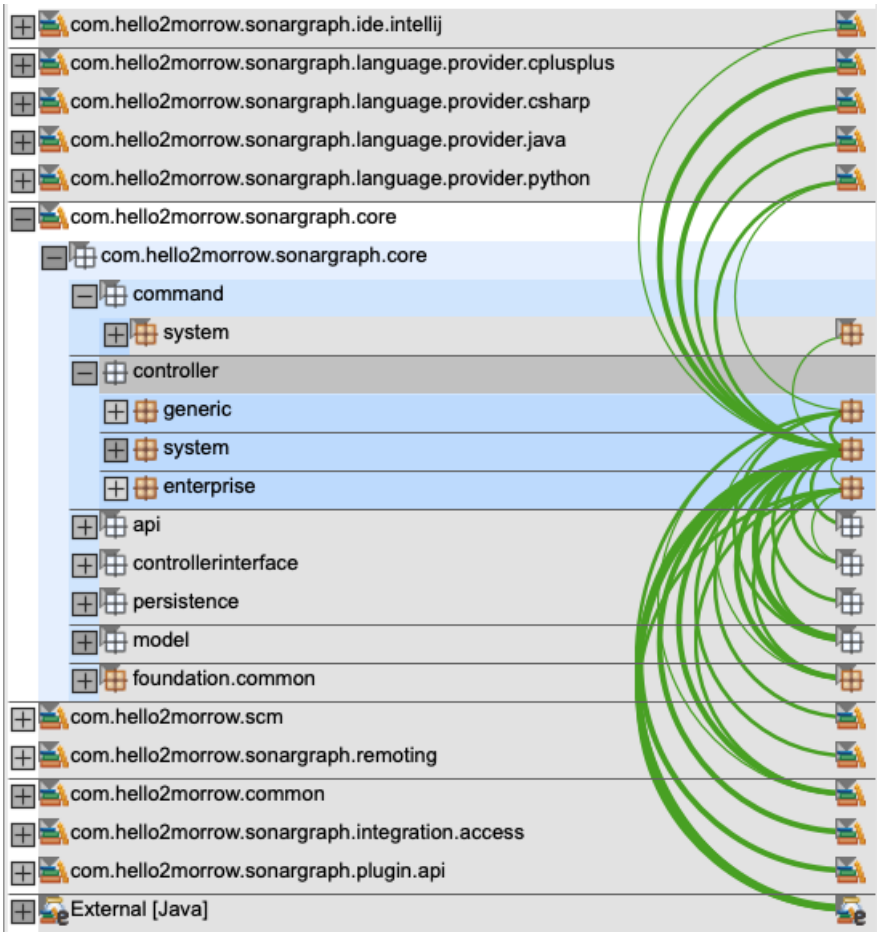


Figure 8.21. Exploration View with Applied Focus

Levelization

In general levelization is applied to all non artifacts shown in an Exploration view. When an Exploration view can potentially show artifacts because it has been created from a checked architecture aspect an additional view option widget is shown allowing the used to select 1 of 2 levelization modes:

- Non Artifacts Only: only non artifacts are levelized and artifacts are shown in their definition order without highlighting potential cycles involving artifacts.
- All: All elements are levelized, therefore levelization is applied also to artifacts, highlighting also potential cycles involving artifacts.

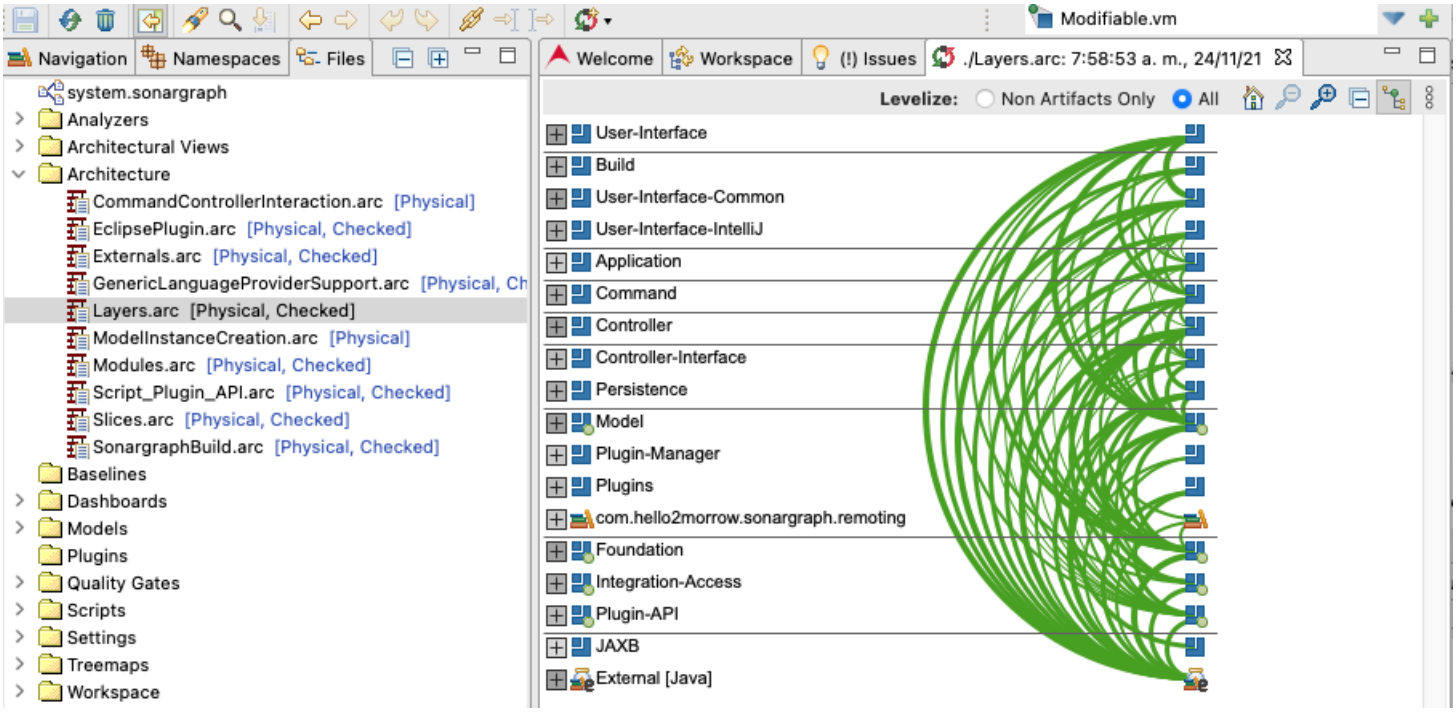


Figure 8.22. Exploration View created from Architecture Aspect showing the Levelization Mode Widget

8.11.1.2. Focus

The primary goal of the focus is to reduce the amount of visible elements and dependencies so that specific aspects of the software system can be analyzed more easily. Several focus operations are available that can be applied to 1 or more elements or dependencies. To apply a focus operation simply select elements or dependencies and choose a focus operation from the context menu.

Remove From Focus

The simplest form of reducing the amount of visible elements/dependencies is to remove elements/dependencies from the view, that are currently not of interest. Select the elements/dependencies, right click to bring up the context menu and choose 'Remove From Focus'.

Set Focus

Another way of reducing the amount of visible elements/dependencies is to explicitly focus elements/dependencies. Select the elements/dependencies, right click to bring up the context menu and choose 'Set Focus'.

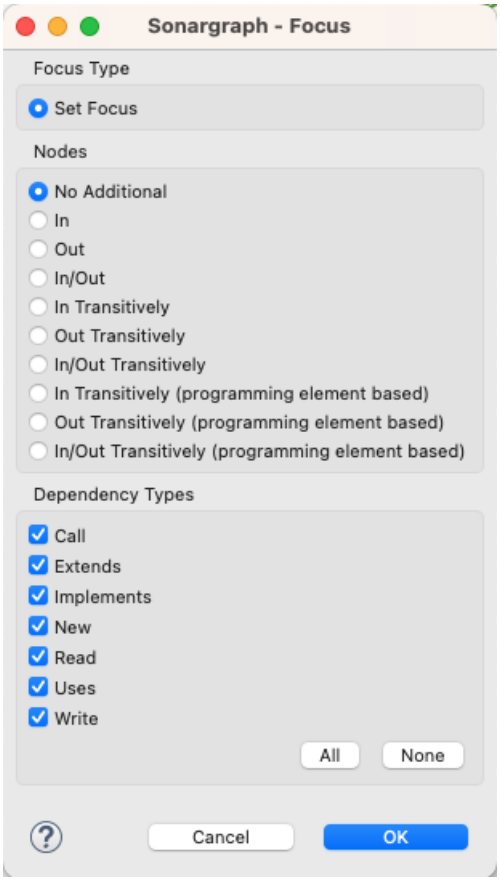


Figure 8.23. Focus Dialog

The transitively options also let you include indirectly connected nodes per incoming/outgoing dependencies.

Difference between 'Transitively' and 'Transitively (programming element based)':

Out Transitively: The selected node(s), their interdependencies and all nodes used by the selected node(s) transitively are considered. Leaf nodes (either components or logical top-level programming elements) are treated as an indivisible unit.

Out Transitively (programming element based): The selected node(s), their interdependencies and all nodes used by the selected node(s) transitively are considered. Leaf nodes (either components or logical top-level programming elements) are not treated as an indivisible unit. Only the connected programming elements are used.

So when components/logical top-level programming elements are treated as indivisible units all those "units" are collected (shown) when any programming element (method,field,type,...) including nested elements uses any programming element of another "unit". That means you can see the interconnections based on those "units". The other programming element based modes only show the interconnections of programming elements.

When you are planning to make changes to 1 (or more) component(s) (source and header files combined in C,C++, source files in all other languages) with 'In Transitively' you would see all affected other components being able to assess the impact of those changes, useful to know what needs to be re-tested or simply to know the magnitude of the impact.

With 'Out Transitively' you would see the affected (connected) components which could help to see what (physically) belongs together and decide how to structure due to architectural aspects (i.e. which components might belong to the same architecture artifact) or even decide which refactorings to apply to bring related things closer together.

'Transitively (programming element based)': Using that mode you normally you see less elements since only the connected programming elements are used. Combining those modes with different dependency types allows you to see inheritance based connections or call bases connections and so forth.

NOTE: If only dependencies are selected only the 'Dependency Types' are available and not the 'Nodes' section.

Add To Focus

Select elements/dependencies, right click to bring up the context menu and choose 'Add To Focus' to potentially extend the visible elements dependencies.

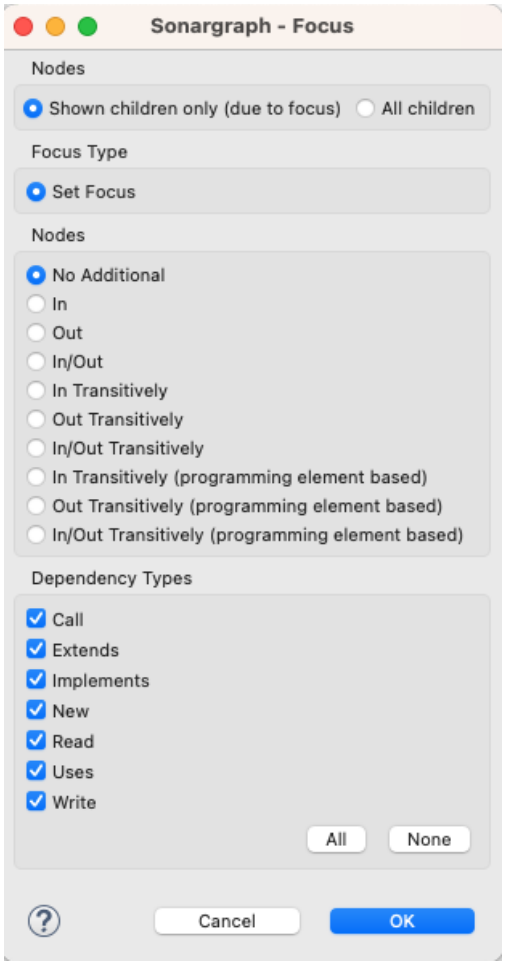


Figure 8.24. Add to Focus Dialog

When not all children of a selected element are currently visible the user can decide to include the not visible children as well by selecting the 'All Children' option.

Re-Adding Not Visible Elements

Once an element is removed from the view it is still possible the locate the element using the Find dialog (context menu 'Find Elements...') select it and use 'Reveal With Focus...'.

Clear Focus

When you have an applied focus, the context menu will show the 'Clear Focus' entry to clear the current focus.

8.11.1.3. Interaction with Auxiliary Views

The Exploration/Architectural view offers interaction with different Auxiliary views of *Sonargraph*.

Properties View

The Properties view shows additional information depending on the selection in the Exploration/Architectural view. There can either be no selection showing information of the Exploration/Architectural view itself about, focus and other properties or elements/dependencies can be selected showing information about size, issues, violations and so forth.

Parser Dependencies In/Out Views

Selecting elements/dependencies in the Exploration/Architectural view the Parser Dependencies views show the underlying parser dependencies. Using the context menu 'Show In Source' entry on a specific parser dependency allows the user to to jump into the Source view revealing the specific dependency.

Exploration/Architectural Metrics View

Selecting elements in the Exploration/Architectural view the corresponding metrics view re-calculates the metrics based on the current focus and selection.

For elements (modules, artifacts, packages/namespaces, components ...) the following metrics are calculated:

- The number of (non-distributed) elements that are currently shown (e.g. '5 Modules' means that 5 different modules are somehow included).
- The number of distributed elements (e.g. when 1 unique module is found '3 Modules (distributed)' means that the 1 module has been found 3 times in the model. This can happen when assigning elements from 1 module to 3 artifacts, resulting in 3 distributed modules based on 1 defined module).
- The number of elements that are marked as cyclic (sometimes 'distributed' - where applicable).

NOTE: Some elements cannot be distributed (e.g. artifacts, components, types, fields). If metrics are not specifically named xxx (external) they are only calculated for internal elements.

There are 4 dependency categories for which the number of parser dependencies, parser dependencies violating and parser dependencies violating ignored are aggregated:

- Outgoing dependencies: The dependencies of the selected elements that point to elements not contained in the selection.
- Incoming dependencies: The dependencies of the not selected elements that point to any selected element.
- Inner dependencies: The dependencies between the selected elements including their children.
- Downward dependencies: All upward dependencies connected to any selected element.
- Upward dependencies: All downward dependencies connected to any selected element.

NOTE: When no element is selected all (not excluded by focus) dependencies are counted as inner dependencies and all down/upward dependencies are counted.

8.11.2. Graph View

The graph-based system exploration allows users to take an arbitrary selection of elements and create a graph representation with nodes and edges to find out what their overall interaction looks like:

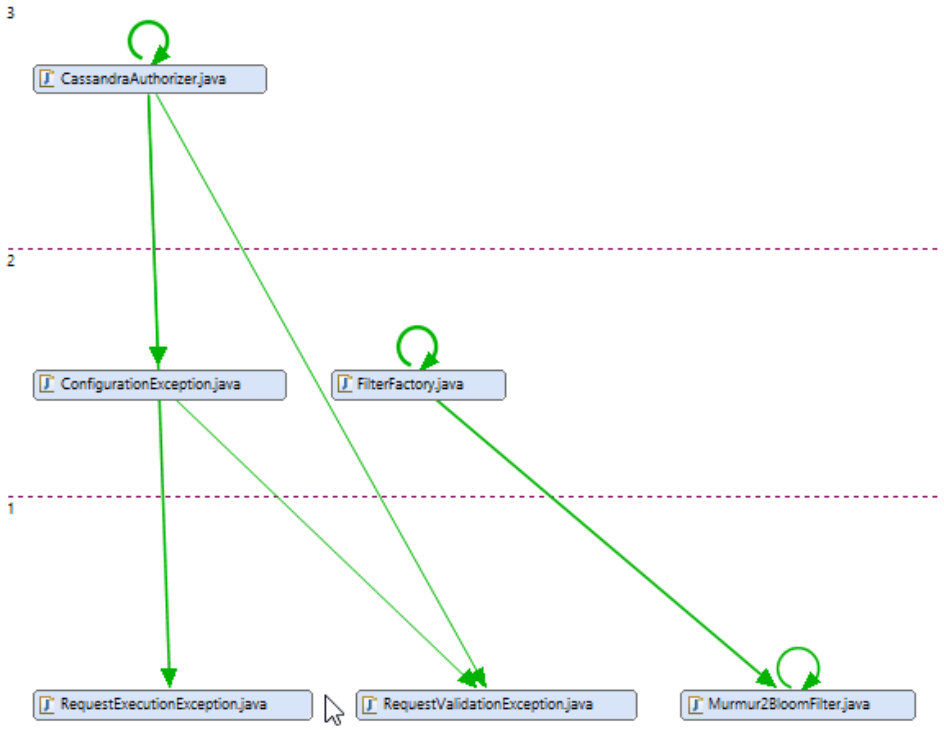


Figure 8.25. Graph View

By default, the graph perspective presents a levelized layout which comes handy to visualize the levels in which the software system elements are classified according to their dependencies to each other.

8.11.2.1. Focus

For Graph and Workspace dependencies view, the focus concept is the key element to understand how these views are created and how they can be modified to make a deeper analysis of the dependencies between elements. Any of these views (except for the Workspace Dependencies view when there are not workspace elements in the system) will have at least one element in focus and any number of elements not in focus.

- Elements in Focus: Elements that are the center of the analysis. Dependencies will be calculated for these elements according to the selected focus mode. When a Graph view for example is requested by the user for a set of selected elements, these elements will be the ones in focus.

- Elements not in Focus: Elements that appear as a result of the calculation of the dependencies for the elements in focus. They appear in the views because they are the endpoint of a dependency from or to an element in focus but they are not the active part of the dependency analysis.

Focus Modes

When creating a new Graph or Workspace Dependencies view or using the focus operation inside them, one of the following focus modes can be selected:

- No Additional Dependencies : Only the selected elements and the dependencies between them are going to be part of the displayed content.

- Incoming Dependencies : The selected elements plus the elements that directly depend on them will be part of the displayed content along with the dependencies between all of them.
- Outgoing Dependencies : The selected elements plus the elements which they depend on will be part of the displayed content along with the dependencies between all of them.
- Incoming and Outgoing Dependencies : The selected elements plus the elements which they depend on and the elements that directly depend on them will be part of the displayed content along with all the dependencies involved.

Transitive Dependencies

Sometimes it is required to analyze relationships between elements beyond the direct dependencies. For this reason, *Sonargraph* offers the option of taking into account the transitive dependencies for the focus operations. To better understand this concept assume a system with the following dependencies between elements A, B, C, D and X:

- Element A depends on element B: A -> B
- Element B depends on element X: B -> X
- Element X depends on element C: X -> C
- Element C depends on element D: C -> D

Taking element X as a reference for this example, we can express the relationship between A and X as A -> B -> X and the relationship between X and D as X -> C -> D. For these relationships the following statements are true:

- X has a transitive incoming dependency from A
- X has a direct incoming dependency from B
- X has a direct outgoing dependency to C
- X has a transitive outgoing dependency to D

Graph-Based Views Properties

Besides the input elements and the focus mode the Graph-based views need the following 3 properties to be created:

- Transitivity: Users must indicate whether they want to see transitive dependencies for the supplied input or not. See Section 8.11.2.1, “Transitive Dependencies”
- Only Internal: If selected, all elements under the External node will be excluded from the view.
- Dependency types: Users might want to focus the analysis on certain types of dependencies. The lower section in the dialog allows the selection of the dependency types that will be considered for view creation or focus operations.

Quick View Creation

Graph-based views can be created by right clicking on a selection of elements, selecting the view to open and providing one of the four focus modes from the context menu.

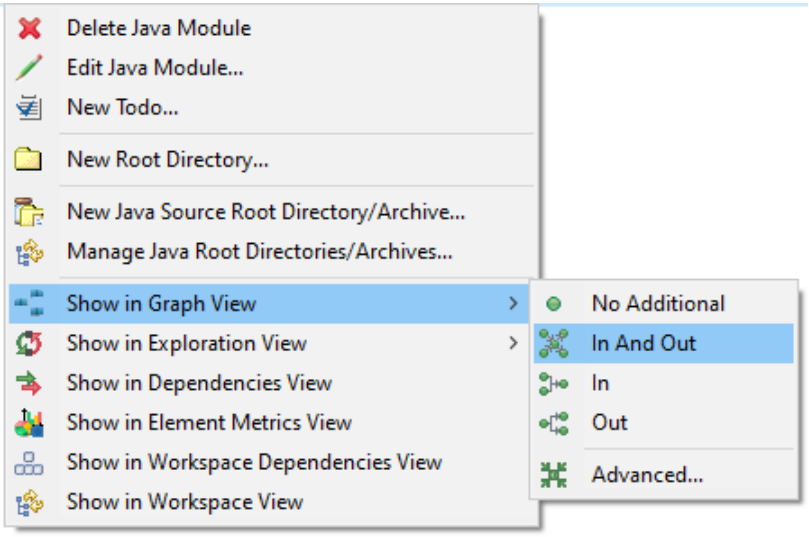


Figure 8.26. Quick View Creation

A new view will be created with the supplied focus parameter, only direct dependencies (not transitive), both internal and external elements and all the parser dependency types available.

Advanced View Creation

If more configuration options are need upon view creation, users can right click on a selection, select the view to open and click on 'Advanced...'. That will open the Advanced View Creation dialog where all available options can be configured.

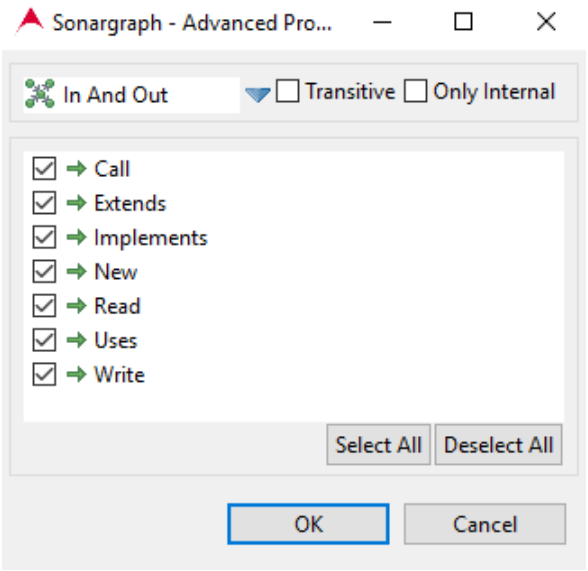


Figure 8.27. Advanced View Creation Dialog

The dialog shown in the previous image shows the advanced representation creation dialog where focus mode, properties and dependency types can be specified prior to creation. See: Section 8.11.2.1, “Focus Modes”, Section 8.11.2.1, “Graph-Based Views Properties”

Applying Focus

Since Graph-based views can display an overwhelming amount of information, it is possible to perform focus operations on these views in order to reduce the amount of displayed information to a set of nodes and edges that the user wants to focus his attention on. Focus operations are performed by using the focus toolbar.

Figure 8.28. Focus Toolbar

Focus Properties

The focus operation requires the combination of the Graph-based properties (See Section 8.11.2.1, “Graph-Based Views Properties”) and the following properties of its own:

- Selection: Users can focus a view using the selected or unselected elements. It is also possible not to use a selection, in which case, only the dependency types will be modified.

- Only Visible : If selected, only nodes and edges visible at the time of the focus operation will be considered, otherwise, all the model will be used for the focus operation.

Quick Focus

The focus button in the focus toolbar offers a dropdown menu with 4 options to perform a quick focus operation.

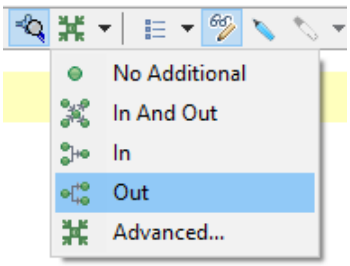


Figure 8.29. Focus Dropdown Menu

As the image shows, there is one menu entry for each focus mode. These menu entries can be used to perform quick focus operations which will use the current selection, the current value of the 'Only Visible' button, the focus mode of the menu entry and keeps all the other properties of the current view (dependency types, only internal and transitive).

Advanced Focus

If more configuration is needed to perform a focus operation, users can push directly the focus button in the focus toolbar or use the 'Advanced..' menu entry in the dropdown menu of this same button. Any of these operations will open the 'Advanced Focus' dialog which allows the configuration of all view and focus properties.

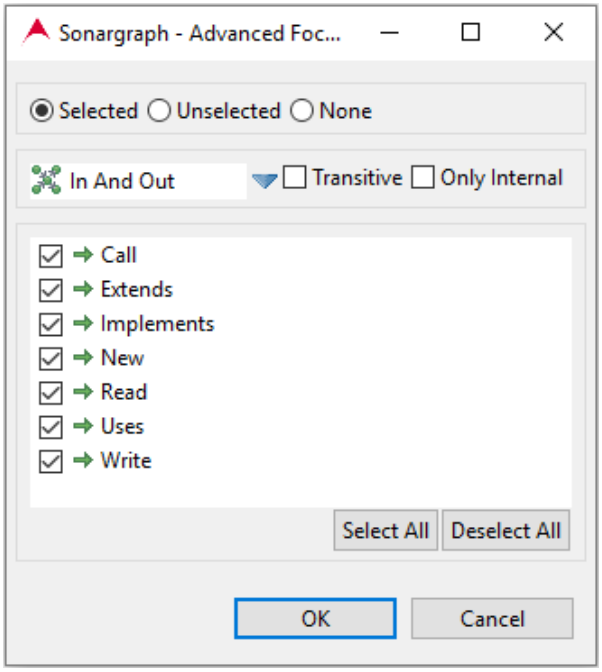


Figure 8.30. Advanced Focus Dialog

Architecture and Colors

When the used license contains the Architecture feature, the Exploration, Graph, Cycle and Dependencies views will use a set of colors to show whether the edges (or dependencies in the case of the Dependencies view) contain architecture violations or not.

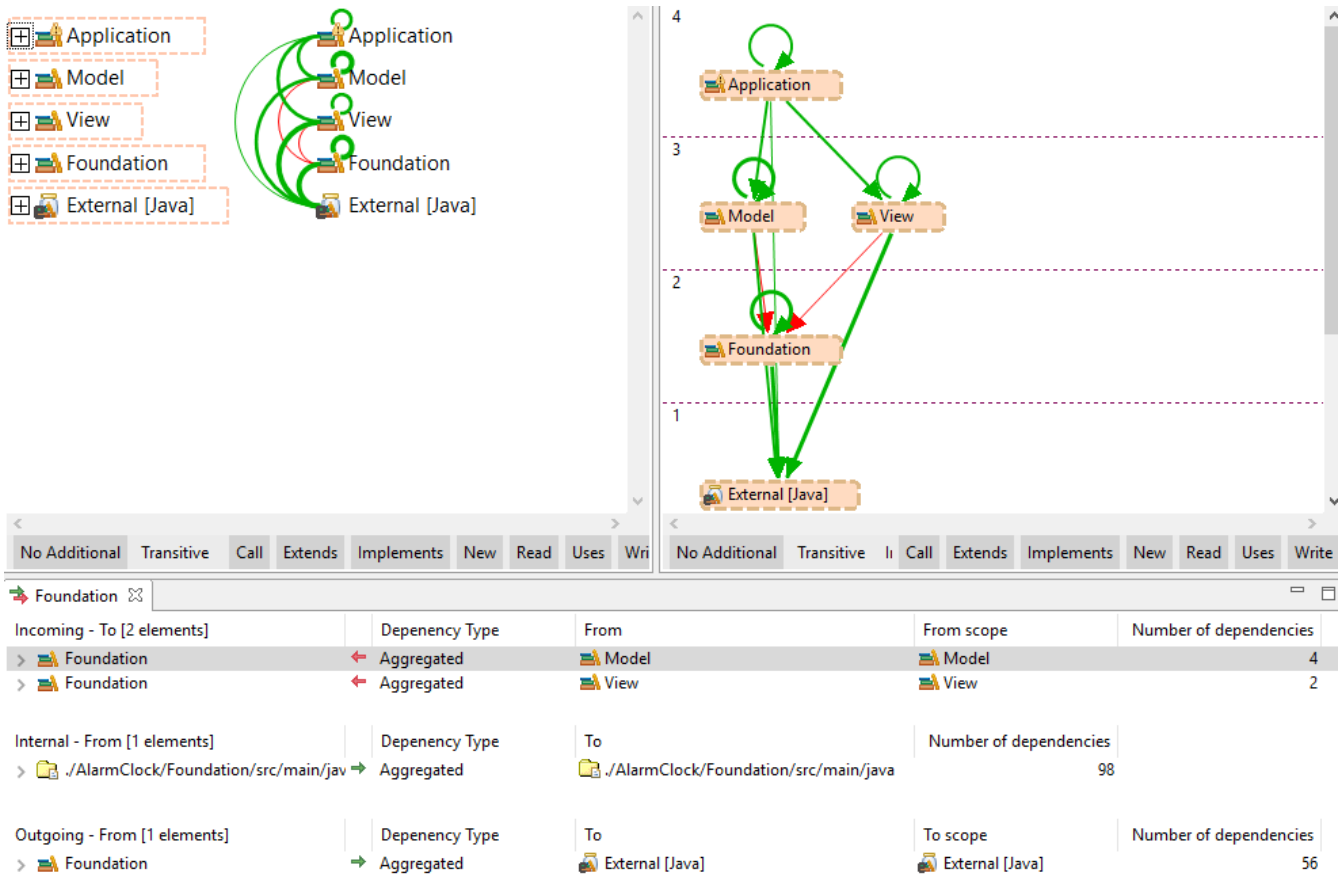


Figure 8.31. Semantics of Colors

- Green Color: All underlying dependencies comply with the architecture.
- Yellow Color: At least one but not all of the underlying dependencies create architecture violations.
- Red Color: All underlying dependencies create architecture violations.

Besides these 3 colors, views can eventually show some dependencies that will always be grayed. This means that these dependencies are not taken into account by the architecture check, thus they are neither allowed nor violating dependencies.

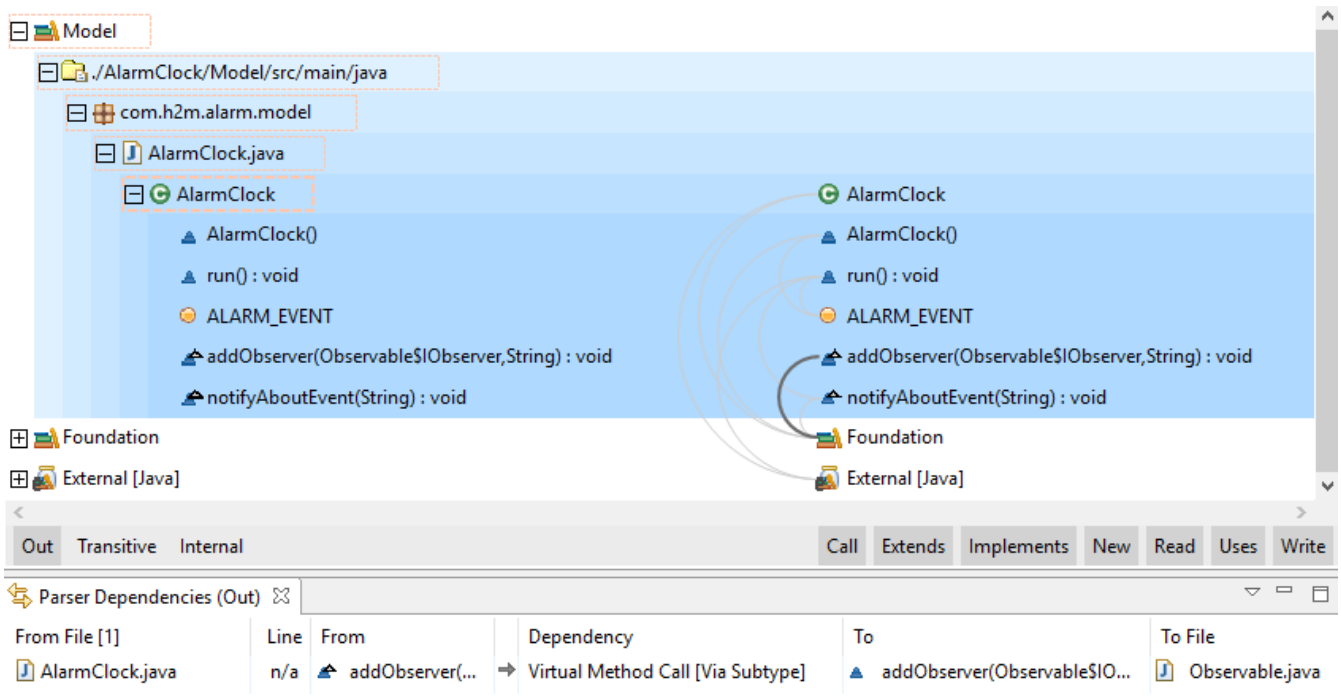


Figure 8.32. Architecture Independent Dependencies

Selection and Colors

In the case of the Exploration Graph view the selection will cause edges to display or hide their architecture-related color. If a node is selected, its incoming and outgoing edges will reveal their architecture-related color, all other edges will be grayed. If an edge is selected, it will be the only one revealing its architecture related color and the rest of them will be grayed. This is of course assuming that a license containing the *Sonargraph* Architecture feature is installed, otherwise, incoming and outgoing edges of selected nodes as well as selected edges will have a darker gray color whereas all other edges will have a lighter gray color. This is also the case for Workspace Dependencies and Include Dependencies views, where edges will always be presented in gray-scale colors.

8.11.2.2. Levels

Showing levelized content is a feature of the Graph view. When a Graph view is requested, *Sonargraph* organizes nodes in a way that given each edge of the graph and both From and To endpoints of the edge, the From endpoint will always be in a greater level than the To endpoint.

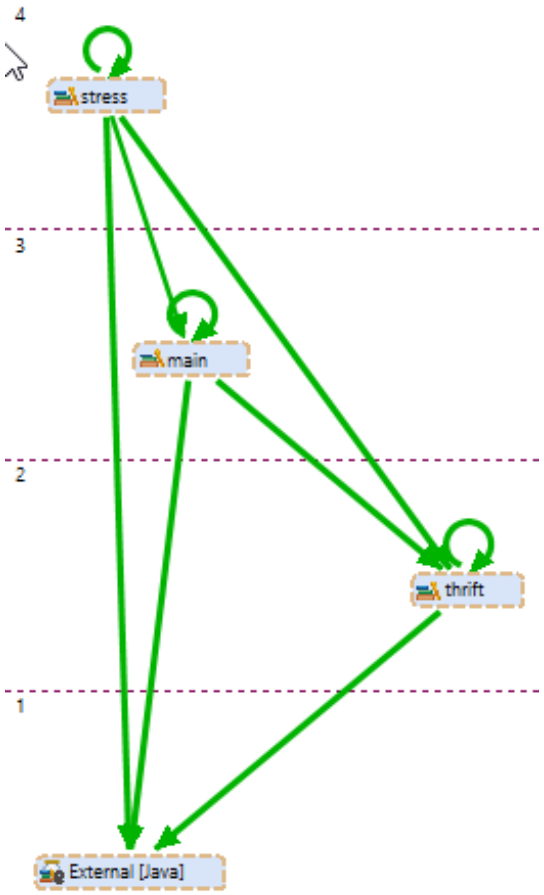


Figure 8.33. Levels in Graph View

This way of layouting immediately gives the user an idea about the coupling among the selected elements that form the content of the view and how modifications will impact elements that belong to different levels.

8.11.2.3. On Demand Cycle Groups

When creating a Graph view for an arbitrary selection of elements, it is possible that there are cyclic dependencies among the elements that make part of the content of the view. In this case, it would be impossible to define levels among the elements that belong to a cycle and with all of them belonging o a same level, the readability of the graph would decrease. To avoid this effect, *Sonargraph* gathers all elements that form cycle groups into elements called "On Demand Cycle Groups".

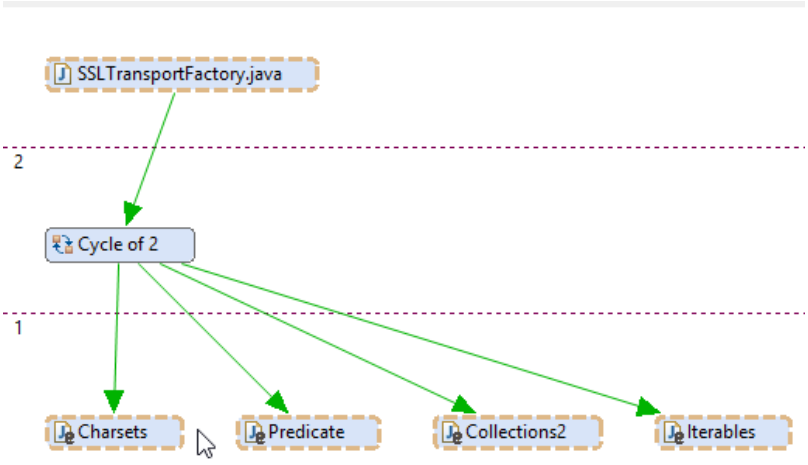


Figure 8.34. On Demand Cycle Groups

For a quick reference of the elements that are involved in an On Demand Cycle Group, hover the node with the mouse and a tooltip with appear with the list of cyclic elements.

8.11.2.4. Interaction with Auxiliary Views

The Graph view offers interaction with the Auxiliary views of *Sonargraph* (Parser Dependencies in and out views to be precise). This interaction allows users to see the underlying parser dependencies that are represented by the edges in the view. Auxiliary views can be used in two ways from the Graph view:

- Edge selection: By selecting an edge and having the Parser Dependencies (Out) view in front, it is possible to see the underlying parser dependencies for that specific edge.

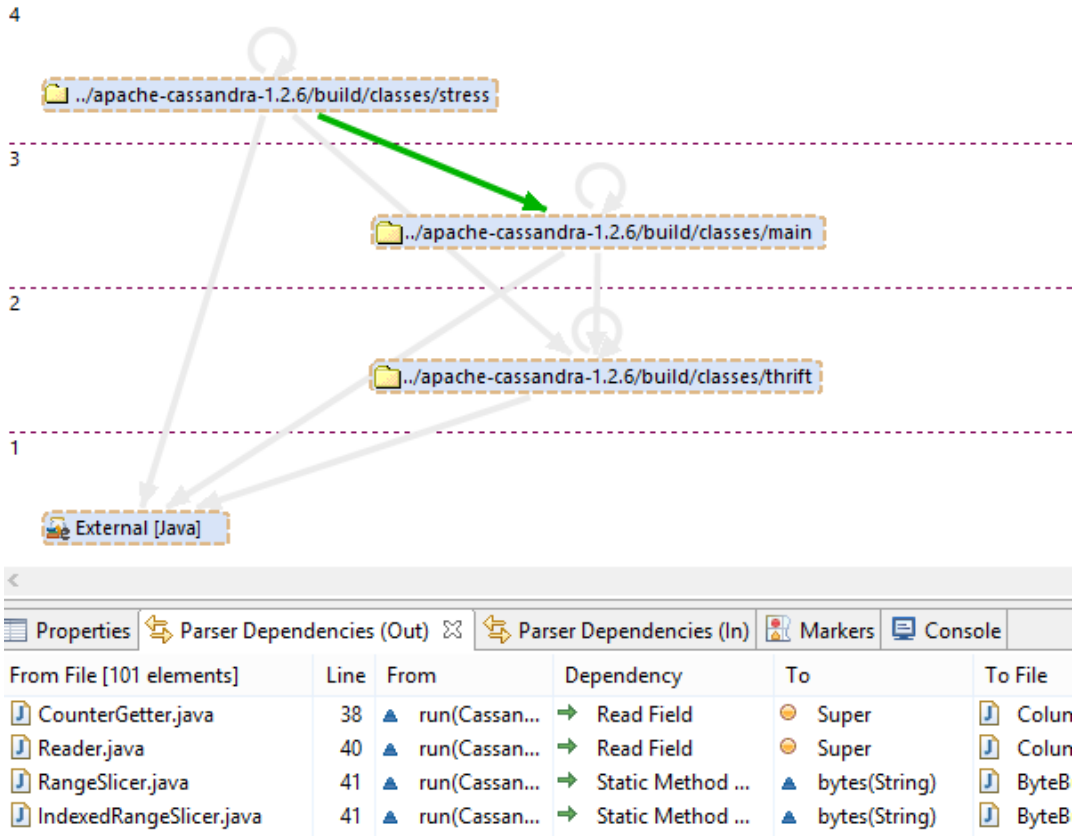


Figure 8.35. Underlying Parser Dependencies for Edge

- Element selection: By selecting only one element and having the Parser Dependencies (In) or (Out) in front, it is possible to see the underlying incoming or outgoing parser dependencies of the edges that come into the node or go out of it.

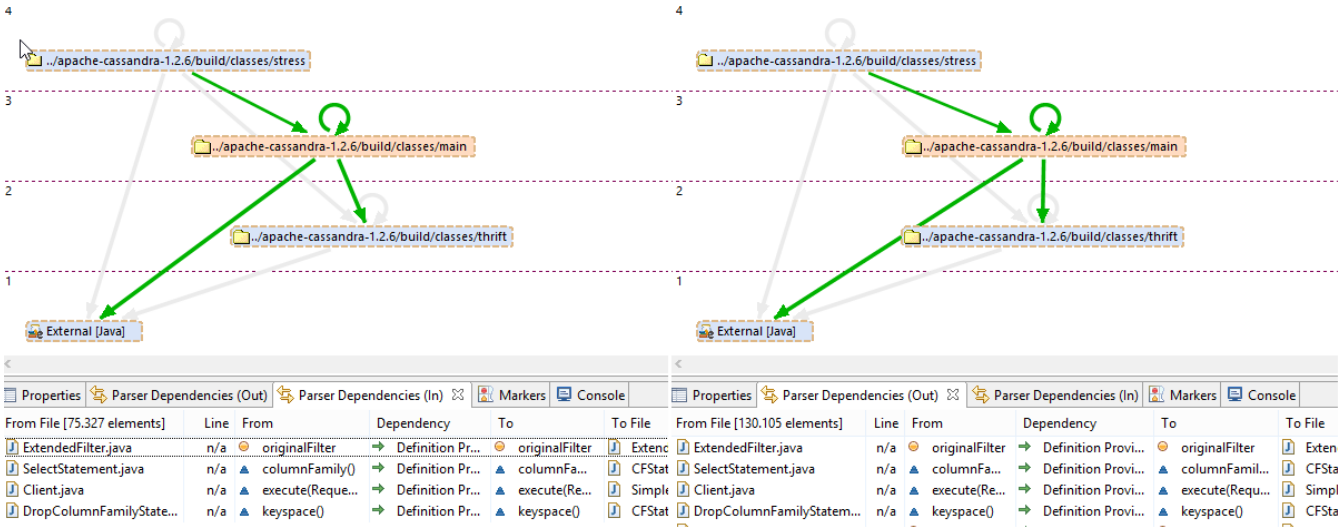


Figure 8.36. Incoming and Outgoing Parser Dependencies

8.11.2.5. Context Menu Interactions

Sonargraph offers navigation possibilities from the Graph view to other views in order to extract the greatest amount of valuable information from the software system analysis. To see the navigation possibilities, select a single edge or an arbitrary number of nodes and press right-click button.

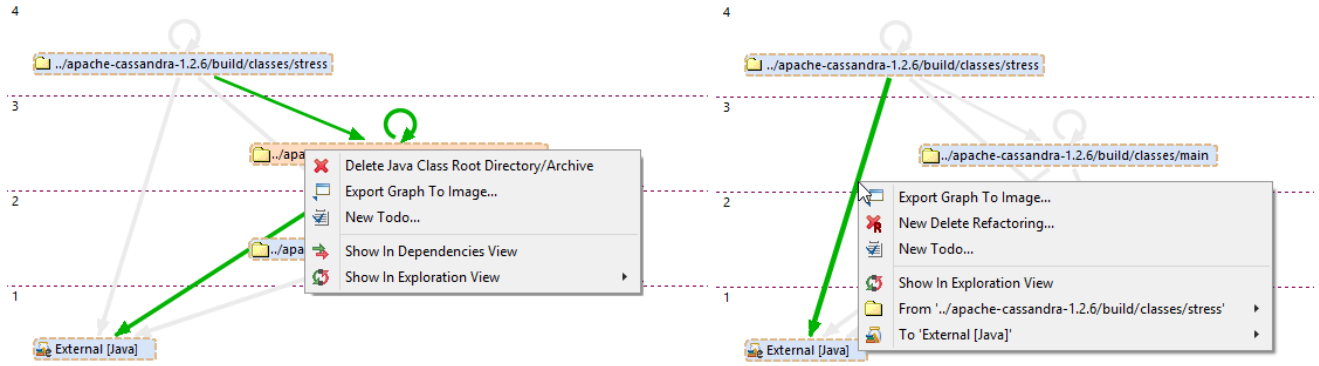


Figure 8.37. Context Menu Interactions

8.11.2.6. Type Based Graph

When selecting a Type (e.g. classes or enums), the regular Graph view will show incoming and outgoing dependencies of the selected elements to all kind of programming elements (fields, methods functions etc). To perform specific analysis like Java hierarchy graphs, it is necessary to show the dependencies between Types (Java classes in this case) that aggregate the underlying parser dependences to other programming elements than are children of types. To show a Type-based graph, select a Type in the navigation view and select 'Show in Graph View (Type-Based).

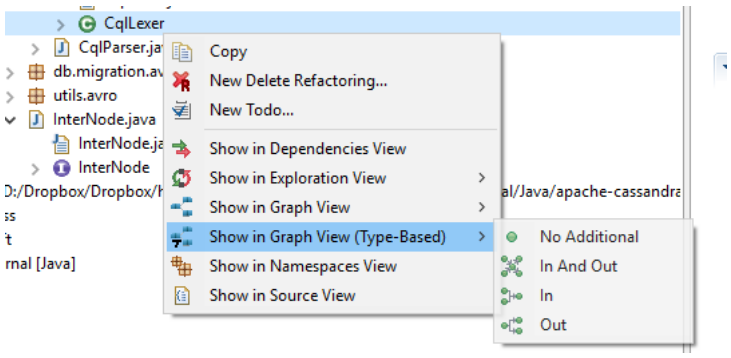


Figure 8.38. Show Type-based Graph view

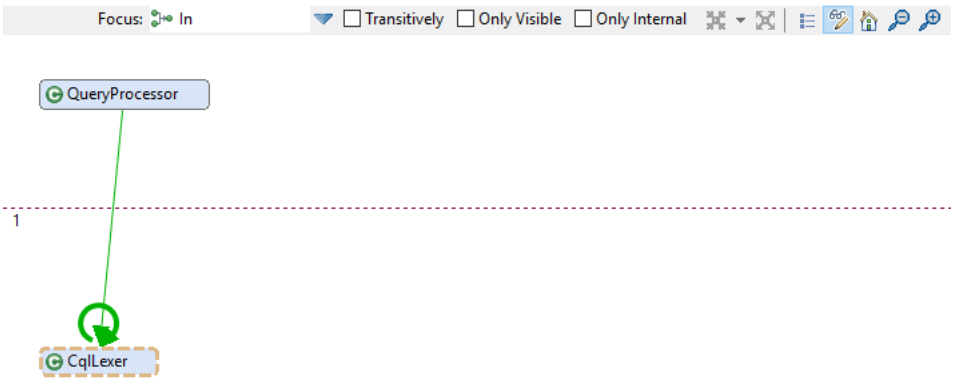


Figure 8.39. Type-based Graph

8.11.2.7. View Options

To change the way the content is displayed in the Exploration view, the options that are located at the right-hand side of the view's toolbar can be used.

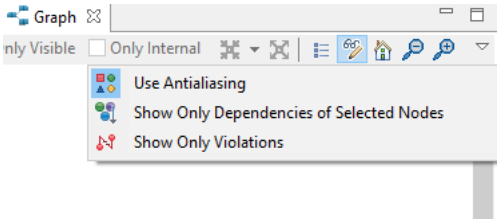


Figure 8.40. View Options

- Highlight Input : When activated, an orange-dashed box will be drawn for the nodes used as input to create the view.
- Use Antialiasing : When activated, the edges will look smoother and better defined, however, it is recommended to deactivate this option when running *Sonargraph* on low-end hardware.
- Show Only Dependencies Of Selected Nodes : When this option is activated, only the incoming and outgoing arcs of the selected elements will be shown, the rest will be hidden. If there is no selection, all arcs will be shown.
- Show Only Violations : When this option is activated, only arcs containing architecture violations are shown. If all underlying parser dependencies of the arc are violations, then the arc will remain unchanged. If the arc has both violating and non-violating parser dependencies, it will change from yellow to red and the width will be adjusted with the weight of the violating dependencies.
- Hide Self Arcs : When activated, edges whose from and to endpoints are the same node will be removed from the view.

8.11.3. Treemap-Based System Exploration

The Treemap View allows users to create a 2D / 3D representation of the system to find out where the hotspots are. See also Section 9.2.2, “Identifying Issue Hotspots”

Leaf elements are shown as squares. Their relative size is determined by the used size source. Their color is determined by the used color source. Optional: Their height is determined by the used height source. Parent elements show up as rectangles using grey color shades representing the nesting depth.

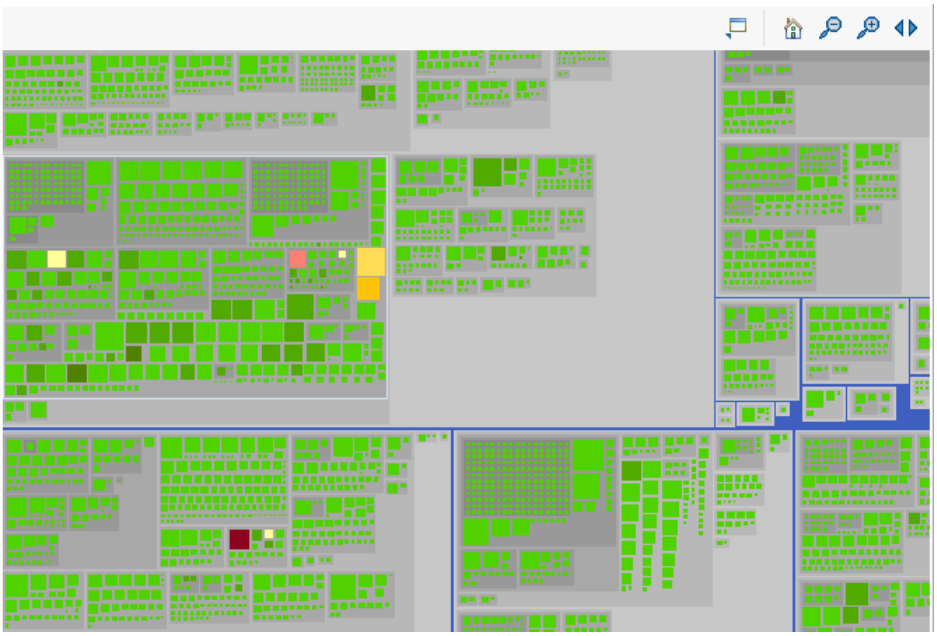


Figure 8.41. Treemap 2D View

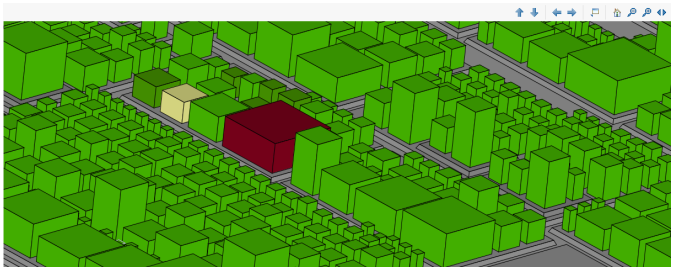


Figure 8.42. Treemap 3D View

Configuration of a Treemap

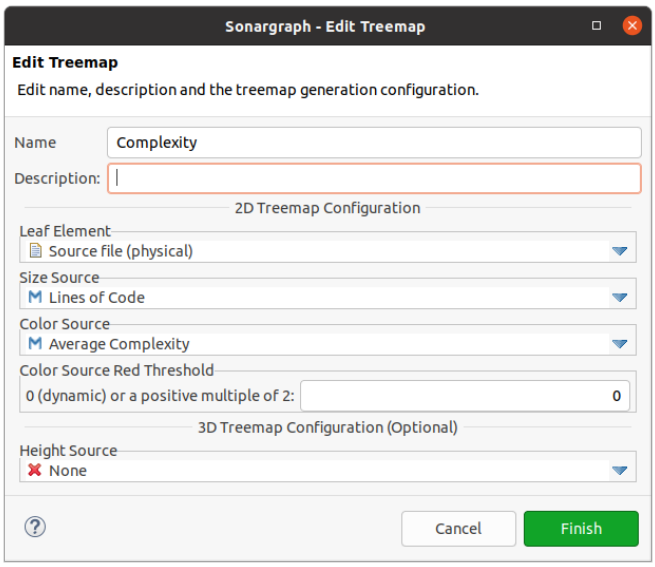


Figure 8.43. Treemap Configuration

The generation of a treemap is controlled by the shown configuration options, which are stored in an XML file with the specified name.

- Select the elements to use as leafs.
- Select the size source which determines the resulting relative size of the leaf elements.
- Select the color source which determines the resulting color of the leaf elements.
- Set the red threshold. If set to '0' an even mapping of values to green, yellow and red is dynamically calculated. When set to a value greater than '0', that value will be the first to be represented with a red color. The red threshold must be 0 or greater than 0 and a multiple of 2.
- Optional (3D): Select the height source which determines the resulting height of the leaf elements.

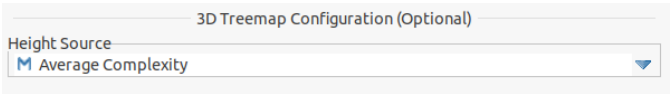


Figure 8.44. Treemap configuration 3D

In the generated treemap leaf elements will be shown as squares making it easier to spot the relative size differences. The color palette used for the leaf elements contains 3 green, 3 yellow and 3 red shades. For an 'ascending' color source (i.e. less is better) a darker color represents a higher number. For a 'descending' color source (i.e. higher is better) a darker color represents a lower number. Parent elements show up as rectangles using grey color shades representing the nesting depth.

There is a special color or height source named Issue Collector, which counts the leaf element's number of issues. The issues to collect can be filtered by resolution and severity.

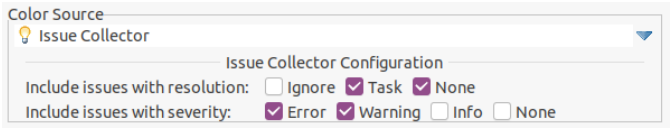


Figure 8.45. Treemap Issue Collector

Interaction with Auxiliary Views

When the option 'Link Master Views' in the top level toolbar is enabled, selecting a square/rectangle will reveal the corresponding underlying element in the master view. The Properties view will show information about the corresponding underlying element of the selected square/rectangle.

Context Menu Interactions

The Treemap view offers the following context-menu interactions:

- New Delete Refactoring: Create a new Delete Refactoring for the selected element.
- New Move/Rename Refactoring: Create a new Move/Rename Refactoring for the selected element.
- New Todo: Create a new Todo for the selected element.
- Export Treemap View To Image: Export the Treemap as image.

Toolbar Interaction

The toolbar of the Treemap view contains interactions to change the size and view of the Treemap:

- Auto Resize : When activated, sets the zoom level to fit the current window size.
- Zoom in / Zoom out : Increases or decreases the zoom level.
- Home : Sets the zoom level to 1.0. Resets any rotation (3D only).
- Roll (3D only) : Rolls the treemap to the left or to the right.
- Rotate vertically (3D only) : Rotates the Treemap up and down.

Mouse Interactions

Scroll wheel: Use the modifier key (CMD, CTRL) of your operating system in combination with the scroll wheel anywhere in a Treemap to zoom in or out at current mouse pointer position.

Left button drag: Drag the treemap around with left mouse button and SHIFT key pressed.

Hover: Hovering over a square/rectangle will open a tooltip showing additional information. That tooltip can be focused by clicking into it with a left mouse click.

8.11.3.1. Tabular Representation of Treemap Data

Treemaps are great for visually spotting hotspots, but if you want to actually work with the information, a tabular representation is better. The 'Treemap Info' view can be opened via the context menu and displays information about the treemap's leaf elements as shown in the following screenshot:

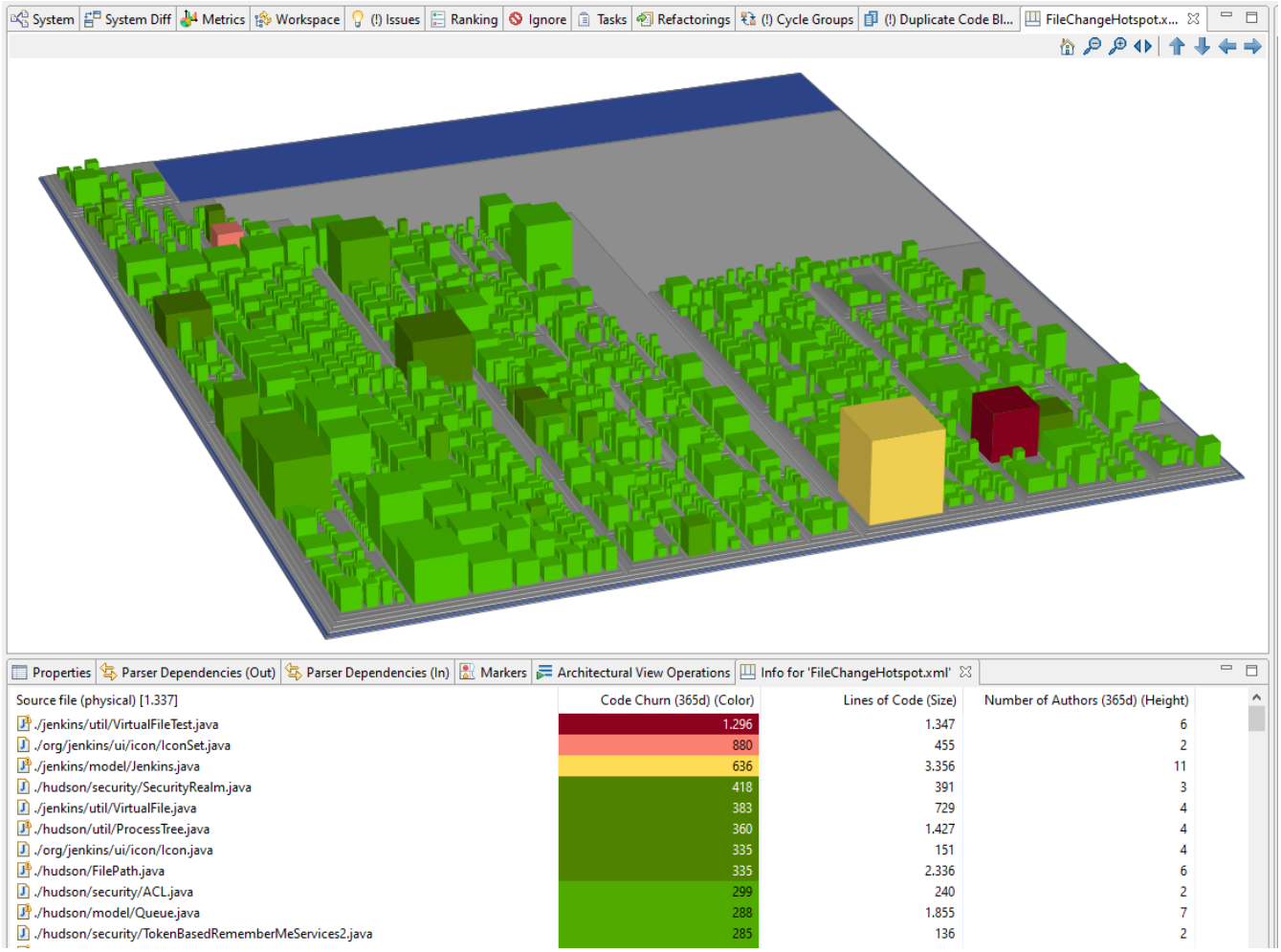


Figure 8.46. Treemap Info View

The table supports the usual interactions like sorting, filtering, export to Excel, etc.. Selecting elements in the 'Treemap Info' view highlights those elements in the treemap by greying all other elements:

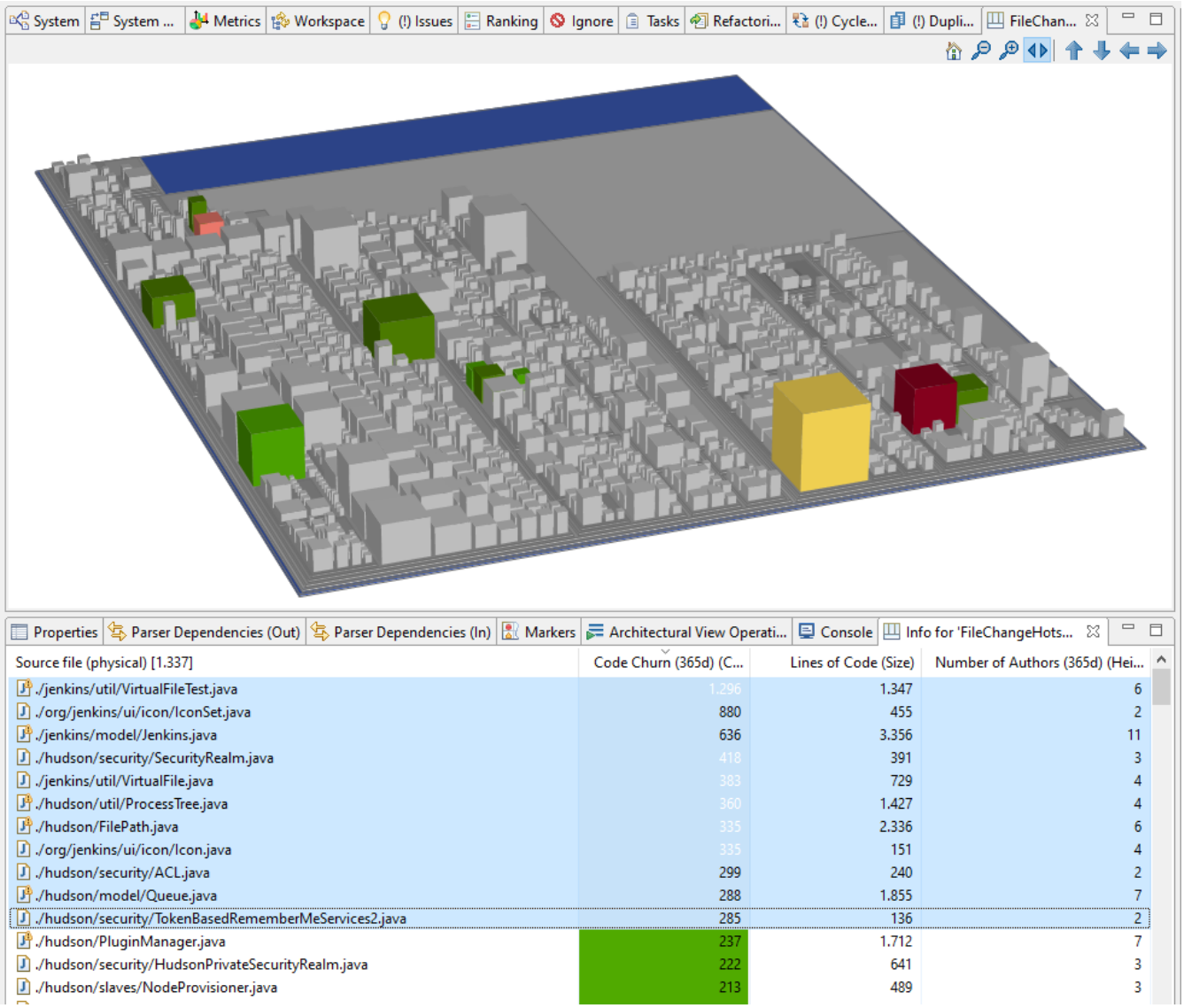


Figure 8.47. Multi-Selection in Treemap Info View

8.11.4. Tabular System Exploration

Sonargraph also offers the possibility of exploring the system in a tabular way through the Dependencies view. By selecting a single element of the parser model, users can observe and explore its incoming, internal and outgoing dependencies.

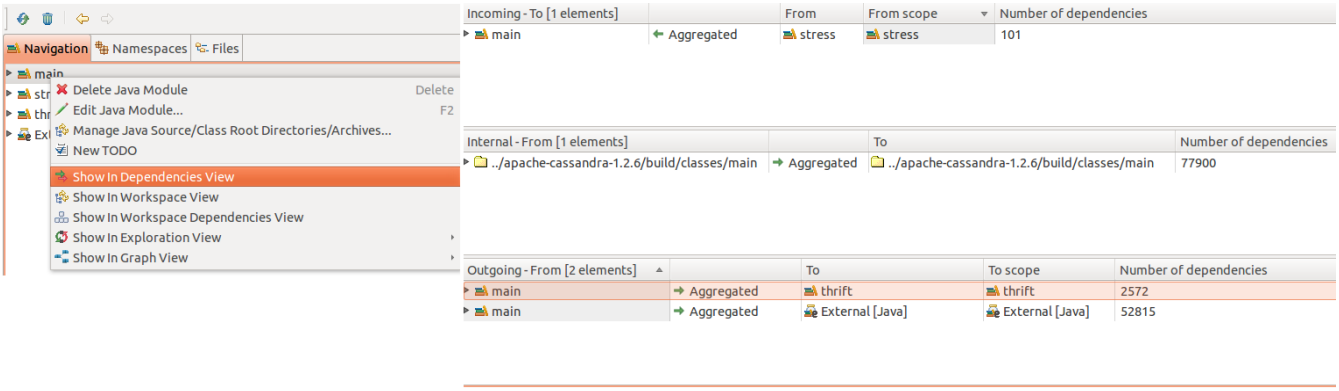


Figure 8.48. Dependencies View

8.11.4.1. Drilldown

One of the most helpful features of the Dependencies view is its capability to allow users to drilldown from the dependencies between the top-most elements in the model to the dependencies between the finest-grained elements in the parser model.

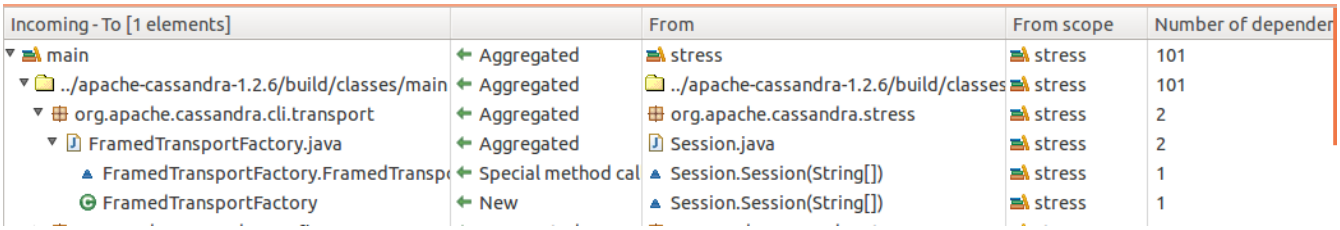


Figure 8.49. Drilling Down Dependencies

It is important to note that as seen in the previous figure, some elements that belong to the parser model are not taking into account when drilling down in the Dependencies view in favor of readability. For example, packages "org", "apache", "cassandra" and "cli" do not play one role in the drilling down other than providing a context for the element that is really makes part of the content which is "transport". This apply as well for other structures that allow nesting such as Namespaces and Directory paths in C/C++ and C# parser models.

In a similar way, when the Dependencies view is requested for an element that allows nesting, the selected element will take part in the content of the view only if it has elements of a different kind as children, otherwise, it will be omitted. Similarly, all children of the selected element that fulfill the same condition will be displayed for this request.

8.11.4.2. Interaction with Auxiliary Views

The Dependencies view offers interaction with the Parser Dependencies (Out) auxiliary view of *Sonargraph* . This interaction allows users to see the underlying parser dependencies that correspond to each entry in the incoming, internal and outgoing dependencies tables. By selecting only one dependency and having the Parser Dependencies (Out) in front, it is possible to see its underlying parser dependencies.

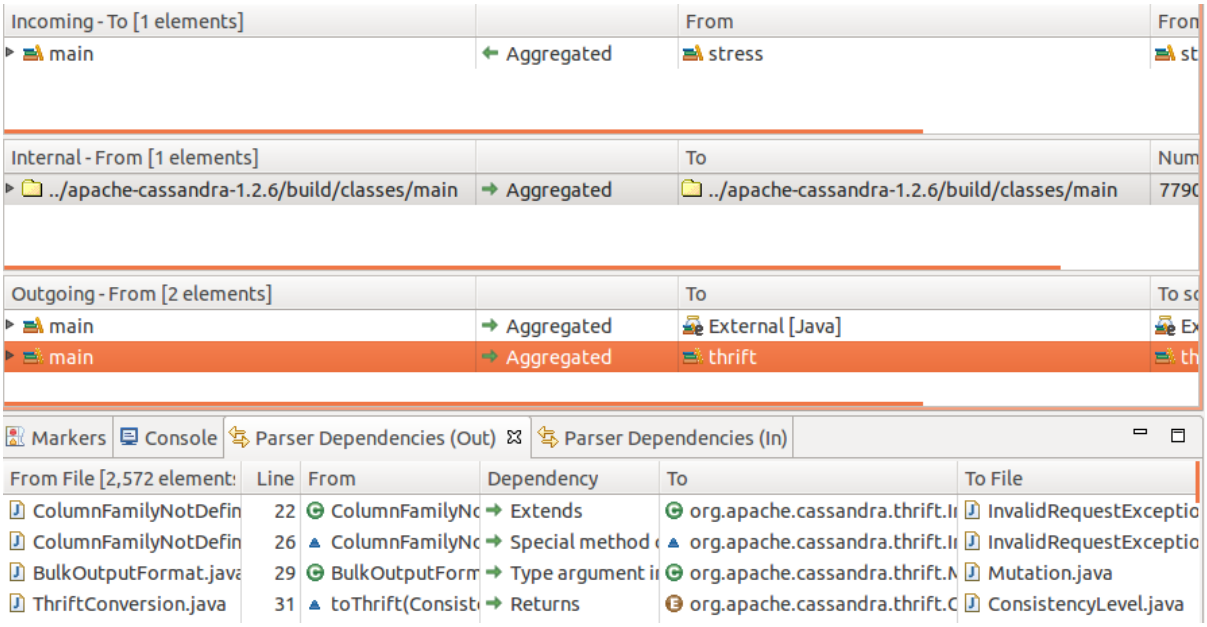


Figure 8.50. Interaction with Auxiliary Views

8.11.4.3. Context Menu Interactions

Sonargraph offers navigation possibilities from the Dependencies view to other views in order to extract the greatest amount of valuable information from the software system analysis. To see the navigation possibilities, right-click any dependency in the view and select whether the interaction should consider the From or the To endpoint of the dependency. Depending on the selected endpoint, navigation possibilities will show up.

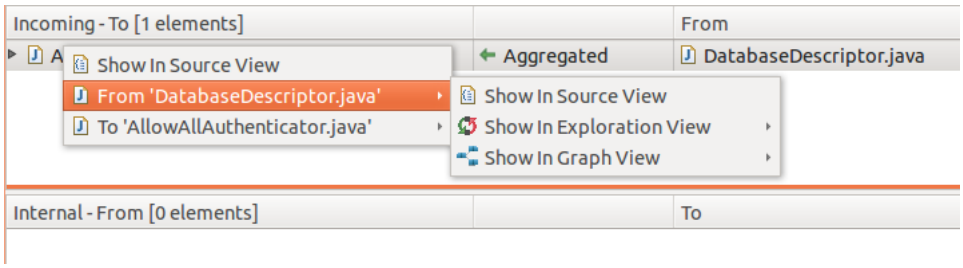


Figure 8.51. Context Menu Interactions

8.12. Searching Elements

For systems with a very large code base, finding *programming elements* can sometimes prove challenging. *Sonargraph* offers a search dialog to quickly locate *programming elements* in the currently open *software system*.

To bring it up select "Edit" → "Search..." .

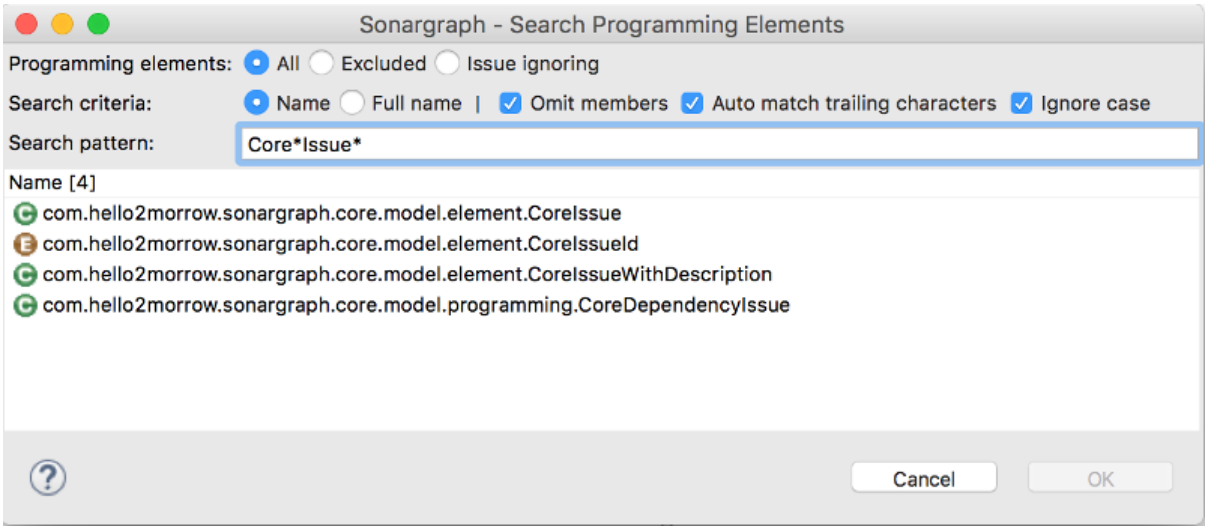


Figure 8.52. Standard Search Dialog

The dialog will start revealing potential matches as soon as you start typing the element you are looking for. You can search for simple or complete name of a *programming elements* and choose to omit members, search for excluded elements only, auto match trailing characters and search ignoring case.

After selecting the correct element, the Navigation view highlights the found element.

If you want to extend the search to also find methods or member variables, deselect the option "Omit members". If also "Full name" is activated, filtering by packages and types is possible as shown in the following screenshot.

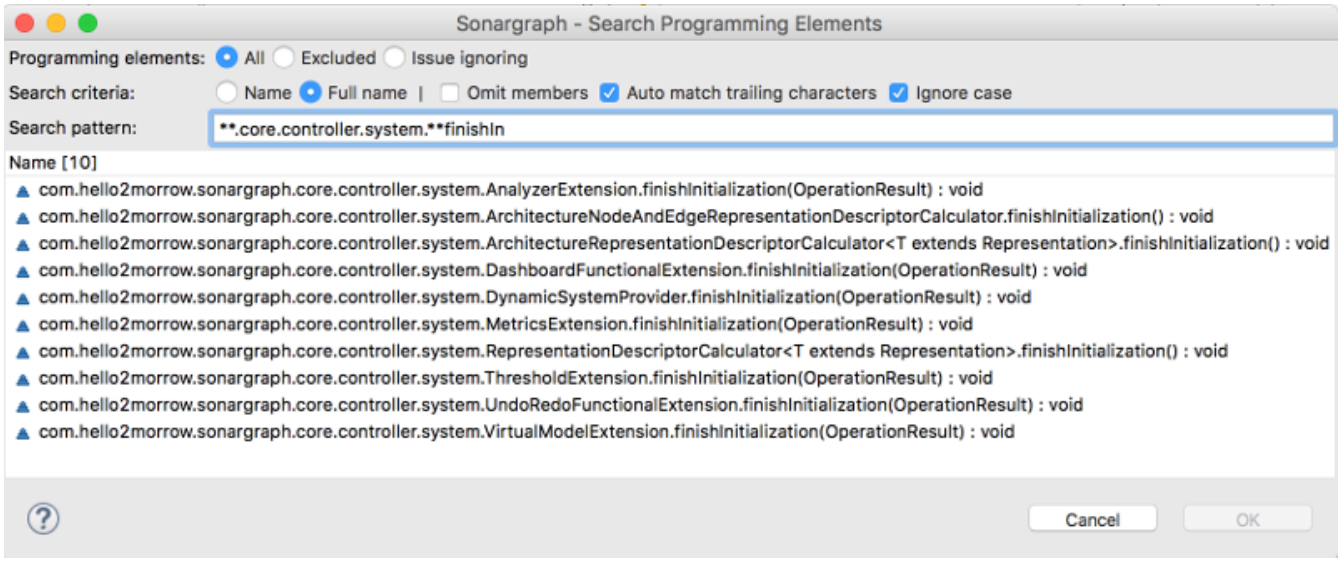


Figure 8.53. Search Dialog to Locate Members

NOTE that you can either search in all programming elements or restrict the search to 'Excluded' or 'Issue ignoring'. This might be used to check if the workspace filters (Production Code filter and Issue filter) have been configured correctly.

8.12.1. Searching Elements in Views

Text search functionality is supported by Graph and Exploration views and most table-based views, like the Issues view, Refactorings view, Metrics view, etc. This functionality makes it easy to navigate a view that displays a high number of elements. Matches are highlighted as shown in the following screenshot.

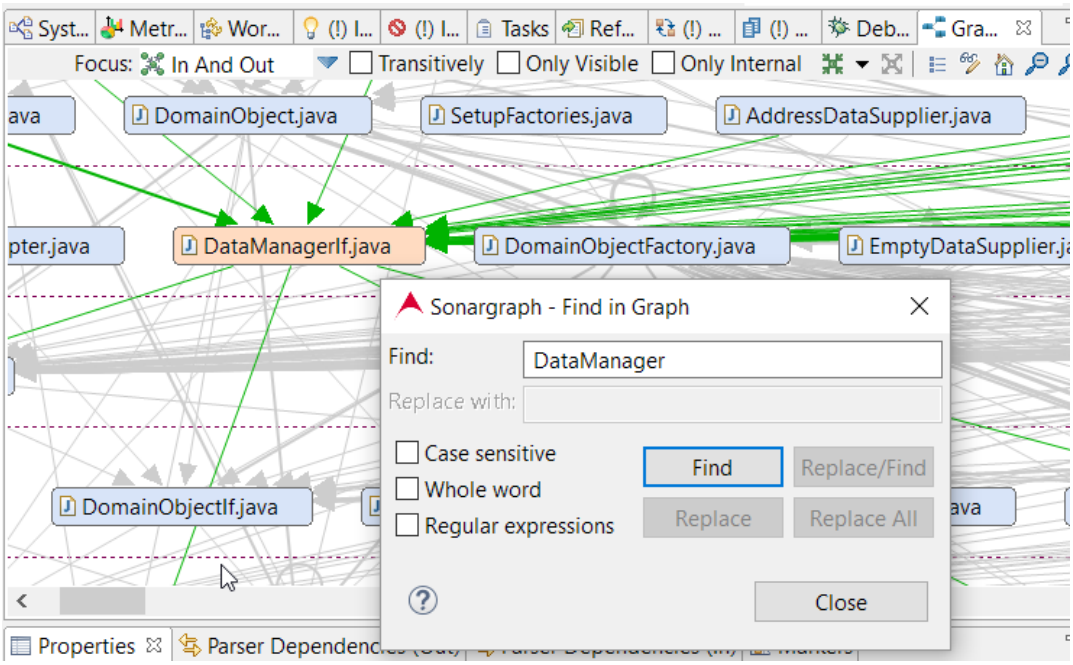


Figure 8.54. Text Search in Views

8.13. Detecting Duplicate Code

Duplicate code analysis in *Sonargraph* is achieved through the Duplicate Blocks view and the Duplicates Source View. The Duplicate Blocks view lists duplicate code blocks that have been found in source files of the system that have not been excluded from analysis via a filter. For each duplicate block, all the occurrences are listed, with source file, length of the block in lines, start line of the block, and the tolerance, i.e., a number of lines that are different to another text block.

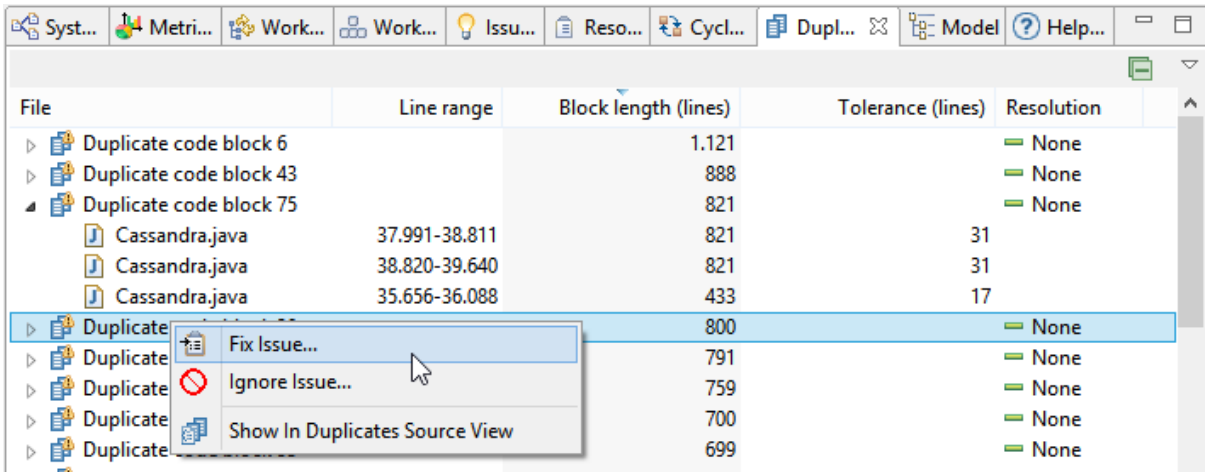


Figure 8.55. Duplicate Blocks View

Duplicate code blocks are considered as issues in *Sonargraph* as they make the maintenance of the code base more difficult. Thus, duplicate code blocks can also be found in the form of Issues in the Issues view of *Sonargraph* (See Section 9.2, “Examining Issues”). The context menu for a duplicate block (both in the Duplicate Blocks view and the Duplicates Source view) allows to take care of it as an issue, by either ignoring it or creating a fix resolution for it.

By double-clicking (or selecting "Show In Duplicates Source View" in the context menu) on a line that represents a duplicate code block, one jumps to the Duplicates Source View where the selected occurrence is presented side by side along with the next occurrence in the block so that the similarities and differences can be appreciated:

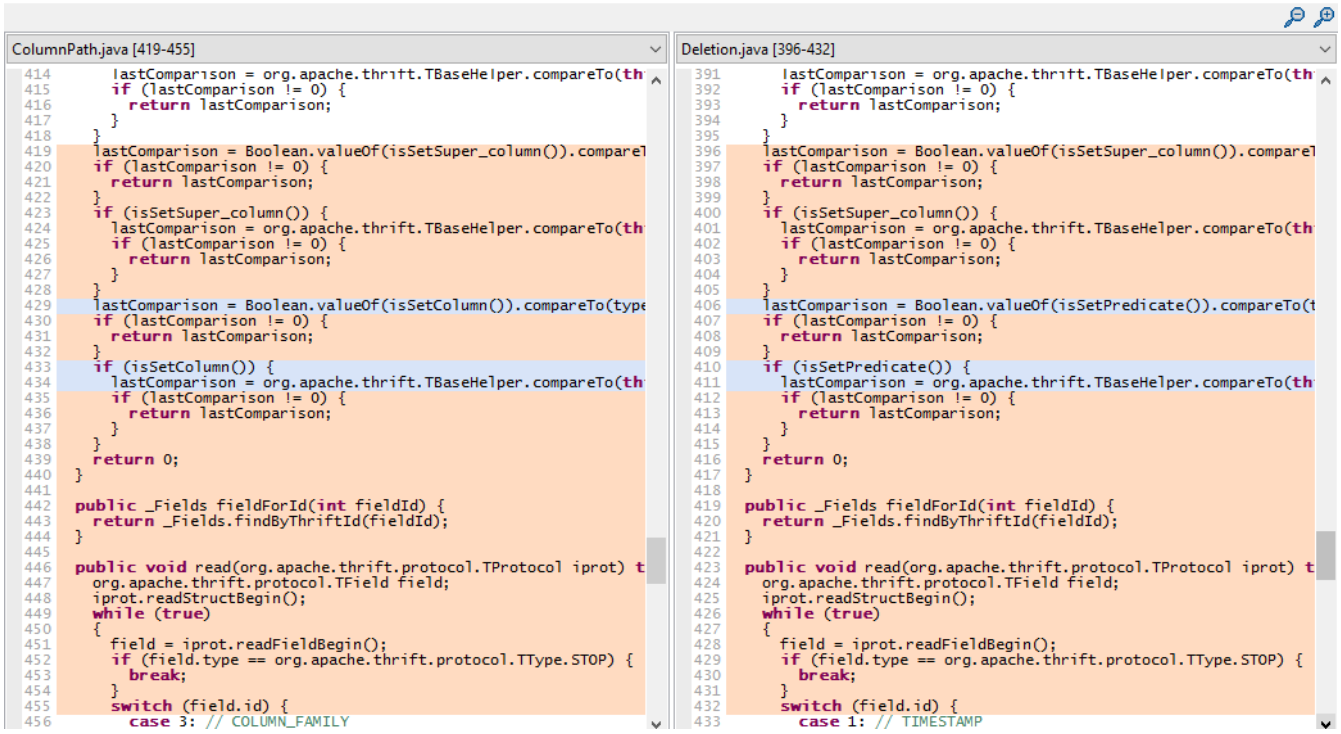


Figure 8.56. Duplicates Source View

Normally, duplicate code blocks are computed automatically on every *software system* open or refresh via the duplicate code analyzer.

8.13.1. Configuration of Duplicate Code Blocks Computation

The settings for how duplicates are located can be adjusted at "System" → "Configure..." → "Duplicate Code" . Usually, the default settings are acceptable. In order to understand how the configuration parameters work, it is helpful to know how the algorithm works. The main process is as follows:

- First, candidates for start lines of duplicate code blocks are determined. For this, all lines of all source files are read.
- If a line is too short (shorter than the number given in the configuration parameter "Minimal Line Length"), it is discarded. This allows to save memory, since all other lines might have to be stored if there occur copies of them.
- Each non-discarded line is space-normalized (i.e., sequences of white space characters are replaced by a single space character; and words that are not separated by whitespace characters are separated by a single space character). This normalization allows to detect almost-copied blocks that only differ from each other by the whitespace in them.
- Lines that occur too often (more often than the number given in the configuration parameter "Maximal Number of Copies") are discarded. This feature is used for excluding e.g. preambles that start every file from duplicate analysis.
- For any pair of identical lines that result from the steps above, it is checked if they are the start of a duplicate code block. Only blocks that have a certain minimum length are reported (configuration parameter "Minimal Block Length").
- Two other parameters allow for a certain "slack" in the comparison so that not only completely identical blocks are found, but also blocks that differ a bit.

  1. The configuration parameter "Maximal Tolerance per Edit" works like this: When two text blocks are compared, the comparison algorithm allows some differences, or "edits". Each single edit may only add, remove or change a number of lines (the one given by this parameter) in order to make the blocks identical. Note that behind the edited region, the two blocks must continue identically for at least one line.
  2. The configuration parameter "Maximal Relative Tolerance Percentage" works like this: When comparing two blocks, the number of edited lines in relation to the number of matched lines may never be larger than this percentage.

  The total number of lines in all the edits that occur in a block comparison is the "tolerance" of the comparison. The larger it is, the more lines need to be changed to consider the two blocks to be copies from one another.
- The algorithm up to this point only identifies pairs of start lines of duplicated blocks. The last step in the identification of duplicated blocks is the aggregation: Not only are code blocks considered to be duplicates of one another when they form result pairs in the algorithm above, but also when they are indirectly copies of one another. E.g., consider two already identified pairs of duplicated blocks A,B on the one hand and C,D on the other hand, where the start of B equals the start of C; then A, B, C and D are all considered to be duplicates of the same code block. This aggregation is done until no more blocks can be aggregated. The tolerance specified for a code block in the view is the minimal tolerance that occurred during the comparison of the block with other code.

8.14. Examining the Source Code

Anywhere in the *Sonargraph* workbench you have the option of double clicking (or right clicking + "Show In Source View") on an element to show the source code of the clicked element if that is available.

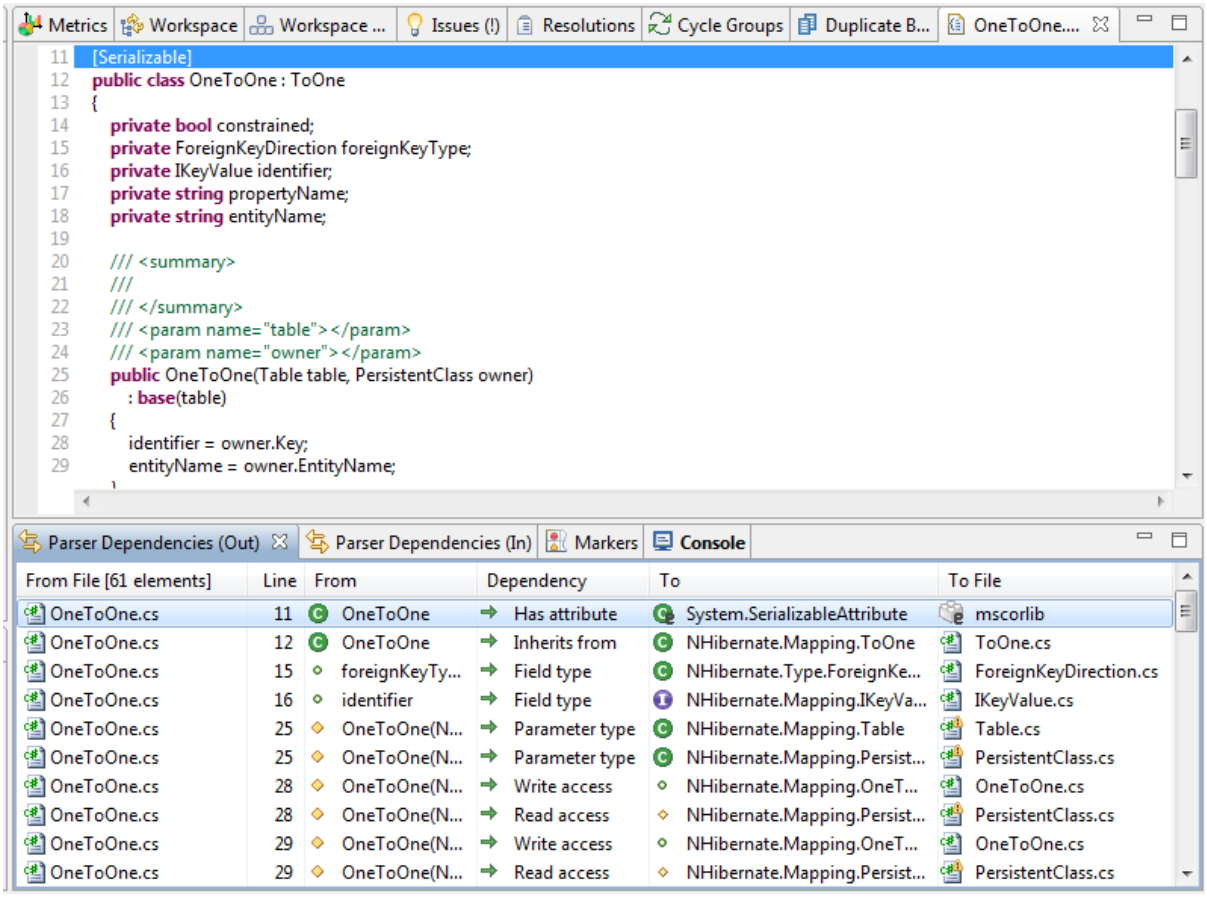


Figure 8.57. Source View

"Find Text" feature can be invoked with Ctrl-F (Command + F on Mac). If more than one occurrence of the search string is found, press F3 to jump to the next search result. In the Script view (see Chapter 16, *Extending the Static Analysis*) and Architecture File view the search feature offers also "replace" and "replace all" to ease content edition.

Regular expressions can be used for advanced match and replace use cases. Line-breaks in the replacement text can be specified with \R. The implementation uses standard Java regular expression API and also allows using capturing groups. More details can be found at the *JavaDoc of java.util.regex.Pattern* and *JavaDoc of Capturing Groups* .

As you move around the mouse cursor through the source code, you can see that some elements (names of fields, methods, types and so forth) are being underlined. By pressing Ctrl (Command on Mac) and clicking on this 'hyper linked' elements you navigate to the definition of the element which might be defined in the same source file or another.

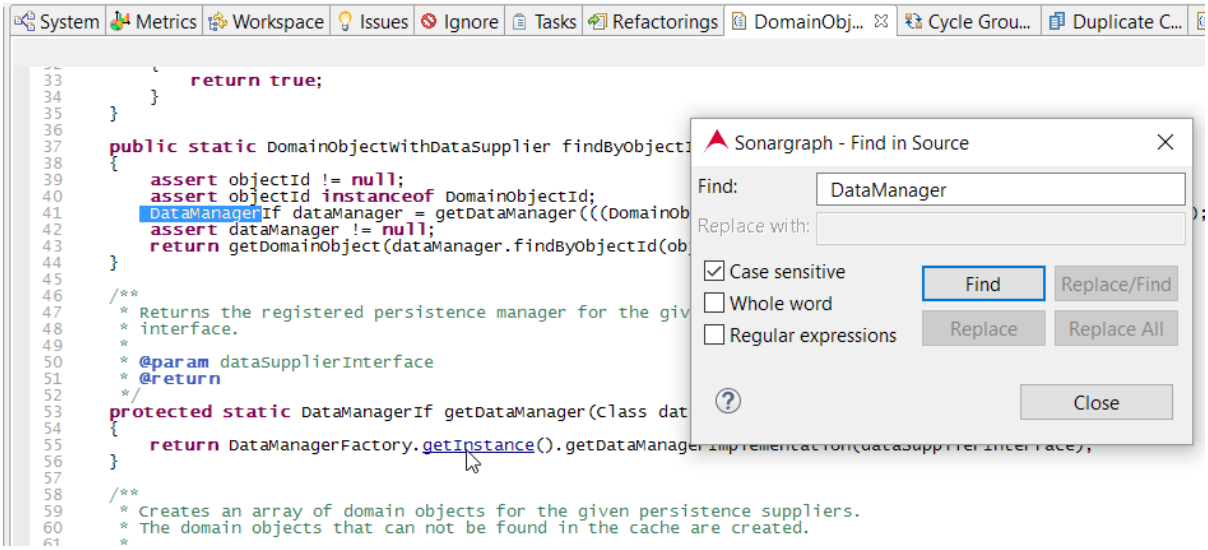


Figure 8.58. Hyperlinking and Find Text Features in Source View

8.14.1. Interaction with Auxiliary Views

The Source view offers interaction with the Auxiliary views: Parser Dependencies (Out), Parser Dependencies (In) and Markers: Below the main source viewer, three tabs provide further information about the currently loaded source file:

- The Parser Dependencies (Out) tab lists all dependencies that depart from the source file. Clicking on a dependency jumps to the respective line in the upper pane of the source view.
- The Parser Dependencies (In) tab lists all dependencies that arrive into the source file. Clicking on a dependency opens up another instance of the Source view showing the file where the selected incoming dependency belongs to.
- The Markers tab lists all the markers of the source file. Markers are graphic indicators of issues in the source file under inspection. It also shows user defined tasks (See Section 9.4, “Defining Fix and TODO Tasks”) and refactorings that pertain to the file or elements in the file. Clicking on a marker jumps to the corresponding line in the upper pane of the source view.

8.15. Examining Metrics Results

Sonargraph calculates metrics on different abstraction levels and displays them in the Metrics view. Metrics are calculated on different levels. Select the level in the combo box at the top left of the view and select the metric in the shown table. The list of metric values is then displayed in the first tab on the right. The scope (i.e. for the whole system or a single module) can be selected via the combo box at the top right of the view. The complete set of metric values can be exported via the context menu. Some basic statistics like average, standard deviation, median, minimum and maximum values are displayed below the list of values on the right.

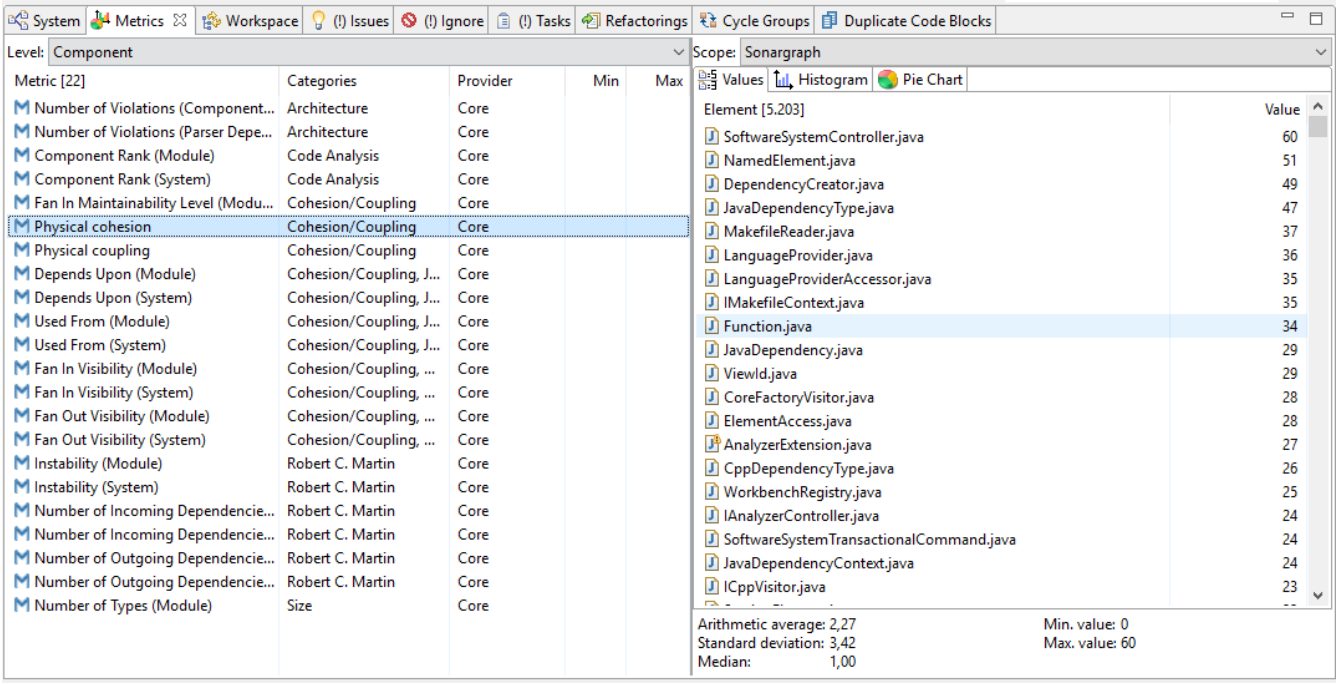


Figure 8.59. Metrics View

The histogram for the selected metric and scope is shown in the second tab on the right. The chart can be exported as an image via the context menu. The pie chart is only available for metrics with a defined threshold.

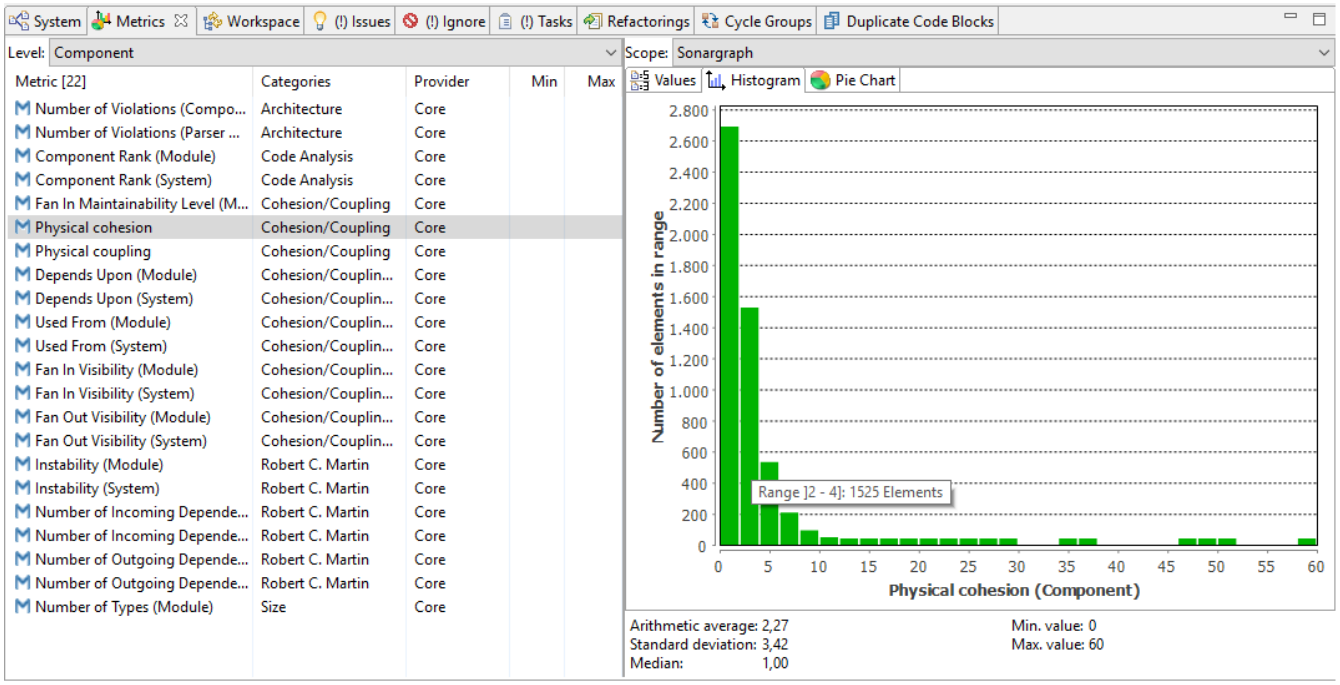


Figure 8.60. Metrics Histogram

If you are interested in all metrics of a specific element, the Element Metrics view can be opened via the main menu "Window" → "Show View" → Element Metrics.

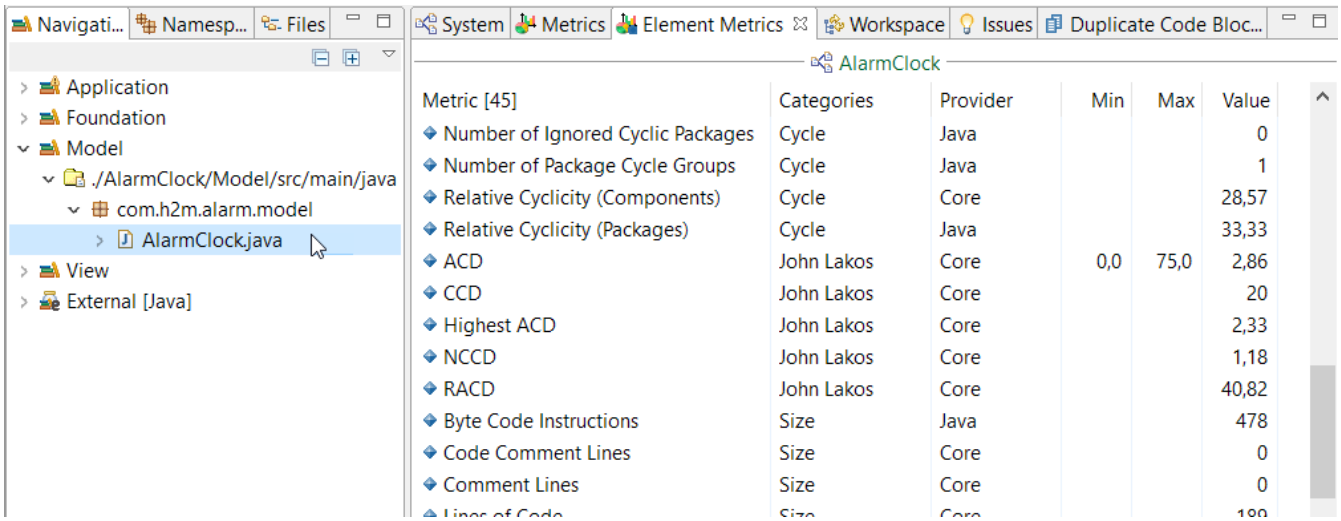


Figure 8.61. Element Metrics View

The metric thresholds configuration allows to define threshold values for those predefined metrics in order to have an accurate control of the behavior of your code base as it evolves.

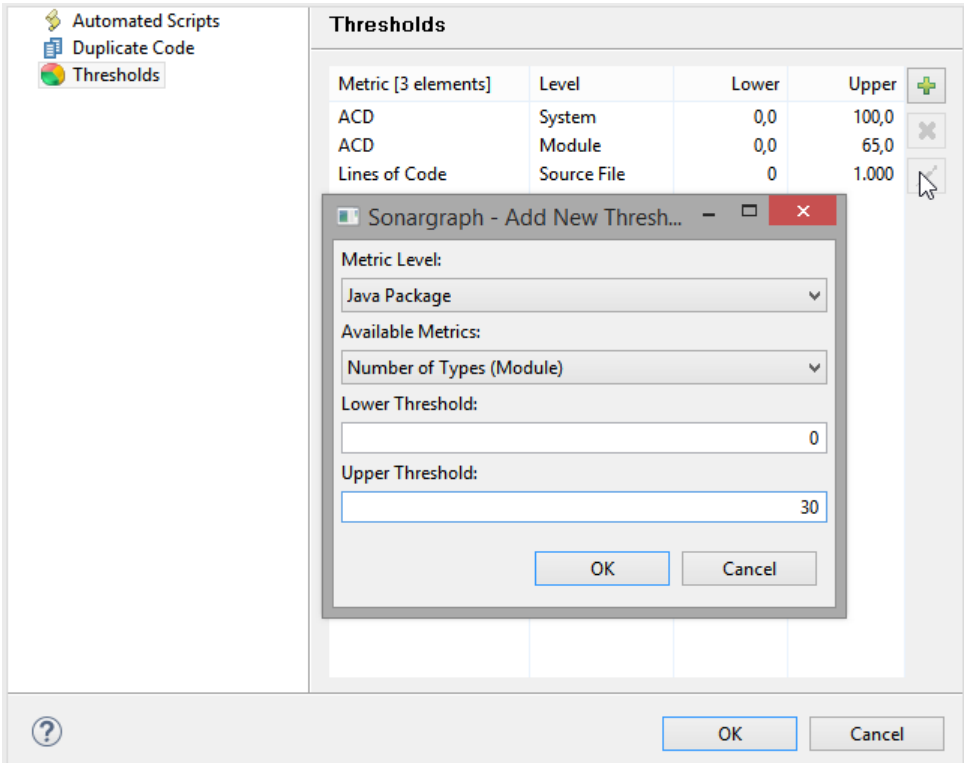


Figure 8.62. Metric Thresholds Configuration

Thresholds can also be defined, edited or deleted via context menu by right clicking on a metric in the Metrics or Element Metrics view.

Related topics:

- More information about the built-in metrics can be found in Chapter 21, *Metric Definitions*.
- Custom metrics can be defined using Groovy Scripts. More information is contained in Chapter 16, *Extending the Static Analysis*.

8.16. Analyzing C++ Include Dependencies

The Include Dependency view is available via the context menu of a C++ source file, as shown in the following screenshot. It allows analyzing the dependencies to header files.

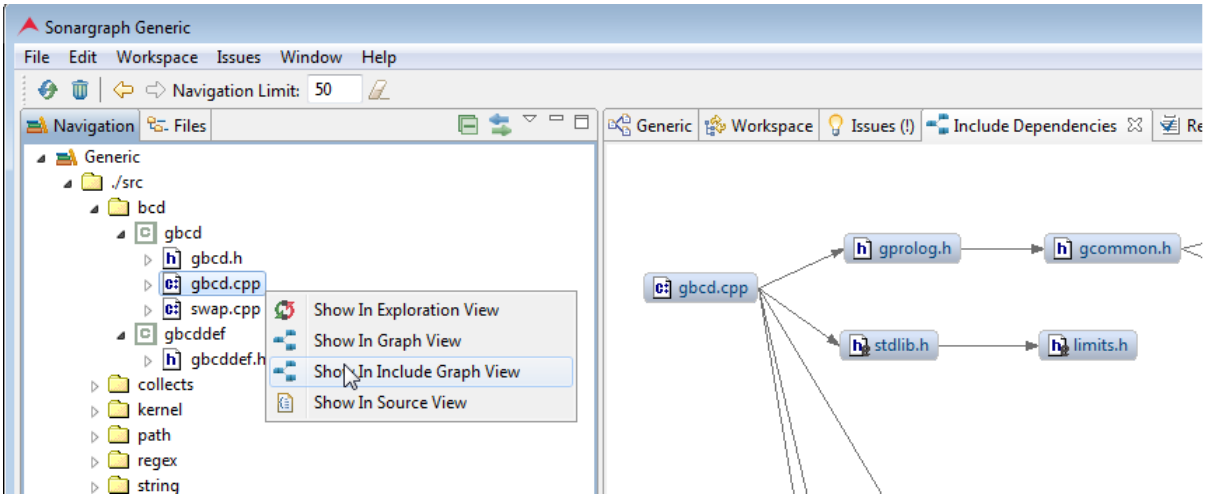


Figure 8.63. C++ Include Dependency View

8.17. Creating a Report

Select "File" → "Export to HTML/XML Report..." to generate an HTML or XML report containing all metric values, issues and resolutions (TODO, Ignore, Fix). You can select the element level for which the metric values are exported.

The corresponding XML schema can be found in <sonargraph-inst>/report.

TIP

The XML output is normalized to minimize the size of the file. To get a more expressive report, set the log level for the ReportExtension to "debug" in <sonargraph-inst>/logback.xml.

NOTE

The report files can easily get several MBs big and take a few seconds to generate. Start with the default configuration first to check the size and then increase the number of levels and the number of values per metric.

The HTML report contains tables that can be filtered, e.g. the table of "Unresolved Issues". The tables provide the following functionality:

- The table header allows to filter for rows containing the specified text as shown in the screenshot. Paging will be enabled for tables containing more than 25 rows. You can select to show 25, 50 rows per page or all on one page by using the combo box on the right.
- The table header provides info about the number of items shown and the current page.
- The matching terms are highlighted as shown in the screenshot.
- Several filter conditions can be connected via logical OR (||) and logical AND (&&).
- Table cells containing numeric values can also be filtered for value ranges as shown in the screenshot.
- Rows can be sorted by clicking on the table column header.
- All filters can be cleared by clicking on the right-most icon.
- A short help function is available by clicking on the question mark on the right.

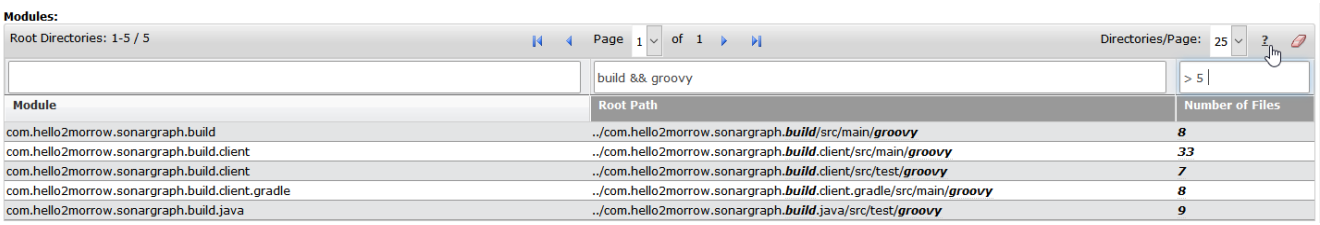


Figure 8.64. Table Filter Options in HTML Report

Chapter 9. Handling Detected Issues

This chapter explains the purpose of virtual models and how they can be used to define standard resolutions (Ignore, Fix) for detected issues. It is also described which views can be used to get a summary of all issues, see how Sonargraph has prioritized them and how to visualize hotspots.

The following views provide relevant information: Issues, Ranking, Ignore and Tasks view.

9.1. Using Virtual Models for Resolutions

Virtual models in *Sonargraph* are resolutions containers used to try different solutions for issues in the system without distorting its original status. *Sonargraph* ships with two *virtual models* already created: "Parser" and "Modifiable".

- *Parser* is the model that is generated by the language specific parsers without structural changes or any resolutions for created issues.
- *Modifiable.vm* is the (initially empty) model ready to save any refactorings or created resolutions by the user. It naturally depends on the *Parser* model.

Virtual models management section is located on the right-hand side of the main toolbar. Using the green plus symbol you can create as many different *virtual models* as you need to try out different resolutions with your *software system* and you will always have your original model available in the "Parser" model.

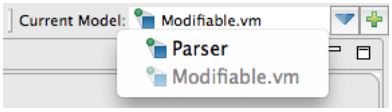


Figure 9.1. Virtual Models

The "Modifiable" model is selected by default so you can start creating resolutions (fixes, ignores or TODO's) right away. On the contrary, the "Parser" model, represents the "facts" model, determined by the parser, which means it can not be modified as it represents the actual state of things on your *software system*.

NOTE

A virtual model might affect metric values since the structure of the system can be changed with refactorings and issues can be transformed into tasks or ignored. So, depending on what you want you should select the corresponding virtual model. If you want to see the unaltered metrics and structure you should select the 'Parser' virtual model (or an 'empty' virtual model - without any refactorings or resolutions). In the user interface you can either select the virtual model in the right-hand side drop down menu in the upper toolbar or in the Files view on the left hand side underneath the Models folder with the corresponding context menu entry.

9.2. Examining Issues

Sonargraph offers views to support several use cases related to issues:

- Get an overview of all issues: The Issues view shows all detected issues, offers advanced filter options and aggregates issue counts into the physical structure (system, modules, roots, files), so that hotspots can be detected easily.

- Get a prioritized list of issues related to the source code (i.e. no issues related to the Sonargraph system are shown): The Ranking view shows the computed score for each issue. The score is based on the issue's urgency and importance. See section Section 9.2.1, “Identifying the Most Relevant Issues to Fix” for details.

- Get a list of ignored issues: Some detected issue might not be are relevant, e.g. a method violates consists of more lines than the defined threshold but splitting it up would make the algorithm more difficult to understand. Those issues can be ignored (i.e. they are no longer shown in the Issues view) and the Ignore view list all these ignore definitions.

- Get a list of defined tasks: Tasks can be defined for issues that must be fixed, including implementation suggestions for the developer. The issues are also no longer shown in the Issues view and the tasks can be tracked in the Tasks view.

The Issues view displays information about the found issues such as their severity, category, affected elements and the associated provider. The upper half of the view displays the affected elements in a tree, following the file structure of the code and the Sonargraph system files. The information about the number of affected elements, and numbers of issues of error, warning and info severity is aggregated for each element and its children, making it easier to identify hotspots, i.e. modules with a high number of detected issues. The list of issues is shown in the lower part for the selected elements and their children. If you want to see all issues of the system, either click on the white space below the tree or select the System and Installation root nodes. If you are only interested in issues for specific modules, select them in the tree and only issues related to code in those modules are shown in the table.

The presentation mode (flat, hierarchical, mixed) of the elements tree can be switched via the view options menu in the top-right corner.

NOTE

The numbers of issues and affected elements on a parent node are not necessarily the sums of the values of its children. This is caused on the one hand by "composite" issues, e.g. "Duplicate Code Block" and "Cycle Group" that affect several elements, but are only counted once for common parents of the affected elements. And on the other hand, the parent element itself might also be involved in issues.

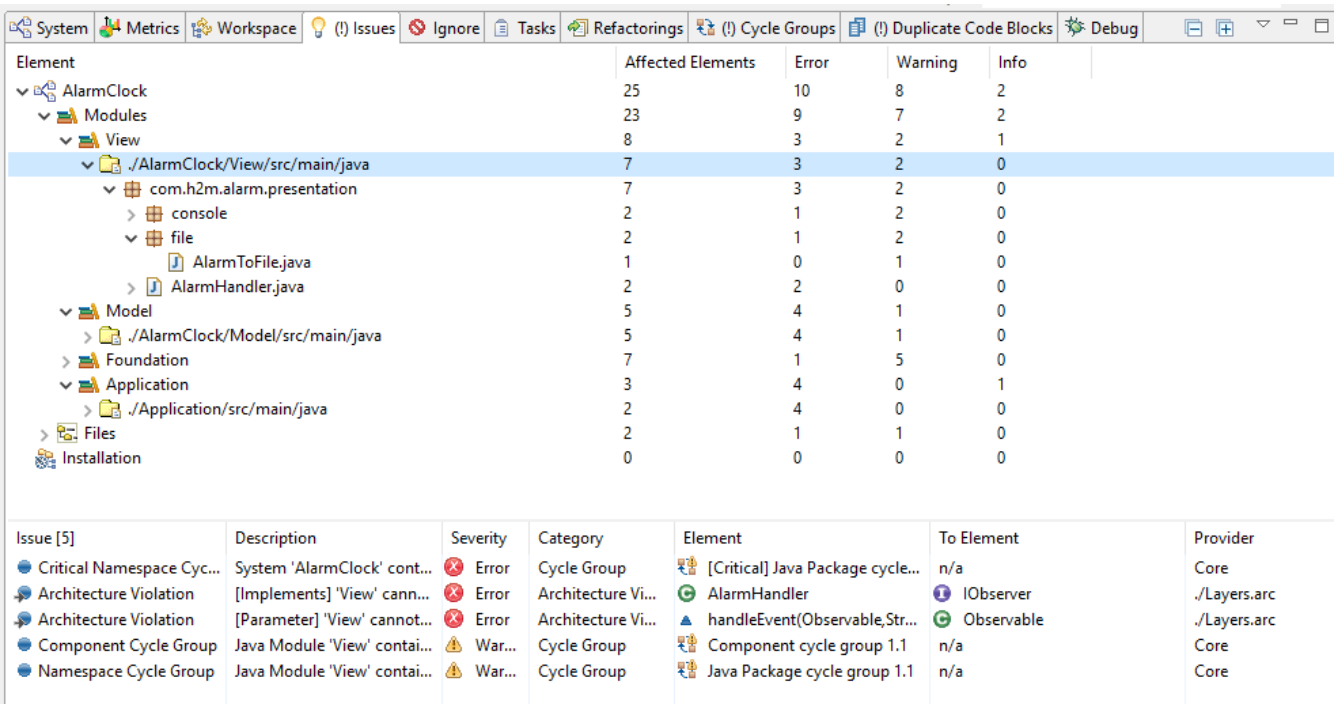


Figure 9.2. Issues View

Context menu and double click interactions give you options to examine the issue in a more suitable view. They also allow to "ignore" or "fix" the issue by either ignoring it or creating a fix request for someone in the development team. These requests are called "Resolutions" in *Sonargraph* and are covered in depth in the following sections.

In case of having too many issues, you can apply filters using the "Filter..." view option on the upper-right corner where several criteria are offered to reduce the amount of visible issues:

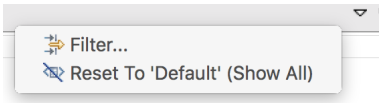


Figure 9.3. Filter Issues

A filtered view is indicated by a yellow background.

TIP

A text filter can also be applied to the table displaying the issues. See Section 8.1.4, “Tables” for details.

You can also focus on issues for certain code regions by defining an *Issue Filter* as described in Section 8.8.1, “Definition of Filters, Modules and Root Directories”.

9.2.1. Identifying the Most Relevant Issues to Fix

For existing systems Sonargraph might produce a huge number of issues. This is expected, if no static code analysis has been used before, so don't be discouraged! Now, what issues should be fixed first? We have implemented an algorithm in Sonargraph that borrows the main idea from "The Eisenhower Method"[1], that a problem has importance and urgency dimensions. The algorithm computes numbers for both and treats them as coordinates. The resulting score is defined as the distance from origin.

The goal of this algorithm is identifying those issues were fixes provide the most benefit. There is not much benefit in fixing issues in code that has not been changed during the last year. On the other hand, recently introduced issues are usually easier to fix since the context is still present in the developer's head. Also, issues that have a great impact like huge cycle groups that involve frequently changed code and that could be resolved by eliminating a small number of dependencies provide a higher benefit than refactoring a slightly too complex method.

The importance of an issue is computed including the issue category (e.g. architecture violation, threshold violation), severity and impact (e.g. the lines of code involved in a cycle group, the number of involved lines in a duplicate code block) as parameters.

The urgency is computed by including data from the source control management (SCM) to generate a boost for issues involving files that have been changed frequently and from the System Diff (see Chapter 14, *Examining Changes*) to generate a boost for new or worsened issues. Additionally, the number of references to break up a cycle group is included in the urgency calculation to generate a boost for cycle groups that are now still easy to fix, also known as 'low-hanging fruit'. Similarly, the 'tolerance' (lines being different) in duplicate code blocks is included to generate a boost for duplicate code blocks where it is now still easy to extract common logic, i.e. duplicate code blocks with a low tolerance.

NOTE

Treat the computed scores and the ranking as hints! Let us know if you notice that a certain type of issue is constantly ranked either too high or too low, or if you require further configuration options.

Details of the algorithm and individual computed values are displayed in the Properties view for a selected issue (see screenshot below). Large cycle groups usually get a very high score, since their impact on the system is high and it is likely that any of the involved sources have been modified. The selected duplicate code block shows a high "Urgency Ease of Fix" as both occurrences are identical (0 reported tolerance) and is therefore a low-hanging fruit.

[1] *The Eisenhower Method* , https://en.wikipedia.org/wiki/Time_management#The_Eisenhower_Method

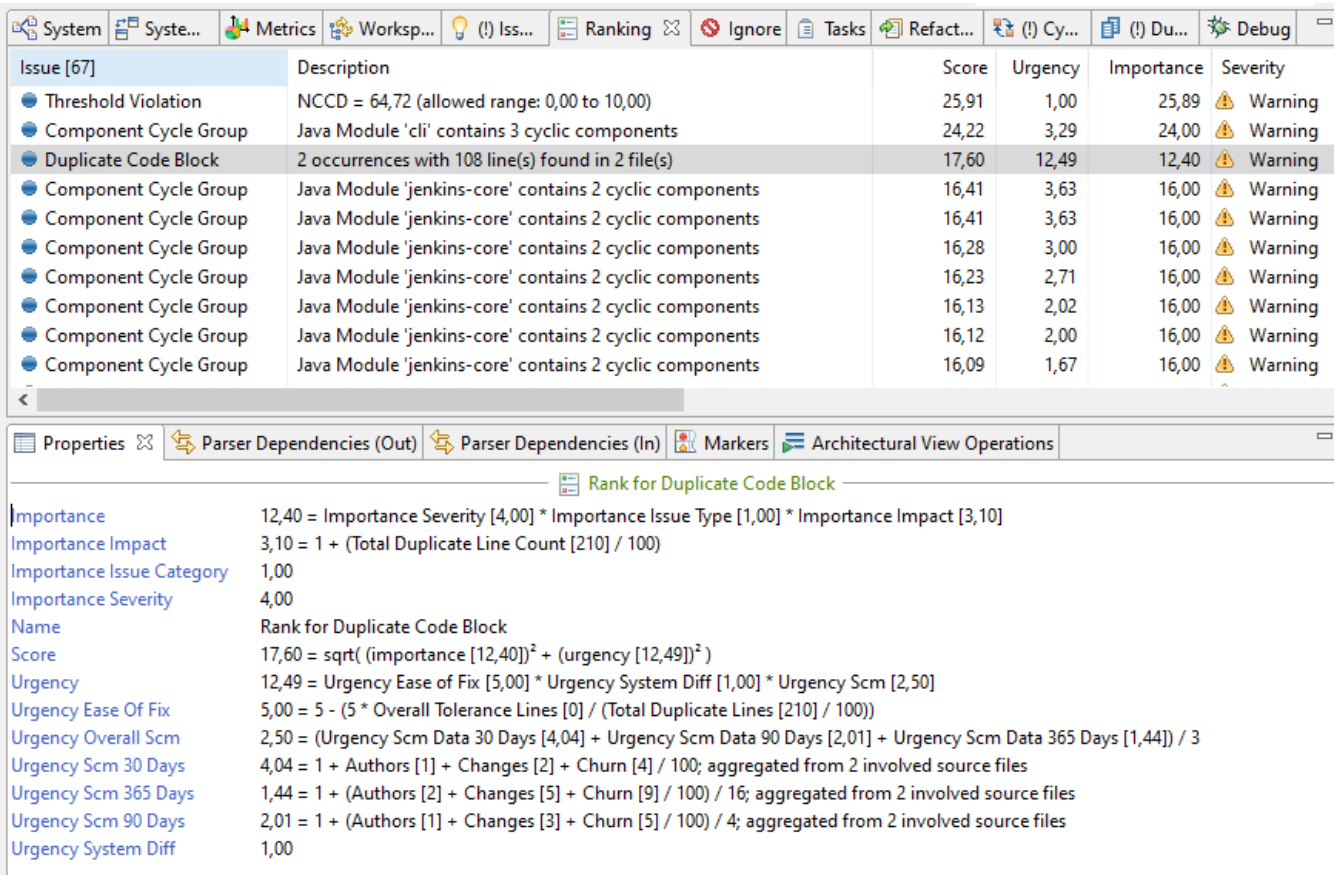


Figure 9.4. Ranking of Issues

9.2.2. Identifying Issue Hotspots

Since release 10.4.1 *Sonargraph* offers treemaps for visualizing the composition of a system with respect to its source files or components. Treemaps allow the easy identification of hotspots as shown in the screenshot below, whereby each file is represented by a square, the size of the square represents the size of the file. Green squares do not have issues, yellow have some, red squares contain many issues.

Figure 9.5. Issue Hotspots Treemap Visualization

Parent elements are represented by rectangles using grey color shades to indicate the nesting depth. The representation of leaf elements as squares makes it easy to spot relative size differences.

A new treemap configuration can be created via application menu "File" → "New" → "Other" → "New Treemap..." or via the context menu of the 'Treemap' folder on the Files view. The configuration of the treemap is currently focused on the type of leaf element ('Component' or 'Source File'), metric represented by square size ('Lines of Code' or 'Source Element Count') and the resolution type ('None', 'Ignore', 'Task'). The red threshold configures the value that will be the first to be represented with a red color. If set to '0' an even mapping of values to green, yellow and red is dynamically calculated.

TIP

When the option 'Link Master Views' in the top level toolbar is enabled, selecting a square/rectangle will reveal the corresponding underlying element in the master view.

TIP

The Properties view will show information about the corresponding underlying element of the selected square/rectangle.

TIP

Hovering over a square/rectangle will open a tooltip showing additional information. That tooltip can be focused by clicking into it with a left mouse click.

NOTE

Currently, no issues affecting elements above source file or component level are represented.

More functionality like filtering for specific issue types, categories, etc. will follow in upcoming releases.

See also Section 8.11.3, “Treemap-Based System Exploration” for more details about the Treemap functionality.

9.3. Ignoring Issues

The upper section of the Ignore view provides information about the type of issue the resolution resolves, the assigned priority for it, the assignee and an optional comment put in by the creator of the resolution among other data. The lower section displays the affected elements of the issues that match the selected "Ignore" resolution.

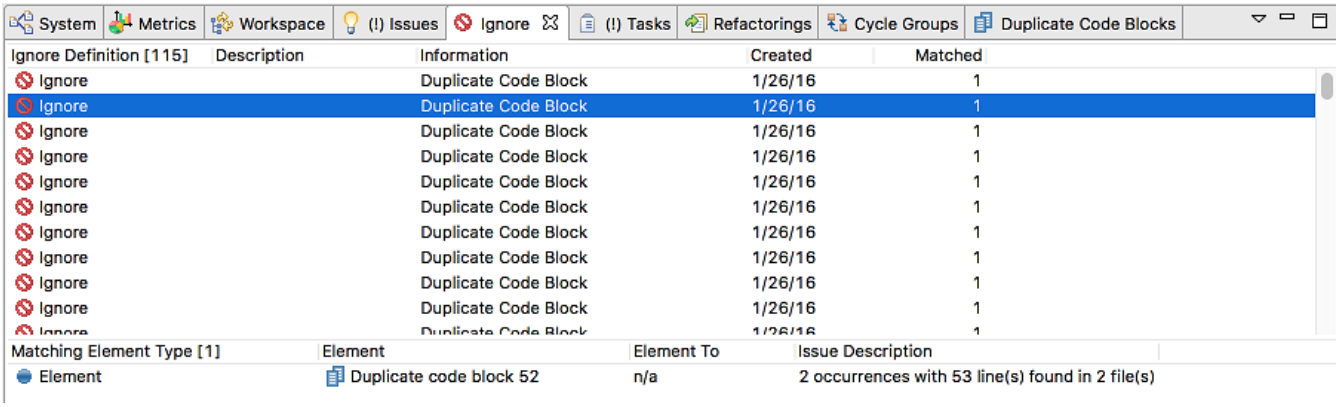


Figure 9.6. Ignore View

As mentioned previously, you can also focus the analysis on certain code regions by defining an *Issue Filter* as described in Section 8.8.1, “Definition of Filters, Modules and Root Directories”.

9.4. Defining Fix and TODO Tasks

"Fix" resolutions represent a proposed solution to one or many issues in the application. They are usually created to make sure that the related issues get eventually dealt with by someone in the development team.

Sonargraph allows to define specific "TODO" tasks for the different system elements. Those tasks can be assigned to a member of the development team to take care of. To define a TODO task, right click on the element and select "New TODO" in the context menu.

The "Fix" and "TODO" tasks are listed in the Tasks view.

9.5. Editing Resolutions

Using the context menu, you can modify the assignee, priority or comment of a resolution and even delete it if you deem it does not represent an appropriate solution anymore.

For some resolutions, right-clicking also offers the option of editing the element pattern of the resolution. The element pattern is an identifier of the element(s) a resolution is applied to:

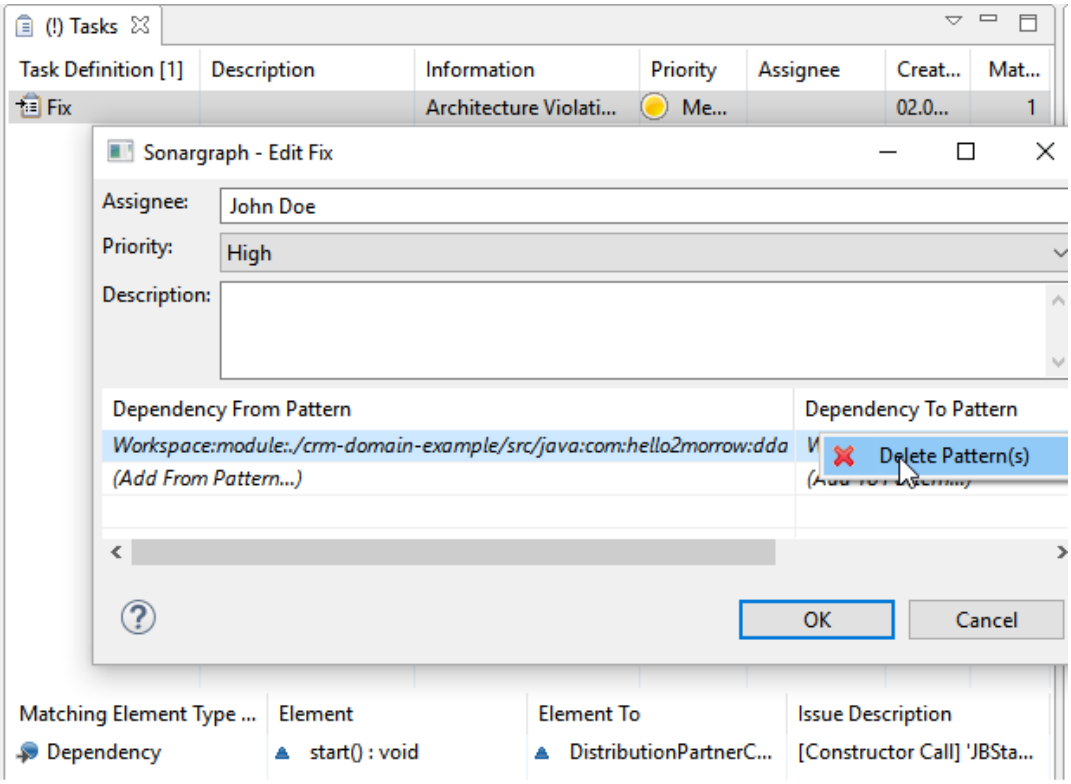


Figure 9.7. Resolution Dialog

The image above shows the element pattern matching mechanism between issues and resolutions. One resolution can be used to match several different elements via a wildcard pattern. This mechanism allows to group together in the same resolution (ignore or fix) current and new related issues as they are generated so to avoid the need to manually resolving each of them as they come about. It also helps when dealing with issues that should likely be taken care of together or by the same person.

You can have an idea of what the pattern for a specific element looks like by creating a resolution for the related issue and then looking at the element pattern section in the edit resolution dialog. You can create as many patterns for a resolution as you deem convenient.

9.6. Details about Sonargraph's Resolution Matching

Resolutions contain the fully qualified name of the affected element, so that they can be applied again when the System is cleared and parsed or opened. As a consequence resolutions are vulnerable against rename operations of directories and files. With Sonargraph version 10.5 advanced resolution matching has been introduced for cycle groups and duplicate code blocks, so that the resolutions are matched, even if the fully qualified names of involved elements got changed. The "confidence" of the resolution match is shown in the "Ignores" and "Tasks" views. Matching succeeds if the confidence is greater than 0.6. The same algorithms are also used by the "System Diff" to identify matching issues from the baseline.

Also with Sonargraph version 10.5, the Script API was improved with the data type ISourceLineAccess that provides access to file content and can be used to create issues that are more resilient against code changes by applying a similarity matching algorithm taking into account the line's text, the line number, and surrounding lines as context. The following program listing shows the key part of the script "FindFixmeAndTodosInComments.xml" (available in the "Core" quality model) that has been improved with the new methods of the Script API:

```
visitor.onSourceFile
{
    SourceFileAccess source ->
    List<ISourceLineAccess> lines = source.getSourceLines();
        for(SourceLineAccess line : lines)
        {
            def fixmeMatcher = (line.getText() =~ fixmePattern);
            if(fixmeMatcher.count > 0)
            {
                numberOfFixmes += fixmeMatcher.count;
                def text = extractText(fixmeMatcher);
                result.addWarningIssue(source, "FIXME", text, line);
            }
    ...
```

Using this API, resolutions are now only applied for the selected issue and no longer automatically for all "FIXME"-issues in the same file. "FIXME"-issues added later to the file need to be resolved separately offering a better control over new issues.

Chapter 10. Simulating Refactorings

Sonargraph allows the simulation of refactorings to quickly analyze different approaches to fix structural problems. Refactorings represent a proposed improvement to your code base. They are usually created to make sure that the related improvement is dealt with by someone in the development team, thus striving towards a healthy code base which is the ultimate goal of *Sonargraph*.

10.1. Creating Delete Refactorings

The *Delete* refactoring is available via "System" → "New Delete Refactoring..." or in the context menu when selecting an appropriate element.

A delete refactoring may be applied to the following (physical) elements (i.e. elements that come from the parsing process and are displayed in the Navigation view):

- Non-external programming elements (e.g. types, methods, fields)
- Non-external Directories (but not root directories)
- Non-external Namespaces
- Dependencies (parser level or aggregated)

When deleting parser level or aggregated dependencies there are up to 3 options. Their appearance, order and selected default option depend on the current context:

- Delete Parser Dependencies: Delete the currently contained parser dependencies of a given edge based on parser dependency patterns.
- Delete Parser Dependencies Based on Endpoints: Delete the parser dependencies of a given edge based on end point patterns, after the next 'refresh' there could be more or less matches.
- Delete Violating Parser Dependencies: Only delete the violating parser dependencies of a given edge based on parser dependency patterns.

Directories are always deleted recursively. Namespaces can be deleted flat or recursively. When deleting a type all its methods and fields or nested types are deleted.

Delete refactorings on (physical) namespaces may also be applied in the (logical) Namespaces view. Since a logical namespace (either in system or module scope) may be based on more than one physical namespace, the deletion of a logical namespace might delete several physical namespaces.

Delete refactorings may also be applied in the Architecture view.

Delete refactorings may also be applied in the Exploration, Graph and Dependencies View which are opened based on arbitrary Navigation, Namespace and Architecture view selections.

Delete refactorings can be managed in the Refactorings view as described in Section 10.3, “Managing Refactorings”.

10.2. Creating Move/Rename Refactorings

The *Move/Rename* refactoring is available via "System" → "New Move/Rename Refactoring..." or in the context menu when selecting an appropriate element.

A Move/Rename refactoring may be applied to the following (physical) elements (i.e. elements that come from the parsing process and are displayed in the Navigation view):

- Directories (but not root directories) for C# and C/C++
- Packages for Java

- Components for Java, C# and C/C++
- Physical top-level programming elements for Java, C# and C/C++ (e.g. types, free functions and global variables). 'Physical' means that logical top-level programming elements are not supported (i.e. types and so forth in logical views)!

Move/Rename refactorings may also be applied in the Architecture view.

Move/Rename refactorings may also be applied in the Exploration, Graph and Dependencies View when they are opened based on arbitrary Navigation and Architecture view selections.

10.3. Managing Refactorings

The upper section of the Refactorings view provides information about the type of refactoring, the provider, the applicability, the assigned priority for it, the assignee and an optional description.

The lower section displays the affected elements of the refactoring.

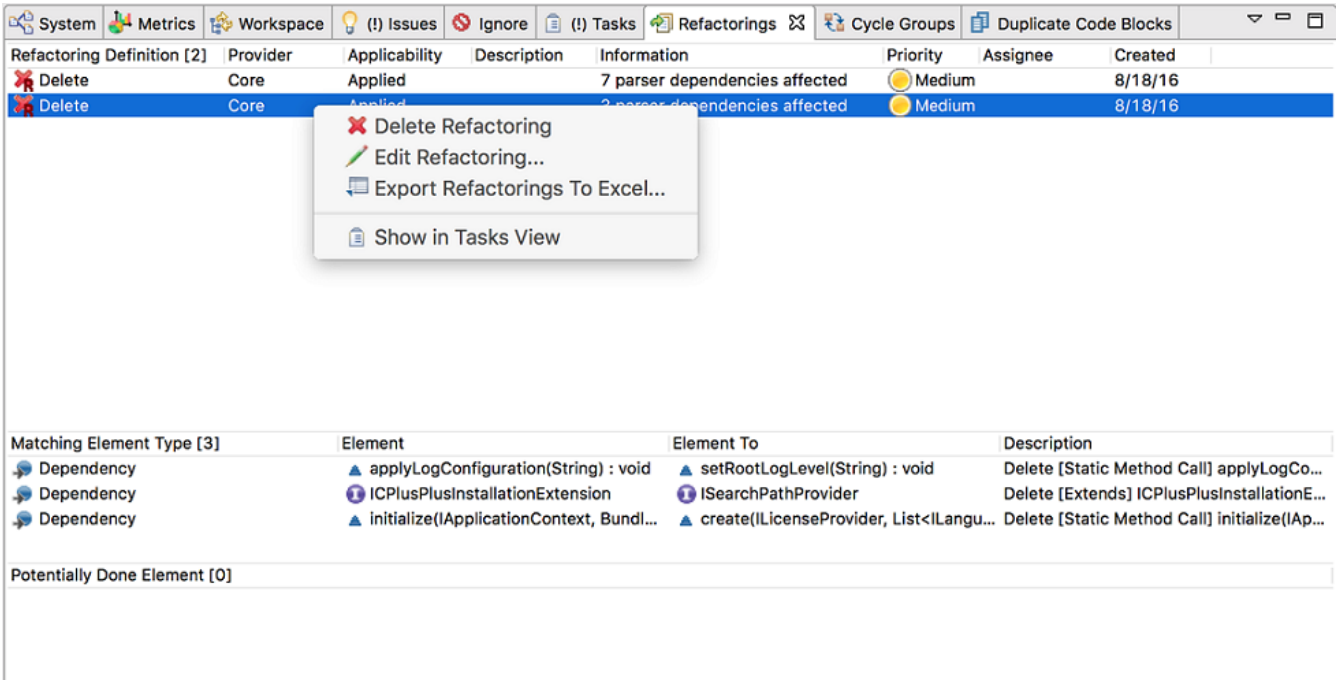


Figure 10.1. Refactorings View

The "Sonargraph Refactorings" view offers filter options in the top right corner. Refactorings can be filtered by status, priority, assignee and description.

10.4. Best Practices

The code base of a living software project changes fast, therefore we recommend the following approach to work with refactorings:

- **Do not get carried away and create hundreds of refactorings! It is better to "simulate a little, refactor a little".**

  Try to limit the impact of individual refactorings. Move and rename a package at the top of the hierarchy might have severe consequences on the code base and are most likely high-risk operations during implementation.

  If a package or class file gets renamed to a different name than specified in the refactorings, the refactorings are no longer applicable. There might be a chance in the future to semi-automatically update refactoring definitions based on the project's history, but we do not know when this will be implemented.

- **Work with only a few virtual models.**

  Note that in the IDE integrations, the standard "Modifiable.vm" is always applied and currently cannot be changed.

  Virtual models are great for experimenting with refactorings in isolation. But, since refactorings are not synchronized between virtual models, it is recommended to have one "main" model that contains approved refactorings and integrate the experiments as frequently as possible. If the same compilation unit is affected by refactoring sequences in different virtual models, implementing the refactorings of the first model will make the refactorings of the second model "inapplicable". We plan to improve the exchange of refactorings between virtual models in future versions.

Related topics:

- Section 20.1.7, “Execute Refactorings in Eclipse”

Chapter 11. Defining an Architecture

Sonargraph allows the definition of an architecture via a Domain Specific Language (DSL) that is expressive and readable enough so that every developer is able to understand it. The graphical representation in *Sonargraph 7* allowed the creation of the architectural blueprint in one single diagram. This led to potentially very big and complex diagrams that are difficult to understand.

The requirements for the new DSL approach were the following:

1. It should be possible to describe an architecture in a set of files. Some of them should be generic enough so that they could be reused by many projects, e.g. a generic template describing the layering of a system.

2. It should be possible to describe an architecture in form of several completely independent aspects. E.g. one aspect describes layering, another aspect describes components and a third aspect looks at separation of client and server logic.

3. On the other hand the language should also be powerful to describe the complete architecture in a single file.

4. The DSL must be easy to read and easy to learn.

5. The restrictions for dependencies should allow also the specification of dependency types (e.g. "new", "inheritance", etc.).

To create an architecture description you select "New Architecture File..." from the menu "File/New...". That will open an editor where you can work on your architecture description. You can have as many architecture files as you like. If the description should be used to check for architecture violations, the architecture file needs to be added to Sonargraph's architecture check. This is done in the "Files" tab of the "Navigation" view by right-clicking on your architecture file and select "Add to Architecture Check..." from the context menu. If you later decide to remove the file from Sonargraph's architecture check you can also do this via the context menu.

It is also recommended to open the "Architecture View" while working on an architectural model via the menu "Window" → "Show View" → "Architecture View". The view is split vertically into two main sections. In the top section there are three tabs to provide a quick overview about the checked files in the physical and logical model as well as which files are currently not part of the architecture check. These unchecked files might also include files that are imported by currently checked architectural files.

TIP

The context menu of a selected architecture model or artifact in the Architecture view offers the option to show the selection in the Exploration. This usually reveals very quickly where the architecture needs adjustment or where violations exist.

TIP

The context menu of a selected (and checked) architecture model also offers the option to show it as an UML Component diagram in an Architecture Diagram view (See Chapter 12, *Visualizing Architecture Aspects* for details).

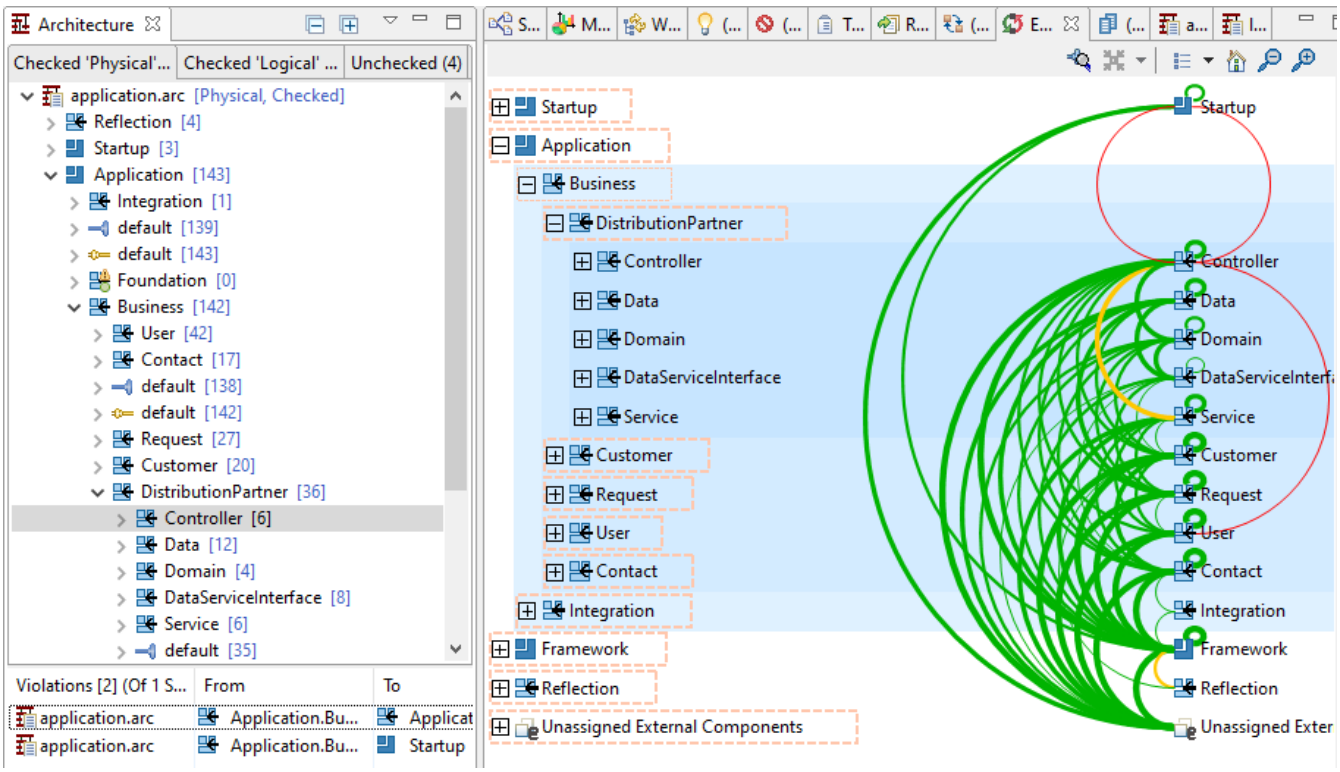


Figure 11.1. Architecture View

The tabs for logical and physical models contain architecture models that are actively checked. That means you will be able to see which elements are assigned to which artifact by browsing through the tree. You can also easily see which elements have not been assigned to any artifact by inspecting the nodes for "Unassigned internal/external components".

The bottom section lists all architecture violations of the element selected in the middle section. If no element is selected all architecture violations from all models are shown. If you click on a line in that table the associated violating dependencies are shown in the "Parser Dependencies Out" auxiliary view.

11.1. Models, Components and Artifacts

To describe architecture in a formal way we first need to think about the basic building blocks that we could use to describe the architecture of a system. The smallest unit of design is what we call a component. What is represented by a component depends on the base model you choose for you architecture.

Since version 9.7 Sonargraph supports two different base models. The "physical" model - which is the default model and the only model that was supported prior to 9.7 - is based on the model in the "Navigation View". Components are based on the physical layout of your project. In Java a component is a single source files. In C# a component is a single C# source file or a top level type in an external assembly. In C/C++ components are created dynamically out of combining associated header and source files.

The "logical" model is based on the model in the module based namespace view. Here our components are top level programming elements, which for Java or C# is always some type, usually a class or an interface. The logical model organizes these types only by their namespaces/packages. The directory structure of the project is not reflected in the model. In C/C++ components can also be functions or other top level programming elements. For Java there is almost no difference between the physical and the logical model. Only in the rare case that a Java file has more than one top level type the logical model would create one component for each top level type, while the physical model only generates one component per source file.

So logical models are more interesting for languages like C++ and C# where the namespace structure is not related to the physical organization of your project. For these languages it makes sense to use the logical model if your namespaces are in some way reflecting your architecture.

To define an architecture you would group associated components into artifacts. Then you could group several of those artifacts together into higher level artifacts and so on. For each artifact you would also define which other artifacts can be used by them.

Each component has a name which we call the architecture filter name. In the physical model the filter name starts with the module name or "External [language]". Then follows the path of the component relative to a module specific root directory. The filter name ends with the name of the source file without an extension, All name parts are separated by slashes.

TIP

To determine the architecture filter name of a component just click on the component in the navigation or namespace view and check the "Properties View". There you should be able to see the architecture filter name and other properties of the selected item.

When using a logical model the filter name again starts with the module name followed by the namespace followed by the name of the programming element. Each name part is again separated by slashes.

In most cases assignment of components to artifacts is based on their architecture filter name. But it is also possible to assign components based on other attributes like annotations or implemented interfaces. This will be explained in more detail later in this chapter.

```
// Main.java in package com.hello2morrow:
"Core/com/hello2morrow/Main"

// The Method class from java.lang.reflection:
"External [Java]/[Unknown]/java/lang/reflect/Method"

// SimpleAction.cs in subfolder of NHibernate:
"NHibernate/Action/SimpleAction"

// An external class from System.dll:
"External [C#]/System/System/Uri"
```

For internal components (components that actually belong to your project) we use the following naming strategy:

module/rel-path-to-project-root-dir/source-name (physical)

module/namespace-or-package/element-name (logical)

For external components (third party components used by your project) we use a slightly different strategy. Here we might not have access to any source files:

External [language]/jar-or-dll-if-present/rel-path-or-namespace/typename (physical)

External [language]/jar-or-dll-or-header/namespace-or-package/element-name (logical)

Now we can use patterns to describe groups of components:

```
// All components from the Core module with "business" in their name:
"Core/**/business/**"

// All components in java.lang.reflect:
"External*/*/java/lang/reflect/*"
```

As you can see a single '*' matches everything except a slash, '**' matches over slash boundaries. You can also use '?' as a wildcard for a single character.

Now we can build our first artifacts:

```
model "physical" // or "logical"

artifact Business
{
    include "Core/**/business/**"
    exclude "**/api/**"
}

artifact Reflection
{
    include "External*/*/java/lang/reflect/*"
}
```

In the first line you specify which model you would like to use. If you omit the model specification we assume "physical".

We grouped all components from module "Core" with "business" in their name into an artifact named "Business". As you can see the include patterns determine which components should belong to an artifact. You can also use "exclude" patterns to specify exceptions. The reflection classes from the Java runtime are now in their own artifact called "Reflection". Artifacts can also have "exclude" filters. They help you to describe the content of an artifact with an "everything except" strategy. Exclude filters will always be applied after all include filters.

TIP

More than one "include" statement can be used to assign components to an artifact. It is also possible to use "exclude" statements to specify exceptions from the elements included above.

The assignment of components to artifacts is usually determined by the order of artifacts in the DSL file. The principle is "first come. first served". If two patterns would match the same artifact the first pattern wins. It is however possible to assign priorities to artifacts which changes the order in which artifacts are assigned. The default priority of an artifact is 0. You can change the priority of an artifact with a priority definition, which always must be the first definition in an artifact. You can also negate patterns with the keyword 'not' or combine them with 'and'.

```
model "physical" // or "logical"

artifact Business
{
    include "Core/**/business/**"
}

artifact Reflection
{
    priority 1
    include "External*/*/java/lang/reflect/*"
}

artifact NotApiController
{
    include "**/controller/**" and not "**/api/**"
}
```

In the example above the artifact 'Reflection' would get the first pick at components since it has a higher priority than 'Business', even though it is defined after 'Business'. You can also assign negative priorities. That can be useful when you want to ensure that an artifact gets a lower priority than the default of 0. Just to be clear, if two artifacts have the same priority the order in the file determines which one picks first. The last artifact will match everything with 'controller' in the artifact name, unless it also contains 'api' in the name.

```
artifact Parent
{
    artifact FirstChild
    {
        include "**/c1/**"
    }

    artifact SecondChild
    {
        include "**/c2/**"
    }
}

artifact OtherParent
{
    include "**/other/**"

    artifact FirstChild
    {
        include "**/c1/**"
    }

    artifact SecondChild
    {
        include "**/c2/**"
    }
}
```

Artifacts can be nested arbitrarily. If a parent artifact does not have any include patterns of its own it becomes a 'transparent' artifact, i.e. it passes all components offered to it to its children for matching. 'Parent' is an example for a transparent artifact. 'OtherParent', on the other hand, defines its own include pattern. Now its children are only offered artifacts that are matched by the parent artifact, i.e. artifacts that contain 'other' somewhere in their component name. You can overrule this behavior by using 'strong' include patterns in the children artifacts. They are introduced in the next section.

TIP

Transparent artifacts can still have 'exclude' statements to limit the elements passed on to their children.

If a pattern has no matches, Sonargraph will put a warning marker on that pattern. You can suppress that warning by either making the artifact 'optional' or mark the pattern as 'optional'

11.1.1. Using other criteria to assign components to artifacts

Sometimes the information needed to properly assign a component to an artifact is not part of its architecture filter name. Imagine for example a code generator that generates classes for different functional modules. If all those classes end up in the same package it becomes very hard to assign the generated classes to the right functional modules unless the class name contains some clue. If those generated classes could be properly assigned based on an annotation that would be a far more effective method of assignment.

The following class shows a practical example:

```
package com.company.generated;

import com.company.FunctionalModule;

@FunctionalModule(name = "Customer")
class E5173
{
    // ....
}
```

Neither the class name nor the package name contain a clue that this class is associated with the functional module "Customer". Only the annotation gives that information away.

Since Sonargraph 9.7 it is possible to use what we call "attribute retrievers" in name patterns. In our example we would do the assignment as shown below:

```
artifact Customer
{
    include "JavaHasAnnotationValue: com.company.FunctionalModule: name: Customer"
}
```

If a search pattern contains colons it is split up into the parts separated by the colons (colons must be followed by a single space). The first part must be the name of an existing attribute retriever, in our example "JavaHasAnnotationValue". The last part is always a pattern describing what we would like to match and can make use of the wildcards "**", "*" and "?". Everything in between the first part and the last part are parameters for the retriever. Here we tell the retriever that we want to match with the "name" attribute of the annotation "com.company.FunctionalModule". Most retrievers don't need parameters, the example above is therefore already a pretty sophisticated use of attribute retrievers.

11.1.2. List of predefined attribute retrievers

Language Independent Retrievers

PhysicalFilterName

This retriever only works in the context of a logical model and will return the physical architecture filter name of a component. The component in this case would be a logical element, e.g. a class. The result is the architecture filter name of the physical component containing this element. Using this retriever allows you to mix physical and logical assignment strategies in a logical model.

WorkspaceFilterName

This retriever will return the workspace filter of any component. The workspace filter name is the relative path of the source file containing an element. This can be useful to separate assignment by root directory (e.g. test code versus generated code), since root directories are not part of any architecture filter name.

FileName

This retriever will return the file name of the component including extension. The path is the identifying path relative to the Sonargraph root directory. For external components an absolute path might be returned.

Java Attribute Retrievers

TIP

Java attribute retrievers require a fully qualified type name. The Properties view shows the "Name" property for a selected type and you can copy it from there as the fully qualified type name.

JavaHasAnnotation

This retriever only works for Java and will match if the pattern matches the fully qualified type name of any annotation of a class, its fields and its methods. In a physical model if a Java file has more than one top level type we only consider the Java class that has the same name as the file. Please note that "*" will match anything except dots (".") for this retriever.

```
artifact Controller
{
    include "JavaHasAnnotation: org.springframework.stereotype.Component"
    include "JavaHasAnnotation: **.Component"
}
```

The "include" statement of the this example will match any component with a top-level type that is annotated with "@Component" of the Spring framework. The second statement matches any component with a top-level type having an annotation ending with "Component".

JavaTypeOf

This retriever only works for Java and will match if the pattern matches the fully qualified type name of any direct or indirect super type (class or interface). In a physical model if a Java file has more than one top level type we only consider a Java type that has the same name as the file. Please note that "*" will match anything except dots (".") for this retriever.

JavaExtendsClass

This retriever only works for Java and will match if the pattern matches the fully qualified type name of any direct or indirect base class of a class. In a physical model if a Java file has more than one top level type we only consider the Java class that has the same name as the file. Please note that "*" will match anything except dots (".") for this retriever.

JavaImplementsInterface

This retriever only works for Java and will match if the pattern matches the fully qualified type name of any interface implemented by the class. In a physical model if a Java file has more than one top level type we only consider the Java class that has the same name as the file. Please note that "*" will match anything except dots (".") for this retriever.

JavaIsInterface

This retriever only works for Java and will match if the pattern matches the fully qualified type name of any interface. In a physical model if a Java file has more than one top level type we only consider the Java class that has the same name as the file. Please note that "*" will match anything except dots (".") for this retriever.

JavaIsClass

This retriever only works for Java and will match if the pattern matches the fully qualified type name of any class. In a physical model if a Java file has more than one top level type we only consider the Java class that has the same name as the file. Please note that "*" will match anything except dots (".") for this retriever.

JavaExtendsImplementsInterface

This retriever only works for Java and will match if the pattern matches the fully qualified Java name of any interface implemented by a class or extended by an interface. In a physical model only the Java main type (i.e. the type matching the component's name) is considered. This only works on 'internal' types. Please note that “*” will match anything except dots (".").

JavaHasAnnotationValue

This retriever only works for Java and will match if the pattern matches value of a specific annotation of a class. It has two parameters: the fully qualified Java name of the annotation class and the name of the annotation property to extract. In a physical model if a Java file has more than one top level type we only consider the Java class that has the same name as the file. Please note that "*" will match anything except dots (".") for this retriever.

C# Attribute Retrievers

TIP

C# attribute retrievers require a fully qualified type name. The Properties view shows the "Fully Qualified Type Name" property for a selected type and you can copy it from there.

CSharpTypeOf

This retriever only works for C# and will match if the pattern matches the fully qualified type name (namespace plus class name separated by ".") of any direct or indirect super type. In a physical model a C# file will only be considered if it contains a type that has the same name as the file. Please note that "*" will match anything except dots (".") for this retriever.

CSharpExtendsClass

This retriever only works for C# and will match if the pattern matches the fully qualified type name (namespace plus class name separated by ".") of any direct or indirect base class of a class. In a physical model a C# file will only be considered if it contains a type that has the same name as the file. Please note that "*" will match anything except dots (".") for this retriever.

CSharpImplementsInterface

This retriever only works for C# and will match if the pattern matches the fully qualified type name (namespace plus class name separated by ".") of any interface implemented by the class. In a physical model a C# file will only be considered if it contains a type that has the same name as the file. Please note that "*" will match anything except dots (".") for this retriever.

CSharpIsInterface

This retriever only works for C# and will match if the pattern matches the fully qualified type name (namespace plus class name separated by ".") of any interface. In a physical model a C# file will only be considered if it contains a type that has the same name as the file. Please note that "*" will match anything except dots (".") for this retriever.

CSharpIsClass

This retriever only works for C# and will match if the pattern matches the fully qualified type name (namespace plus class name separated by ".") of any class. In a physical model a C# file will only be considered if it contains a type that has the same name as the file. Please note that "*" will match anything except dots (".") for this retriever.

CSharpIsEnum

This retriever only works for C# and will match if the pattern matches the fully qualified type name (namespace plus class name separated by ".") of any enum. In a physical model a C# file will only be considered if it contains a type that has the same name as the file. Please note that "*" will match anything except dots (".") for this retriever.

C/C++ Attribute Retrievers

CppExtendsClass

> ### TIP
>
> This attribute retriever requires a fully qualified type name. The Properties view shows the "Fully Qualified Type Name" property for a selected type and you can copy it from there.

This retriever only works for C++ and will match if the pattern matches the fully qualified type name (namespace plus class name with "." as separator) of any direct or indirect base class of a class. In a physical model a C++ component will only be considered if it contains a type that has the same name as the component. Please note that "*" will match anything except dots (".") for this retriever.

CppHeaderPath

This retriever only works for physical models in C and C++ and will match if the pattern matches the identifying path of the main header file of a component. The main header is the header file that has the name of the component, while the identifying path is the relative path of the header relative to the Sonargraph system directory. Use this retriever if the location of a header file is more relevant for the architecture than the source file location.

TypeScript Attribute Retrievers

TypescriptTypeof

This retriever only works for TypeScript and will match if the pattern matches the fully qualified type name (separated by ".") of any direct or indirect super type. Since TypeScript does not have a logical model, a component (always a source file) contains any type with a base type matching the pattern.

TypescriptExtendsClass

This retriever only works for TypeScript and will match if the pattern matches the fully qualified type name (separated by ".") of any direct or indirect base class of a class. Since TypeScript does not have a logical model, a component (always a source file) contains any type with a base type matching the pattern.

TypescriptImplementsInterface

This retriever only works for TypeScript and will match if the pattern matches the fully qualified type name (separated by ".") of any interface implemented by the class or extended by the interface. Since TypeScript does not have a logical model, a component (always a source file) contains any type with a base type matching the pattern.

11.2. Interfaces and Connectors

To define allowed relationships between artifacts it helps to use some simple and effective abstractions. Lets assume every artifact has at least one incoming and one outgoing named port. Artifacts can connect to other artifacts by connecting an outgoing port with an incoming port of another artifact. We will call outgoing ports "Connectors" and incoming ports "Interfaces". By default each artifact always has an implicit connector called "default" and an implicit interface also called "default". Those implicit ports always contain all the elements contained in an artifact, unless redefined by the architect.

Let us now connect our artifacts:

```
artifact Business
{
    include "Core/**/business/**"
    connect default to Reflection.default
}

artifact Reflection
{
    include "External*/*/java/lang/reflect/*"
}
```

This will allow all elements contained in "Business" use all elements contained in "Reflection" by connecting the default connector of "Business" with the default interface of "Reflection". In our architecture DSL you can also write this shorter:

```
artifact Business
{
    // ...
    connect to Reflection
}

// ...
```

If we reference an artifact without explicitly naming a connector or an interface the language will assume that you mean the default connector or interface. Connections can only be established between connectors and interfaces. The syntax of the connect feature is as follows:

connect [connectorName] to interfaceList

The interface list is a comma separated list of interfaces to connect to. The connector can be omitted, in that case the default connector will be used.

A dependency from a component A to another component B is not an architecture violation if any of the following conditions is true:

- Either A and/or B do not belong to any artifact.
- A and B belong to the same artifact.
- The artifact of B is nested in the artifact of A.
- There is an explicit connection from a connector that contains A to an interface that contains B.
- B belongs to the default interface of a "public" artifact that is a sibling of a artifact that has a default connector containing A. The artifact of A must be defined before the artifact of B. In other words, "public" artifacts are accessible by sibling artifacts defined above them. ("public" will be introduced later)
- The artifact of A or one of its parent artifacts is "unrestricted" and B is assigned directly or indirectly to a sibling of the unrestricted artifact. In other words, unrestricted artifacts have access to all of their siblings. ("unrestricted" will also be introduced later)

- The artifact of A or one of its parent artifacts is "strict" (i.e. is a strict layer) and B is assigned directly or indirectly to the next following sibling of the strict artifact. More precisely: A must be part of the default connector of the strict artifact, while B must be part of the default interface of its next sibling. Strict layers are allowed to access the layer (artifact) directly below them.

- The artifact of A or one of its parent artifacts is "relaxed" (i.e. is a relaxed layer) and B is assigned directly or indirectly to the any of the siblings of the relaxed artifact that are defined after it. More precisely: A must be part of the default connector of the relaxed artifact, while B must be part of the default interface of any of its siblings that are defined after it. Relaxed layers are allowed to access all the layers (artifacts) defined below them.

Any dependency that does not meet any of the above conditions is considered to be an architecture violation.

"strict", "relaxed" and "unrestricted" are mutually exclusive, i.e. an artifact can have at most one of those three stereotypes.

Now let us assume that we would not want anybody to use the class "Method" of the reflection artifact. This can be achieved by redefining the default interface of "Reflection":

```
artifact Reflection
{
    include "**/java/lang/reflect/*"

    interface default
    {
        include all
        exclude "**/Method"
    }
}
```

Doing that makes it impossible to access the *Method* class from outside the "Reflection" artifact because it is not part of any interface. Here we used an include all filter to add all elements in "Reflection" to the interface. Then by using an exclude filter we took out *Method* from the set of accessible elements in the interface.

Most of the time you will not need to define your own connectors. This is only necessary if you want to exclude certain elements of the using artifact from accessing the used artifact. Using more than one interface on the other hand can be quite useful. But for the sake of completeness let us also define a connector in "Business":

```
artifact Business
{
    include "Core/**/business/**"

    connector CanUseReflection
    {
        // Only include the controller classes in Business
        include "**/controller/**"
    }

    connect CanUseReflection to Reflection
}

// ...
```

Now only classes having "business" and "controller" in their name will be able to access "Reflection".

Let us do something more advanced and assume that the architect wants to make sure that "Reflection" can only be used from elements in the "Business" layer. To achieve that we can simply nest "Reflection" within the "Business" artifact and hide it from the outside world:

```
artifact Business
{
    include "Core/**/business/**"

    hidden artifact Reflection
    {
        // Need a strong pattern to bypass patterns defined by parent artifact
        strong include "**/java/lang/reflect/*"
    }
}
```

By declaring a nested artifact as "hidden" it will be excluded from the default interface of the surrounding artifact. We also don't need to connect anything because parent artifacts always have full access to the artifacts nested within them. In general an artifact can access anything that belongs to itself including nested artifacts and all components that are not part of any artifact. Access to other artifacts requires an explicit connection.

Notice the strong include pattern. Without using a strong pattern the elements belonging to reflection would not make it past the pattern filters defined by "Business".

You can also use the "local" modifier for artifacts. A local artifact will not be part of the default connector of the surrounding artifact.

If you later find out that another part of your software needs access to "Reflection" too you have several options. You could add an interface to "Business" exposing "Reflection" or you could again make a top level artifact out of it. Here is how you'd expose it:

```
artifact Business
{
    include "Core/**/business/**"

    hidden artifact Reflection
    {
        // Need a strong pattern to bypass patterns defined by parent artifact
        strong include "External*/*/java/lang/reflect/*"
    }

    interface Refl
    {
        export Reflection
    }
}
```

With *export* you can include nested artifacts or interfaces of nested artifacts in an interface. Now clients can connect to the "Business.Refl". The counterpart of *export* for connectors is the keyword *include*. It will include nested artifacts or connectors from nested artifacts in a connector.

In that particular example we can expose "Reflection" even more easily:

```
artifact Business
{
    include "Core/**/business/**"

    exposed hidden artifact Reflection
    {
        // Need a strong pattern to bypass patterns defined by parent artifact
        strong include "External*/*/java/lang/reflect/*"
    }
}
```

Now that looks a little strange on first sight, doesn't it - "exposed" and "hidden" at the same time? Well, "hidden" will exclude "Reflection" from the default interface of "Business", while "exposed" makes it visible to clients of "Business". Now clients can

connect to "Business.Reflection" which is a shortcut for "Business.Reflection.default". If "Reflection" had more interfaces they could also connect to those other interfaces.

That brings us to another important aspect of our architecture DSL - encapsulation. An artifact only exposes its interfaces or the interfaces of exposed artifacts to its clients. It is not possible for a client to connect to a nested artifact until it is explicitly exposed by its surrounding artifact.

export and *include* can be used together with the keyword *any*. The following example shows how you could explicitly define the default interface and the default connector of any artifact:

```
artifact SomeArtifact
{
    include "**/something/**"

    hidden artifact Hidden
    {
        // ...
    }

    local artifact Local
    {
        // ...
    }

    artifact Nested
    {
        // ...
    }

    interface default
    {
        include "**"
        export any    // will export 'Local.default' and 'Nested.default'
    }

    connector default
    {
        include "**"
        include any   // will include 'Hidden.default' and 'Nested.default'
    }
}
```

If you use *any* by itself it will include all nested artifacts except hidden artifacts for *export* and local artifacts for *include*. You can also explicitly name an interface or a connector of a nested artifact after *any*. In that case the interface or connector is included if it exists, even if its artifact is marked as hidden or local (see next example).

```
artifact SomeArtifact
{
    include "**/something/**"

    hidden artifact Hidden
    {
        // ...
        interface UI { /* ... */ }
    }

    artifact Nested
    {
        // ...
        interface UI { /* ... */ }
    }

    interface default
    {
        export any.UI    // will export 'Hidden.UI' and 'Nested.UI'
    }
}
```

This feature can become quite useful if there are many nested artifacts with a similar structure.

We mentioned before that an artifact can have the modifier *unrestricted*. This means that dependencies coming out of such an artifact to any of its siblings will not be checked. That can be useful if you are creating an architecture description for an existing system with many violations. By declaring some artifacts as *unrestricted* you are not being overwhelmed by violations and can focus on the most important violations first. It is also useful for grouping legacy code that you want to exclude form architecture checks.

```
strict artifact SomeArtifact
{
    include "**/something/**"
}
strict artifact OtherArtifact
{
    include "**/other/**"
}
unrestricted artifact Legacy
{
    // All remaining internal components
    include "**"
    exclude "External*/**"
}
```

In the example above the two artifacts above "Legacy" have clear architecture rules. They are both defined as strict layers, i.e. they have access to the artifact defined directly below them. All remaining internal components are assigned to "Legacy". Since "Legacy" is unrestricted, its dependencies towards its siblings are not checked. That can be quite useful when you start defining an architecture for an existing system and only want to focus on certain parts of the system. Just keeping components unassigned would have a slightly different effect. I our example we do not allow dependencies from "SomeArtifact" to "Legacy" because we have defined "SomeArtifact" as a strict layer. That restriction could not be checked if we had kept the components in "Legacy" unassigned.

Here is a summary of the different stereotypes that can be used on artifacts:

Stereotype	**Description**
hidden	The artifact will not be included in its parents default interface.
local	The artifact will not be included in its parents default connector.
public	All sibling artifacts defined above this artifact can implicitly access the default interface from this artifact using their default connector.

Stereotype	Description
unrestricted	All elements of this artifact can freely access the default interfaces of all the siblings of this artifact.
strict	Creates an implicit connection from the default connector of this artifact to the default interface of its next sibling. (strict layering)
relaxed	Creates implicit connections from the default connector of this artifact to the default interfaces of all sibling artifacts defined after this artifact. (relaxed layering)
exposed	Makes this artifact visible to clients of its parent.
optional	Don't warn if this artifact has no components assigned to it.
deprecated	Do create a warning if any components are assigned to this artifact.

Table 11.1. Artifact stereotype summary

Additionally, in order to ease the visualization of the different stereotypes that can modify the behavior of an artifact, *Sonargraph* uses the following icons and/or decorators:

		via "apply"	via "require"	public	local	hidden
artifact						
unrestricted artifact						
strict artifact						
relaxed artifact						

Table 11.2. Icons/Decorators for Artifacts

Note that artifacts via "apply" or "require" can also have decorators for public, local and hidden stereotypes.

At the end of this section let us have a look at the general syntactic structure of artifacts, interfaces and connectors:

```
artifact name
{
    // include and exclude filters
    // nested artifacts
    // interfaces and connectors
    // connections
}

interface iname
{
    // include and exclude filter
    // exported nested interfaces
}

connector cname
{
    // include and exclude filters
    // included nested connectors
}
```

The order of the different sections is important. Not following this particular order will lead to syntax errors.

Now that we have covered the basic building blocks we can progress to more advanced aspects. In the next section I will focus on how to factor out reusable parts of an architecture into separate files that can best be described as *Architecture Aspects*. We will also cover the restriction of dependencies by dependency types.

11.3. Reusing Architecture Aspects

Let us assume we want to use a predefined layering for several modules of our software system. Without a mechanism for reuse we would have to write something like that:

```
artifact Module1
{
    include "Module1/**"

    artifact UI
    {
        include "**/ui/**"
        connect to Business
    }
    artifact Business
    {
        include "**/business/**"
        connect to Persistence
    }
    artifact Persistence
    {
        include "**/persistence/**"
    }
    public artifact Model
    {
        include "**/model/**"
    }
    interface Service
    {
        export Business, Model
    }
}

artifact Module2
{
    include "Module2/**"

    artifact UI
    {
        include "**/ui/**"
        connect to Business
    }
    artifact Business
    {
        include "**/business/**"
        connect to Persistence
    }
    artifact Persistence
    {
        include "**/persistence/**"
    }
    public artifact Model
    {
        include "**/model/**"
    }
    interface Service
    {
        export Business, Model
    }
}
```

As you can see the inner structure of both modules is completely identical. Now imagine having dozens of modules. We clearly need a better way to model that. That is where the apply directive (see below) comes into the game that allows splitting the architecture into several *architecture aspects*, each contained in its own file.

We also introduced a new artifact modifier on the fly: *public*. All artifacts marked as public can be used by all non-public artifacts on the same level (siblings in the artifact tree). "UI", "Business" and "Persistence" therefore have an implicit connection to "Model" (from default connector to default interface).

```
// File layering.arc
artifact UI
{
    include "**/ui/**"
    connect to Business
}
artifact Business
{
    include "**/business/**"
    connect to Persistence
}
artifact Persistence
{
    include "**/persistence/**"
}
public artifact Model
{
    include "**/model/**"
}
// Top level interfaces only make sense, when used together with "apply" (see below)
interface Service
{
    export Business, Model
}

// New file modules.arc
artifact Module1
{
    include "Module1/**"

    apply "layering"
}

artifact Module2
{
    include "Module2/**"

    apply "layering"
}
```

Now we only have to describe the inner structure of modules in one separate file and apply this structure to them using the apply directive. That is a very powerful construct that will enable you to define reusable patterns.

Let us introduce two additional artifact modifiers that can be useful in certain situations: "optional" is used for artifacts defined within an aspect that could potentially be empty. Using "optional" will suppress the warning marker that is attached to artifacts that have no components assigned to them.

"deprecated" works the other way around. Artifacts declared as "deprecated" will get a warning marker if they have components assigned. That features is very useful to catch components that are not named correctly. The next example will show both modifiers in action:

```
// File layering.arc
artifact UI
{
    include "**/ui/**"
    connect to Business
}
artifact Business
{
    include "**/business/**"
    connect to Persistence
}
artifact Persistence
{
    include "**/persistence/**"
}
public artifact Model
{
    include "**/model/**"
    connect to Util // since Model is public this is required
}
optional public artifact Util
{
     include "**/util/**"
}
deprecated artifact Deplorables
{
    include "**"
}
// Top level interfaces only make sense, when used together with "apply" (see below)
interface Service
{
    export Business, Model
}
```

We added two more artifacts. "Util" is for utility classes that might or might not be present. That is why we added the "optional" modifier. "Util" is also "public"" so that all non-public sibling artifact can use the utility classes implicitly. Since "Model" is also declared to be "public" we need to make an explicit connection to "Util" if we want "Model" to have access to "Util".

The artifact "Deplorables" catches all remaining components that are assigned to the surrounding artifact. Note that the order of artifacts is critical in this case. "**" matches everything, so if we would move "Deplorables" to the top of the artifact list it would get all available components assigned. At the end of the list it will only get those components that have not been assigned to the artifacts above. If we did not have the "Deplorables" artifact those would usually stay assigned to the parent artifact or stay unassigned if there is no parent artifact.

So, having an unconnected deprecated artifact like "Deplorables" is useful for several reasons:

- It catches all components that are not properly named.
- Usually it is desirable that parent artifacts distribute all their components among their children and do not keep components to themselves. This is achieved by using the "**" pattern in "Deplorables".
- If there are components that are not properly named the artifact will get a warning marker and all dependencies to those components are marked as architecture violations.

11.4. Extending Aspect Based Artifacts

Now let us assume we want to refactor one of our modules to have an extra layer. We cannot do this change in the aspect file because this would apply to all modules. If we still want to be able to use the aspect for this module we need some way to extend or modify the elements in the aspect file:

```
artifact Module2
{
    include "Module2/**"

    apply "layering"

    // New layer
    artifact BusinessInterface
    {
        include "**/businessinterface/**"
    }

    // Now Business and UI need access to BusinessInterface
    extend Business
    {
        connect to BusinessInterface
    }
    extend UI
    {
        connect to BusinessInterface
        // UI should not use Business directly
        disconnect from Business
    }
}
```

Extending an artifact only makes sense in the context of apply directives. It allows us to add nested elements to an artifact and/or modify its connections to other artifacts. Within an extended artifact you can also use the keyword override to override the definitions of interfaces or connectors defined in the original version of the artifact:

```
artifact Module2
{
    // ...
    extend Business
    {
        // This assumes that the imported version of Business has an interface named "X"
        override interface X
        {
            // Use other patterns or other exports
            include "**/x/*"
        }
        connect to BusinessInterface
    }
    // ...
}
```

This allows you to adapt the architecture elements derived from an aspect file when needed.

11.5. Extending Interfaces or Connectors

It is also possible to extend interfaces or connectors defined via apply. That is sometimes quite useful as shown in the following example:

```
artifact Module2
{
    include "Module2/**"

    apply "layering"

    // New artifact that should also be part of the service interface
    artifact ServiceInterface
    {
        include "**/serviceinterface/**"
    }

    // Make the ServiceInterface artifact part of the interface Service
    extend interface Service
    {
        // add an extra export
        export ServiceInterface
    }
}
```

Without the possibility to extend interfaces or connectors we would be forced to create a completely new interface with a different name.

11.6. Adding Transitive or Deprecated Connections

Transitive dependencies are a useful addition to formal architecture descriptions. The following example shows a typical use case:

```
artifact Controller
{
    include "**/controller/**"
    connect to Foundation
}

artifact Foundation
{
    include "**/foundation/**"
}
```

Here *Controller* depends on *Foundation*. We also assume that classes from *Foundation* are used in the public interface of the controller classes. That means that each client of *Controller* must also be able to access *Foundation*.

```
artifact ControllerClient
{
    include "**/client/**"
    connect to Controller, Foundation
}
```

This is certainly not ideal because it requires the knowledge that everything that uses the *Controller* artifact must also connect to *Foundation*. It would be better if that could be automized, i.e. if anything connects to *Controller* it will automatically be connected to *Foundation* too.

Using transitive connections this is easy to implement:

```
artifact ControllerClient
{
    include "**/client/**"
    connect to Controller // No need to connect to Foundation explicitly
}

artifact Controller
{
    include "**/controller/**"
    connect to Foundation transitively
}
// ...
```

Using the new keyword *transitively* in the connect statement will add *Foundation* to the default interface of *Controller*. That means that anybody connecting to the default interface of *Controller* will also have access to *Foundation* without needing an explicit dependency.

The new keyword only influences the default interface. For explicitly defined interfaces the transitive export also has to be made explicit:

```
artifact ControllerClient
{
    include "**/client/**"
    connect to Controller.Service // Will also have access to Foundation
}

artifact Controller
{
    include "**/controller/**"

    interface Service
    {
       include "**/service/**"
       export Foundation // Transitive connection must be explicit here
    }

    connect to Foundation transitively // only affects default interface
}
// ...
```

Before we had transitive connections an interface could only export nested artifacts. Now interfaces can also export connected interfaces. In the example above we add the default interface of *Foundation* to the *Service* interface of *Controller*. Exporting interfaces that are not a connection of the parent artifact will cause an error message.

Deprecated dependencies are used when to warn about dependencies that are tolerated for now, but should be removed from the code. Instead of producing architecture violation errors they produce deprecation warnings on dependencies. The following example shows a typical use case:

```
artifact Controller
{
 // ...
 connect to Other deprecated // All dependencies to "Other" will now produce warnings
}

artifact Other
{
 // ...
}
```

You can add "deprecated" at the end of most "connect to" statements.

11.7. Restricting Dependency Types

Sometimes you are in a situation, where you allow one artifact to use another one, but would like to restrict the usage to dependencies of a certain type. For example let us assume you do not want the UI layer to create new instances of classes defined in the "Model" layer. Only "Business" and "Persistence" would be allowed to create "Model" instances. You can solve this by creating a new interface that restricts the usage of certain dependency types:

```
artifact UI
{
    include "**/ui/**"
    connect to Business, Model.UI
}
artifact Business
{
    include "**/business/**"
    connect to Persistence, Model
}
artifact Persistence
{
    include "**/persistence/**"
    connect to Model
}
artifact Model
{
    include "**/model/**"
    interface UI
    {
        include all // everything in "Model"
        exclude dependency-types NEW
    }
}
```

Now it would be marked as an architecture violation if a class from the UI layer would create a new instance of an object from the model layer. Please note that we had to remove the public modifier from "Model". If we had kept it there would have been an implicit connection from UI to the default interface of Model bypassing our special restriction.

Currently the language supports the following list of language agnostic abstract dependency types:

```
        // instance creation
        NEW

        // inheritance
        EXTENDS

        // interface implementation
        IMPLEMENTS

        // function or method calls
        CALL

        // reading a field or variable
        READ

        // writing to a field or variable
        WRITE

        // all other uses
        USES
```

In the next section we will look at another advanced concept called "connection schemes".

11.8. Connecting Complex Artifacts

In this section we will examine the different possibilities to define connections between complex artifacts. Let us assume we use the following aspect file to describe the inner structure of a business module:

```
// File layering.arc
exposed artifact UI
{
    include "**/ui/**"
    connect to Business
}
exposed artifact Business
{
    include "**/business/**"

    interface default
    {
        // Only classes in the "iface" package can be used from outside
        include "**/iface/*"
    }

    connect to Persistence
}
artifact Persistence
{
    include "**/persistence/**"
}
exposed public artifact Model
{
    include "**/model/**"
}
```

This example also show a special feature of our DSL. You can redefine the *default* interface if you want to restrict incoming dependencies to a subset of the elements assigned to an artifact. Our layer "Business" is now only accessible over the classes in the "iface" package.

Now lets bring in some business modules:

```
// File modules.arc
artifact Customer
{
    include "Customer/**" // All in module "Customer"
    apply "layering"
    connect to Core
}
artifact Product
{
    include "Product/**" // All in module "Product"
    apply "layering"
    connect to Core
}
artifact Core
{
    include "Core/**" // All in module "Core"
    apply "layering"
}
```

Here "Customer" and "Product" are connected to "Core". We used the most simple way to connect those artifacts which means that all elements in "Customer" or "Product" can use everything in the default interface of "Core". Since we redefined the default interface of "Business" this is not everything in "Core". The default interface of "Core" exports all default interfaces of non-hidden nested artifacts which means that the restrictions defined in "Business" are respected by surrounding artifacts.

Nevertheless this way of connecting artifacts does not give us enough control. For example "Product.Model" could now access "Core.UI" - not pretty. That means we need to put a bit more effort into the connection:

```
// File modules.arc
artifact Customer
{
    include "Customer/**" // All in module "Customer"
    apply "layering"

    connect UI to Core.UI, Core.Controller, Core.Model
    connect Controller to Core.Controller, Core.Model
    connect Model to Core.Model
}
artifact Product
{
    include "Product/**" // All in module "Product"
    apply "layering"

    connect UI to Core.UI, Core.Controller, Core.Model
    connect Controller to Core.Controller, Core.Model
    connect Model to Core.Model
}
artifact Core
{
    include "Core/**" // All in module "Core"
    apply "layering"
}
```

Now we are more specific about the details of our connection. Please note that we can only connect to "UI", "Controller" and "Model" of "Core" because we have marked those artifacts as exposed. Otherwise they would be encapsulated and not directly accessible. The "Persistence" layer is not exposed and can therefore only be used from inside its enclosing artifact.

11.9. Introducing Connection Schemes

If you look closely you will find that both connection blocks in "Customer" and "Product" are absolutely identical. Now image you had to connect dozens of artifacts in this way. That would be quite annoying and error prone. To avoid this kind of duplication we added the concept of connections schemes:

```
// File modules.arc
connection-scheme C2C
{
    connect UI to target.UI, target.Controller, target.Model
    connect Controller to target.Controller, target.Model
    connect Model to target.Model
}

artifact Customer
{
    include "Customer/**" // All in module "Customer"
    apply "layering"

    connect to Core using C2C // connection scheme C2C
}
artifact Product
{
    include "Product/**" // All in module "Product"
    apply "layering"

    connect to Core using C2C
}
artifact Core
{
    include "Core/**" // All in module "Core"
    apply "layering"
}
```

Now I hope you agree that this is cool. Using connection schemes it becomes possible to describe the wiring between artifacts in an abstract way. That makes it easy to change the wiring if the architect comes up with a new idea or wants to add or remove restrictions.

In big systems you may need some additional nesting to avoid having too many toplevel artifacts:

```
artifact SystemPartA
{
    //...
    artifact A
    {
        //...
        apply "layering"
    }

    artifact B
    {
        //...
        apply "layering"
    }

    connect to SystemPartB using Part2Part
}

arifact SystemPartB
{
    //...
    artifact C
    {
        //...
        apply "layering"
    }

    artifact D
    {
        //...
        apply "layering"
    }
}

connection-scheme Part2Part
{
    // Please note the use of "any"
    connect any.UI to target.any.UI, target.any.Controller, target.any.Model
    connect any.Controller to target.any.Controller, target.any.Model
    connect any.Model to target.any.Model
}
```

Here parts contain nested parts which share a common layering. The use of the keyword *any* allows to insert a wildcard for those nested parts into the scheme. In our example each wildcard connection defined in the scheme would result in 4 real connections since each part has 2 nested parts here (A.x to C.x, A.x to D.x, B.x to C.x and B.x to D.x). To keep the number of connections under control only one *any* is allowed on each side of a wildcard connection.

11.10. Artifact Classes

Artifact classes have been added as an optional and advanced feature that can be really useful in larger projects or in conjunction with connection schemes. An artifact class basically names the connectors and interfaces an artifact is supposed to have. If an artifact is declared to have a specific class Sonargraph will verify that it defines all the interfaces and connectors required by the class. Moreover connection schemes can now also define source and target classes which allows immediate checking of correctness.

Another benefit is that artifact classes make it a lot easier to organize artifacts into a tree so that the number of top-level artifacts stays manageable.

Let us introduce a real example:

```
// File "layering.arc"
artifact Service
{
    // ...
    connect to Controller
}
artifact Controller
{
    // ...
    connect to DataAccess
}
artifact DataAccess
{
    // ...
}
public exposed artifact Model
{
    // ...
}
interface IService
{
    export Service, Model
}

// Main file "business.arc"
class BusinessComponent
{
    interface IService, Model
    connector Controller, Model
}

connection-scheme BC2BC : BusinessComponent to BusinessComponent
{
    connect Controller to target.IService
    connect Model to target.Model
}

artifact Customer : BusinessComponent
{
    apply "layering"
}

artifact Order : BusinessComponent
{
    apply "layering"

    connect to Customer using BC2BC
}
```

The artifacts "Customer" and "Product" are specifying "BusinessComponent" as their artifact class. Therefore they must have "IService" and "Model" either as an interface or as an exposed artifact. They also must have connectors or artifacts named "Controller" and "Model". In our example the artifacts conform to the class. Otherwise Sonargraph would report an error.

The advantage of using artifact classes together with connection schemes: Now we can check the connection scheme for correctness at the point of definition. Without the use of classes we can only do checks at the point of use.

Another aspect of artifact classes is that they help grouping components together in an elegant way. Let's look at another example:

```
artifact OrderProcessing : BusinessComponent
{
    local artifact Customer : BusinessComponent
    {
        apply "layering" // see above
    }
    artifact Order : BusinessComponent
    {
        apply "layering"
        connect to Customer using BC2BC // defined above
    }
    connect to ProductManagement using BC2BC
}

artifact ProductManagement : BusinessComponent
{
    artifact Product : BusinessComponent
    {
        apply "layering"
        connect to Part using BC2BC
    }
    hidden artifact Part : BusinessComponent
    {
        apply "layering"
    }
}
```

The first thing you should notice is that neither "OrderProcessing" nor "ProductManagement" define the interfaces and connectors required by "BusinessComponent". They don't have to, because their nested artifacts do provide those connectors and interfaces. If an artifact belongs to a class and does not explicitly define a required interface or connector Sonargraph will check if it has nested artifacts that do.

In the case of interfaces Sonargraph will implicitly create a missing interface by exporting the matching interfaces of nested artifacts that are not hidden. In the case of connectors Sonargraph will implicitly create a missing connector by including the matching connectors of nested artifacts that are not local.

Here is the same example with all those implicitly defined interfaces and connectors explicitly defined:

```
artifact OrderProcessing : BusinessComponent
{
    local artifact Customer : BusinessComponent
    {
        apply "layering" // see above
    }
    artifact Order : BusinessComponent
    {
        apply "layering"
        connect to Customer using BC2BC // defined above
    }
    // Implicitly defined
    connector Controller
    {
        include any.Controller  // will not include Customer.Controller because Customer is local
    }
    connector Model
    {
        include any.Model       // will not include Customer.Model because Customer is local
    }
    interface IService
    {
        export any.IService
    }
    interface Model
    {
        export any.Model
    }
    // end of implicit definitions
    connect to ProductManagement using BC2BC
}

artifact ProductManagement : BusinessComponent
{
    artifact Product
    {
        apply "layering"
        connect to Part using BC2BC
    }
    hidden artifact Part
    {
        apply "layering"
    }
    // Implicitly defined
    connector Controller
    {
        include any.Controller
    }
    connector Model
    {
        include any.Model
    }
    interface IService
    {
        export any.IService // will not include Part.IService because Part is hidden
    }
    interface Model
    {
        export any.Model    // will not include Part.Model because Part is hidden
    }
    // end of implicit definitions
}
```

The implicit definitions only occur when you do not make an explicit definition. So you can always override those definitions although this should hardly ever be necessary.

Using artifact classes can become a very powerful pattern especially for the design of larger systems with many components that have a similar internal structure.

You can also use classes to define connections:

```
artifact C1 : BusinessComponent
{
    apply "layering"
    connect to class BusinessComponent using BC2BC
}

artifact C2 : BusinessComponent
{
    apply "layering"
    connect to class BusinessComponent using BC2BC
}

artifact C3 : BusinessComponent
{
    apply "layering"
    connect to class BusinessComponent using BC2BC
}
```

This allows each of the 3 components to talk to the two other using the given connection scheme. Since this approach will lead to cyclic dependencies between artifacts the cycle check for architectures using this feature is disabled.

11.11. How to Organize your Code

In this article I am going to present a realistic example that will show you how to organize your code and how to describe this organization using our architecture DSL. Let us assume we are building a micro-service that manages customers, products and orders. A high level architecture diagram would look like this:

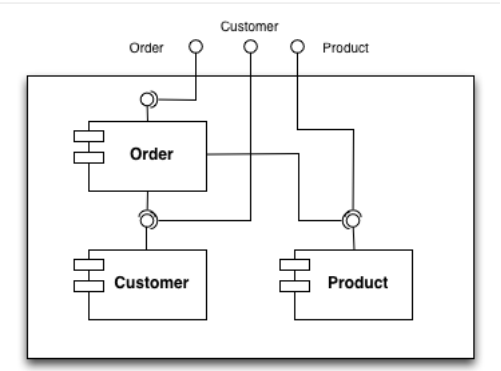


Figure 11.2. Architecture of the order management micro service

It is always a good idea to cut your system along functionality, and here we can easily see three subsystems. In Java you would map those subsystems to packages, in other languages you might organize your subsystem into separate folders on your file system and use namespaces if they are available.

Let us assume the system is written in Java and its name is "Order Management". In that case we would organize the code into those 3 packages:

```
com.hello2morrow.ordermanagement.order
com.hello2morrow.ordermanagement.customer
com.hello2morrow.ordermanagement.product
```

This can easily be mapped to our DSL:

```
artifact Order
{
    include "**/order/**"
    connect to Customer, Product
}

artifact Customer
{
    include "**/customer/**"
}

artifact Product
{
    include "**/product/**"
}
```

Internally all three subsystem have a couple of layers and the layering is usually the same for all subsystems. In our example we have four layers:

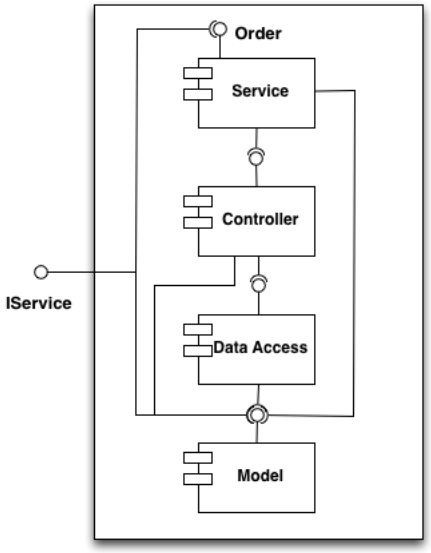


Figure 11.3. Layering of a subsystem

A service would only expose its service ad its model layer to the outside. The service layer contains all the service interfaces and talks to the controller and the model layer. The controller layer contains all the business logic and uses the data access layer to retrieve or persist data using JDBC. The model layer is defining the entities we are working with.

We will use a separate architecture file named "layering.arc" to describe our layering:

```
// layering.arc
artifact Service
{
    include "**/service/**"
    connect to Controller
}

artifact Controller
{
    include "**/controller/**"
    connect to DataAccess
}

require "JDBC"

artifact DataAccess
{
    include "**/data/**"
    connect to JDBC
}

public artifact Model
{
    include "**/model/**"
}

interface IService
{
    export Service, Model
}
```

Please note, that we declared "Model" as a public artifact. That saves us the need to explicitly connect all the other layers to "Model". Also note the "require" statement. Here refer to a third architecture file, that contains the definition of the artifact JDBC. This way we can ensure that only the data access layer can make JDBC calls. using "require" will only declare the artifacts

contained in the required file, but not define them. This means that the artifacts in "JDBC" have to be defined on another level. The interface is used to define the exposed parts of a subsystem. When connecting to the "IService" interface you have only access to the "Service" and the "Model" layer.

NOTE

Architecture files using "require" are not self-contained and cannot be added to the architecture check!

Now we use apply statements to apply the layering to our three subsystems:

```
artifact Order
{
    include "**/order/**"
    apply "layering"
    // Connect to the IService interface of Customer and Product
    connect to Customer.IService, Product.IService
}

artifact Customer
{
    include "**/customer/**"
    apply "layering"
}

artifact Product
{
    include "**/product/**"
    apply "layering"
}

// By using apply we define the artifacts of "JDBC" in this scope
apply "JDBC"
```

We also apply "JDBC" in the outermost scope to ensure that the artifacts in there are defined exactly once.

For the sake of completeness, here is the definition of "JDBC.arc":

```
// JDBC.arc
artifact JDBC
{
    include "**/javax/sql/**"
}
```

By using smart package naming it becomes easy to map your code to the architecture description. For example the order subsystem would have four packages:

```
com.hello2morrow.ordermanagement.order.service
com.hello2morrow.ordermanagement.order.controller
com.hello2morrow.ordermanagement.order.data
com.hello2morrow.ordermanagement.order.model
```

As you can see it required relatively little effort to create a formal and enforceable architecture description for our example.

11.12. Designing Generic Architectures Using Templates

Many companies already have some established design patterns which are supposed to be used in most of their applications. For example it makes sense to standardize the layering of business components. It also makes sense to establish specific rules how one business component can access another one. The template feature in our architecture DSL makes it very easy to add generic architecture blueprints to a quality model which would allow automatic verification of those architecture design patterns on any business component without having to create a component specific architecture.

For generic architectures to work properly it is a good idea to think about code organization, in particular the efficient use of name spaces or packages to reflect architectural intent. That can be easily done by using naming conventions:

```
com.hello2morrow.{component-name}.{layer-name}
```

In this simple example we assume that the component name is always the third part of a package/namespace name. The fourth part represents the layer. Knowing that we can now create a generic architecture description for this example:

```
// aspect file layering.arc
strict exposed artifact Service
{
    include "**/service/**"
}

strict exposed artifact Controller
{
    include "**/controller/**"
}

require "JDBC"

exposed artifact DataAccess
{
    include "**/data/**"
    connect to JDBC
}

public exposed artifact Model
{
    include "**/model/**"
}

public exposed optional artifact Util
{
    include "**/util/**"
}

deprecated hidden artifact Leftovers
{
    include "**"
}

// main file components.arc
template Components
{
    include "**/com/hello2morrow/(*)/**"
    exclude "**/com/hello2morrow/framework/**"

    artifact capitalize($1)+"Component"
    {
        apply "layering"
    }
}

public artifact Framework
{
    include "**/com/hello2morrow/framework/**"
}
```

In the aspect file "layering.arc" we define our standardized layering. At this point the layer artifacts do not really need to be exposed. That will be needed later when we add connection schemes to our example.

In the main file we use the new template feature. An template is a special kind of artifact that can dynamically create children artifacts out of elements that are matched by the pattern. The pattern must include at least one pair of parentheses so that we can extract the component name and use it as part of the name of a generated artifact. Inside of a template there always is a prototype artifact that uses a string typed expression as its name. '$1" represents the first extracted name part from the matched architecture filter name. We append "Component" to the capitalized extracted name part to form the name of a generated artifact. We explicitly exclude classes of a framework that is mapped to an extra artifact that has been declared to be public so that everything defined in "Components" can use it.

For our example we assume there are 3 components distributed over the following 3 packages:

```
com.hello2morrow.order
com.hello2morrow.customer
com.hello2morrow.product
```

Then the template artifact "Components" would generate 3 children artifacts named "OrderComponent", "CustomerComponent" and "ProductComponent". All of those would have access to "Framework" because it is a public artifact defined beneath "Components". But on the other hand the three generated components would not be allowed to access each other. Using templates there are currently three ways to regulate dependencies between generated artifacts:

- No dependency allowed (like in the above example)
- By marking the prototype artifact as "unrestricted" the generated artifacts could use each other (from default connector to default interface). It is always possible to restrict the default interface and/or the default connector by defining them explicitly.
- By using connection schemes in combination with artifact classes. That approach will be explained further down.

11.12.1. Using unrestricted generated artifacts

In the next example we use "unrestricted" in combination with a redefined default interface:

```
// main file components.arc
template Components
{
    include "**/com/hello2morrow/(*)/**"
    exclude "**/com/hello2morrow/framework/**"

    unrestricted artifact capitalize($1)+"Component"
    {
        apply "layering"

        interface default
        {
            export Service, Model, Util
        }
    }
}

public artifact Framework
{
    include "**/com/hello2morrow/framework/**"
}
```

Now the 3 generated artifacts can call each other, but only the "Service", "Model" and "Util" layers are exposed. If one of those generated artifacts were to access the "DataAccess" layer of another one this would be marked as an architecture violation.

11.12.2. Using connection schemes to regulate accessibility

If you need more control about the way generated artifacts can interact with other generated artifacts we need to use connection schemes in combination with artifact classes.

```
// main file components.arc
class Layered
{
    interface Service, Controller, DataAccess, Model, Util
    connector Service, Controller, DataAccess, Model, Util
}

connection-scheme C2C : Layered to Layered
{
    connect Service to target.Service, target.Controller, target.Model, target.Util
    connect Controller to target.Controller, target.Model, target.Util
    connect DataAccess to target.DataAccess, target.Model, target.Util
    connect Model to target.Model, target.Util
    connect Util to target.Util
}

template Components : Layered
{
    include "**/com/hello2morrow/(*)/**"
    exclude "**/com/hello2morrow/framework/**"

    artifact capitalize($1)+"Component"
    {
        apply "layering"
    }
    connect all using C2C
}
// ...
```

Now you have total control about the way components access each other. The connection scheme determines for each of the layers which layers can be used in the target artifact. The "connect all" statement declares the connection scheme to be used. The scheme has to connect an artifact class to itself ("Layered" to "Layered" in this example) and prototype artifact must be of the matching class. In our example that happens implicitly by using the class on the template.

11.13. Best Practices

This section explains how the views of Sonargraph can be used efficiently while working with the architecture definition and investigating reported architecture violations.

Investigate Violations

Architecture violations listed in the Issues view can have different root causes: a) There is a violating dependency, b) the architecture definition needs to be updated.

For further investigation, you can do the following:

- **Check the source code:** Simply double-click on the violation.
- **Investigate the dependency with more context:** Right-click on the violation and select "Show in Exploration View" → "Out".
- **Check the architecture definition:** Right-click on the violation and select "Show in Architecture View". The Architecture view is opened with the artifact selected that contains the element causing the violation. Double-click on the artifact and the Architecture File view is opened, highlighting the line of the artifact's definition.
- **Check the artifact dependencies:** The description of the architecture violation contains more details, e.g. "[Static Method Call] 'X' cannot access 'TypeA.java' from 'Y' ...". If you are interested in the context of the involved artifacts 'X' and 'Y', open the Architecture view via right-click "Show in Architecture View". Open the context menu for the selected artifact and open the architecture-based Exploration view via "Show in Exploration View" → "In And Out".

  ### NOTE

  The architecture-based Exploration view offers dependency information down to the *component* level, since that is the smallest unit of assignment to an artifact.

Related topics:

- Section 8.11.2.1, “Focus”
- Section 8.11.1, “Exploration View”

11.14. Architecture DSL Language Specification

For the sake of completeness please find a formal grammar of our architecture DSL in EBNF form. The semantics of the language have been described in the preceding sections.

```
Body            = Declaration* Connection*
                ;

Declaration     = ArtifactDecl
                | ExtendDecl
                | ApplyDecl
                | RequireDecl
                | IncludeDecl
                | ExcludeDecl
                | InterfaceExt
                | ConnectorExt
                | InterfaceDecl
                | ConnectorDecl
                | Scheme
                | ClassDecl
                | TemplateDecl
                ;

ClassDecl       = "class" IDENT "{" ClassMember* "}"
                ;
ClassMember     = "interface" List
                | "connector" List
                ;

List            = IDENT ("," IDENT)*
                ;
ArtifactDecl    = StereoTypes "artifact" IDENT (":" IDENT)? "{" Priority? Body "}"
                ;

ExtendDecl      = "extend" IDENT "{" ExtendBody "}"
                | StereoType+ "extend" IDENT "{" ExtendBody "}"
                ;

Priority        = "priority" NUMBER
                ;

ExtendBody      = Declaration* DisconnectDecl* Connection*
                ;

DisconnectDecl  = "disconnect" IDENT? "from" IdentList
                ;

StereoTypes     = StereoType*
                ;

StereoType      = "public"
                | "hidden"
                | "local"
                | "exposed"
                | "unrestricted"
                | "relaxed"
                | "strict"
                | "optional"
                | "deprecated"
                ;
```

```
ApplyDecl       = "apply" STRING
                ;

RequireDecl     = "require" STRING
                ;

IncludeDecl     = "strong"? "include" STRING
                | "include" "all"
                | "include" "dependency-types" DependencyTypes
                ;

ExcludeDecl     = "exclude" STRING
                | "exclude" "dependency-types" DependencyTypes
                ;

DependencyTypes = DependencyType ("," DependencyType)*
                ;

DependencyType  = IDENT
                ;

InterfaceDecl   = ("override"|"optional")? "interface" IDENT "{" InterfaceBody "}"
                ;

InterfaceExt    = "extend" "interface" IDENT "{" InterfaceBody "}"
                ;

InterfaceBody   = IDeclaration*
                ;

IDeclaration    = IncludeDecl
                | ExcludeDecl
                | Export
                ;

ConnectorDecl   = "override"? "connector" IDENT "{" ConnectorBody "}"
                ;

ConnectorExt    = "extend" "connector" IDENT "{" ConnectorBody "}"
                ;

ConnectorBody   = CDeclaration*
                ;

CDeclaration    = IncludeDecl
                | ExcludeDecl
                | Include
                ;

Export          = "export" SpecIdentList
                ;

Include         = "include" SpecIdentList
                ;

SpecIdentList   = SpecIdent ("," SpecIdent)*
                ;

SpecIdent       = "any" ("." IDENT)*
                | IDENT ("." IDENT)*
                ;
```

```
IdentList       = Identifier ("," Identifier)*
                ;

Identifier      = IDENT ("." IDENT)*
                ;

Connection      = "connect" "to" IdentList "transitively"?
                | "connect" Identifier "to" IdentList "transitively"?
                | "connect" "to" IdentList "using" IDENT
                ;

Scheme          = "connection-scheme" IDENT (":" IDENT "to" IDENT)? "{" TargetUse* "}"
                ;

TargetUse       = "connect" Identifier "to" TargetIdentList
                | "connect" "any" "." Identifier "to" TargetIdentList
                ;

TargetIdentList = TargetIdent ("," TargetIdent)*
                ;

TargetIdent     = "target" ("." IDENT)+
                | "target" "." "any" ("." IDENT)+
                ;

IDENT           = ("A" .. "Z" | "a" .. "z")("A" .. "Z" | "a" .. "z" | "0" .. "9" | "_" | "-")*
                ;

TemplateDecl    = StereoTypes "template" IDENT (":" IDENT)? "{" TemplateBody "}"
                ;

TemplateBody    = TemplateInclude+ TemplateExclude* TemplArtifact TemplateConnect*
                ;

TemplateInclude = "include" STRING
                ;

TemplateExclude = "exclude" STRING
                ;

TemplateConnect = "connect" "to" IdentList "transitively"?
                | "connect" Identifier "to" IdentList "transitively"?
                | "connect" "to" IdentList "using" IDENT
                | "connect" "all" "using" IDENT
                ;

TemplArtifact   = StereoTypes "artifact" NameExpr (":" IDENT)? "{" Body "}"
                ;

NameExpr        = NameItem
                | NameItem "+" NameExpr
                ;

NameItem        = STRING
                | "$" ["1"-"9"]
                | IDENT "(" NameExpr ")"
                ;

                // '@' stands for any character except the context specific terminator
STRING          = '"' (@ | '\\' @)* '"' | "'" (@ | '\\' @)* "'"
                ;
```

Chapter 12. Visualizing Architecture Aspects

Sonargraph's domain specific language (DSL) to describe architecture aspects is very powerful. An architecture aspect consists at least of 1 top-level architecture file that has been added to the architecture configuration and is checked automatically. Such a top-level architecture file can include other architecture files reusing common definitions. For any checked architecture file (i.e. an aspect) it is possible to generate a UML component diagram.

A UML component diagram complements in several ways our text based architecture aspects:

1. It is a commonly accepted form of communicating architecture definitions.

2. It shows the resulting architecture aspect in 1 diagram even if it is spread over several files.

3. It can be used to cross-check the underlying text based architecture aspect (i.e. are the resulting restrictions the intended ones?).

Let's have a look at a concrete example. Suppose we want an architecture aspect containing 2 (vertical) slices (domain driven divisions) *Customer* and *Common* and 3 layers (technical divisions) *View*, *Model* and *Persistence*. Furthermore we want a separate *License* component usable only from the Common slice and a *JDBC* component usable only from the Persistence layers.

One way to express this with our architecture DSL would be the following:

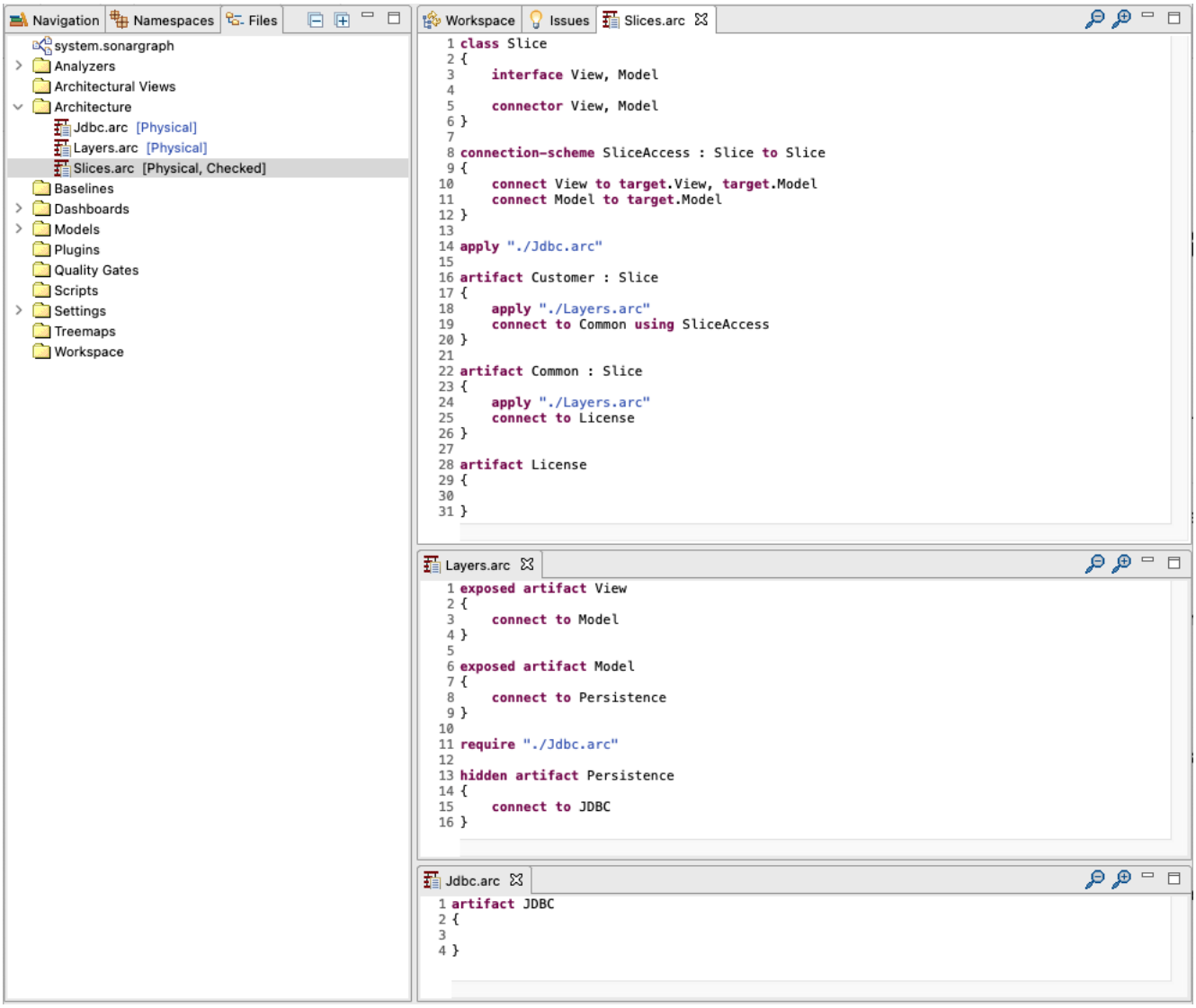


Figure 12.1. DSL Example

Our top-level architecture file would be *Slices.arc*, using *Layers.arc* and *Jdbc.arc*. Make sure to add the top-level architecture file to the architecture check via the context menu entry *Add To Architecture Check* on the corresponding file. As you can see above the Slices.arc is checked. Note that in the example above we have omitted all include patterns that would be needed to match the code for the sake of simplicity.

Once we have a checked architecture file we can simply generate an UML component diagram via the context menu entry *Show in Architecture Diagram View* on the corresponding file.

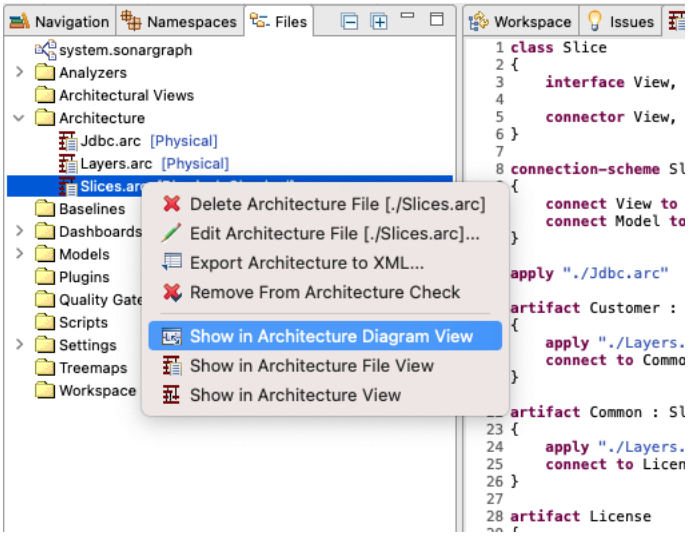


Figure 12.2. Generate UML Component Diagram

In our example that results in the following UML component diagram:

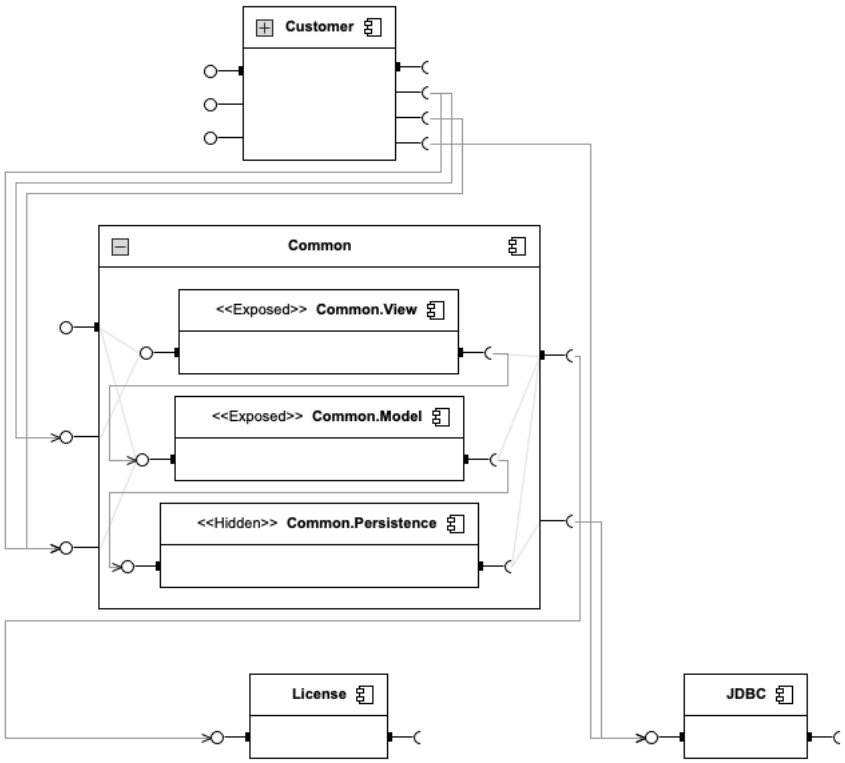


Figure 12.3. UML Component Diagram

The generated UML component diagram is *interactive* in the sense that you can select different elements (e.g. components, connectors, interfaces). The selected element is highlighted in yellow along with it's connected elements (e.g. a connector that is included in a higher level connector), reachable (i.e. allowed) elements are highlighted in green. Components also might have child components, such components can be expanded and collapsed (e.g. *Customer* and *Common*).

Connectors and interfaces that show a small solid black rectangle on their anchor point are directly defined by their component. The ones without that decorator are coming from child components.

The different elements also offer a tooltip showing interesting information about incoming/outgoing connections and other aspects. Clicking into the tooltip window will leave it open (until pressing ESC). The content of the tooltip may be selected and copied.

The currently visible UML diagram may be exported as an image using the context menu.

The components are layout in a levelized grid. Components with no incoming connections are on top, components with no outgoing connections are on the bottom. The further down a component is, the more other components depend on it. Components on the same level have more incoming dependencies from left to right. Components with the same number of incoming dependencies have more outgoing dependencies from left to right.

As long as the view stays open it is also updated when saving changes to the underlying architecture file(s).

Lets cross-check the usage of our *JDBC* component by selecting it:

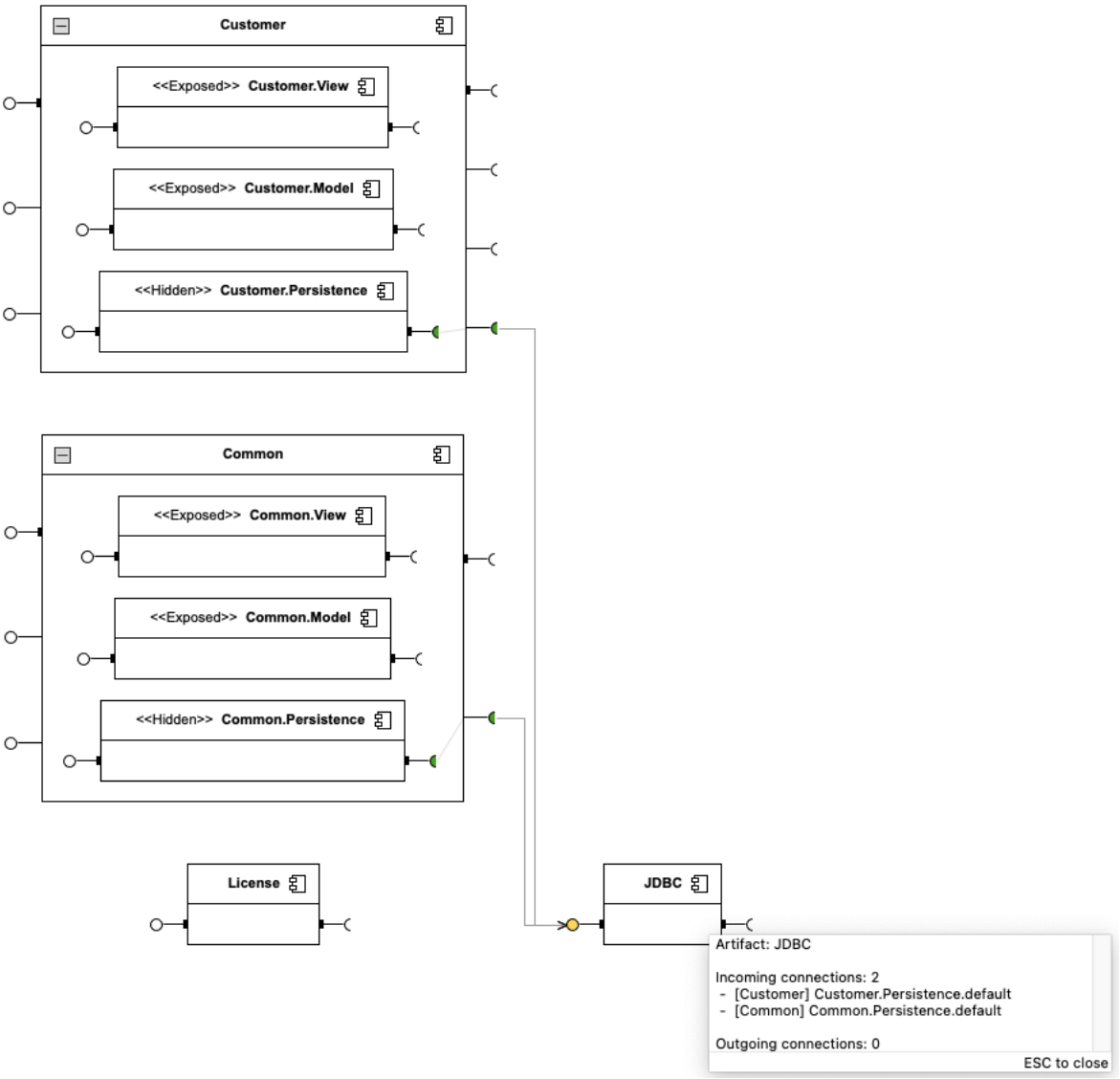


Figure 12.4. Cross-check the JDBC Component

As we can see *JDBC* may only be used from the Persistence layers. More details are provided by the tooltip of the component.

Let’s check our *License* component:

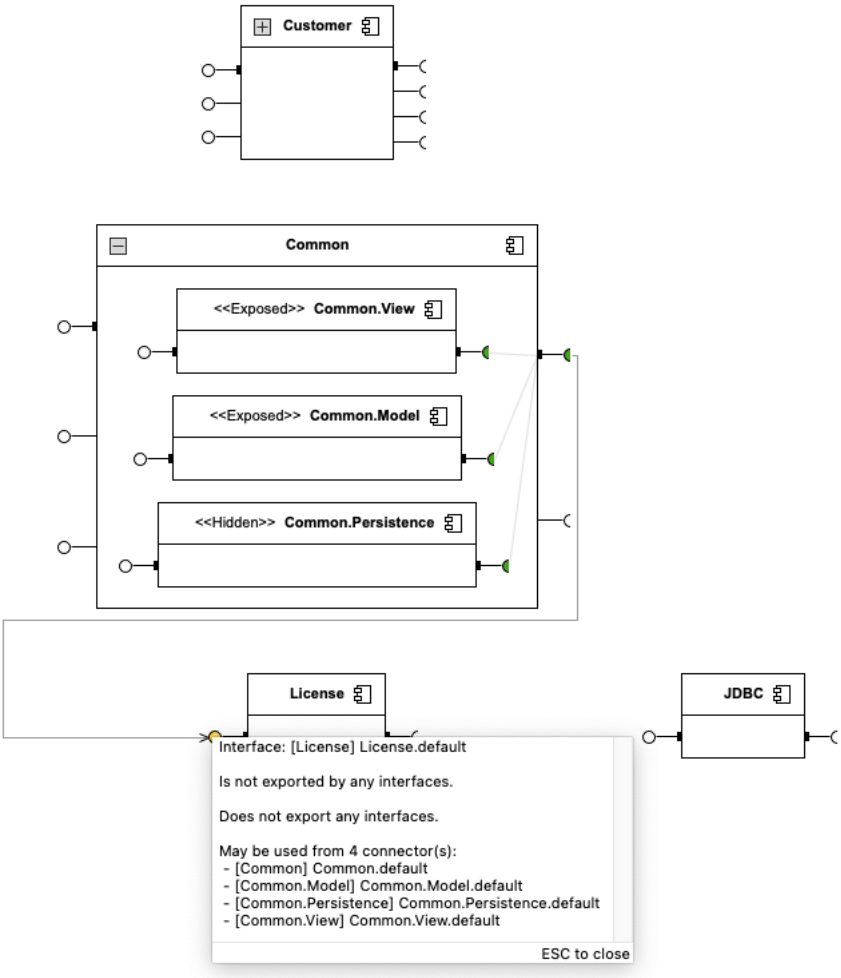


Figure 12.5. Cross-check the License Component

Great, our *License* component may only be used from the *Common* slice. Again, more information is provided by the tooltip of the interface.

Chapter 13. Interactive Restructuring and Code Organization

Sonargraph offers the concept of the architectural view to enable:

- Interactive restructuring a *software system* via refactorings.
- Interactive organization of a *software system* via architecture artifacts. Components (physical) or top-level programming elements (logical) are assigned to those artifacts based on the package, namespace or directory structure or based on an architectural pattern language overcoming structural constraints.
- Generation of architecture DSL files that can be automatically checked.
- Generation of refactoring lists that are applied to the "production" modifiable (virtual) model.

An Architectural view is an Exploration view with architecture modeling capabilities (see Section 8.11.1, “Exploration View”). An Architectural view is created using a specific structure mode and by applying operations to it. It is created based on the currently loaded (parsed) system after applying the selected virtual model. Multiple architectural views can be defined that are each persisted in an individual file.

- Creation of an architectural view which is persisted in a file. Use menu "File" → "New" → "Architectural View" → "New Architectural View..." or right-click in the Files view on the Architectural Views node and use the corresponding context menu entry. Choose between 4 model structures (physical with or physical without root directories, logical system or logical module scope) when creating an architectural view. An existing architectural view can be opened from the files view via the context menu or via double-click. Both "create" and "open" interactions require a loaded parser model.
- The architectural view shows a tree-like structure down to component level (physical) or top-level programming element (logical).
- Selection support by pressing SHIFT in combination with the arrow keys (up and down) or left mouse click for bulk selection. Use the modifier key (CMD, CTRL) of your operating system in combination with left mouse-click to add or remove elements to/from the current selection.
- Artifacts with their properties can be created either based on a selection or empty. Try the context menu on selections (right mouse click) to see what is possible. **F2** allows editing 1 or multiple artifacts.
- Assign/remove elements to/from existing artifacts via drag and drop.
- Change parent/child structures of artifacts via drag and drop.
- Assign components (physical) or top-level programming elements (logical) to artifacts with manual filter definition using patterns. All assignment strategies from the architecture DSL are supported, see Section 11.1, “Models, Components and Artifacts” and following chapters.
- Define explicitly allowed artifact connections.
- Hide non-artifact/non-module nodes in their corresponding artifacts to make them unaccessible.
- Apply delete refactorings to non-artifact/non-module nodes.
- Apply delete refactorings to dependencies.
- Apply move refactorings to components (physical) or top-level programming elements (logical).
- Apply rename refactorings to non-artifact/non-module nodes.
- Use focus operations to visualize only certain elements of the Architectural view.
- Create packages, namespaces or directories when needed as targets for move refactorings.

- Create 'Findings' based on dependencies. Those findings appear in a list, have a name an optional description. Optionally dependencies violating the architecture contained in a finding can be ignored. It is also possible to apply focus operations based on findings.
- The operations you apply (e.g. create/edit artifacts, move elements, delete elements ...) may be seen in the operations view. The operations may also be deleted from that list using the context menu. Operations that have no effect (e.g. a previous element is no longer their because the code changed) are marked with a warning marker.
- The architecture is checked in real-time meaning that you see the dependencies changing their colors according to their violation state when applying operations.
- Undo/redo of operations.
- Forward/backward navigation.
- Export what you see into images files on disk.
- Generation of an architecture DSL file based on the model defined in the architectural view.
- Export of resulting refactorings into an Excel file.
- Export of resulting violations into an Excel file.
- The views Properties, Parser Dependencies (Out) and Parser Dependencies (In) react to the selection in the Architectural view and show the corresponding additional information of the selected element.

Levelization Mode and Operations

Since the Architectural view is designed to create architecture aspects it always shows the levelization mode widget (see Section 8.11.1.1, “Levelization”). Architectural view operations can only be applied when the levelization mode is set to 'Non Artifacts Only' since the artifacts are not shown in their definition order.

13.1. Assigning Elements to Artifacts

The Architectural view offers 2 strategies to assign elements to artifacts:

- Explicit assignment per drag and drop based on either package/namespace/directory or components (physical) or top-Level programming elements (logical). Alternatively the 'Move' dialog can be used.
- Manual filter management based on components (physical) or top-Level programming elements (logical) based on patterns supporting different underlying identifiers or annotation, super class or super interface dependencies.

NOTE: An artifact has either explicitly assigned elements or has a manual filter but not both. Artifacts with explicitly assigned elements and manual filters can be mixed in one Architectural view.

When creating/editing an artifact the manual filter can be managed on the second page of the artifact wizard.

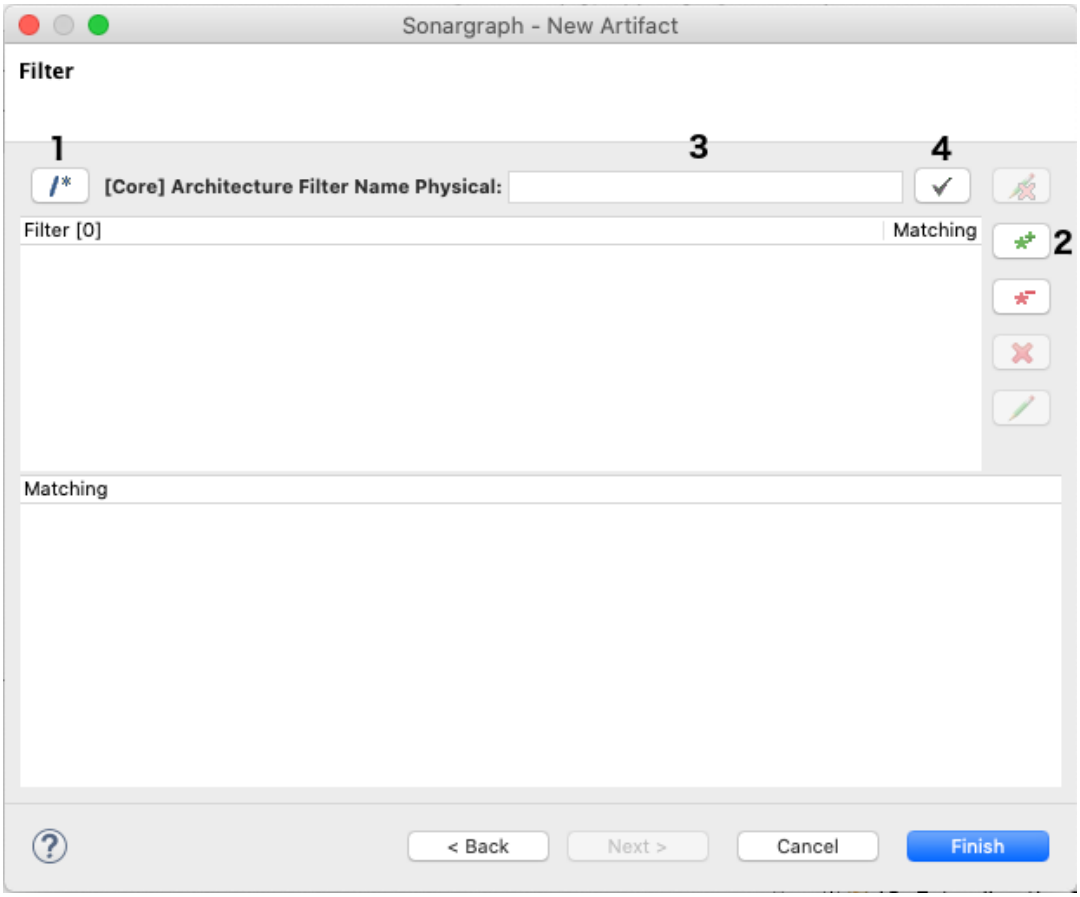


Figure 13.1. Artifact Filter

The basic flow to add a new pattern is as follows:

- Choose the assignment strategy by pressing the button marked with '1'. There you can choose between several options summarized below.
- Add a new pattern by pressing the button marked with '2'.
- Edit the pattern in the field marked with '3'. The currently matching elements are show in the area labeled with 'Matching'.
- Once the pattern is valid the icon on the button marked with '4' turns green. If you are satisfied press the button to add it to the already existing pattern shown in the area labeled with 'Filter'.

HINT

Moving the mouse pointer on top of the assignment strategy label ('[Core] Architecture Name Physical:' in the screenshot) will show a tool tip explaining the pattern usage with example. The tool tip is focusable. Clicking into it will convert it into a small dialog staying o top supporting selection/copy of the shown text.

Available assignment strategies:

Strategy	Structure Mode Type	Languages
Architecture Filter Name Physical	Physical	All
Architecture Filter Name Logical	Logical	All
Physical Filter Name	Logical	All
Workspace Filter Name	Physical, Logical	All
Extends Class	Physical, Logical	All
Implements Interface	Physical, Logical	Java, C#
Has Annotation	Physical, Logical	Java
Has Annotation Value	Physical, Logical	Java
Header Path	Physical	C,C++

Table 13.1. Available Assignment Strategies

See Section 11.1, “Models, Components and Artifacts”, Section 11.1.1, “Using other criteria to assign components to artifacts” and Section 11.1.2, “List of predefined attribute retrievers” in the architecture DSL documentation for examples.

Chapter 14. Examining Changes

Sonargraph provides an overwhelming amount of information for large systems. Most of the times comparing the information against a baseline and focussing on the changes is enough - like a newspaper versus a whole encyclopedia. This feature is available in Sonargraph via the "System Diff" view that can be opened via "Window" → "Show View" → "System Diff" . The baseline is represented by an XML report and can be applied by clicking the links at the top of the view opens the dialogs:

1. "New Baseline": Allows creating and directly applying a new baseline, e.g. at the beginning of a feature implementation or before changing the software system. Adding some context info makes it easier to identify the report later on.

2. "Open Baseline": Allows selecting an existing XML report from disk, e.g. to compare the current system against a report generated by Sonargraph-Build at the end of the last sprint. Previously selected baselines are displayed in the table with the most recently used at the top.

3. "Download Baseline": Allows downloading an existing XML report from your Sonargraph-Enterprise server.

4. "Export Diff Report": Generates a HTML report of the current diff information.

5. "Detach Baseline": Disables the System Diff analyzer. This is useful to speed up processing when you modify the configuration for a large system.

TIP

The same can be achieved by changing the analyzer execution level to anything below "Full" via the menu "System" → "Analyzer Execution Level".

Two types of baselines can be created: "System" and "Local" baselines. A system baseline is meant to be useful for all users of the Sonargraph system and is used by Sonargraph-Build. Thus, it is stored in the Sonargraph system's directory "Baselines". System baselines are usually created at the beginning or end of a release. A local baseline is only useful for the current user and usually has a shorter life-span. Examples are baselines created before a feature is implemented.

TIP

If a baseline is generated at the beginning of a feature implementation, the System Diff view provides a quick overview about changes related to Sonargraph issues and how the overall state of the system has developed.

NOTE

This functionality is only available in the commercial version of Sonargraph.

Differences are detected related to the system configuration (path, name, features, analyzers, scripts, plugins), system-level metrics, workspace (filter, modules, root directories), issues, resolutions (ignores, tasks, refactorings), cycle groups, duplicate code blocks and *Architecture Models*, i.e. checked architectures. More information about a detected change is provided in the "Details" column. The comparison for architectures is done on the 'model' level, i.e. changes in comments, formatting changes and changes that do not alter the semantic of the model are not reported. The following screenshot shows changes in the issues view with a focus on cycle group issues:

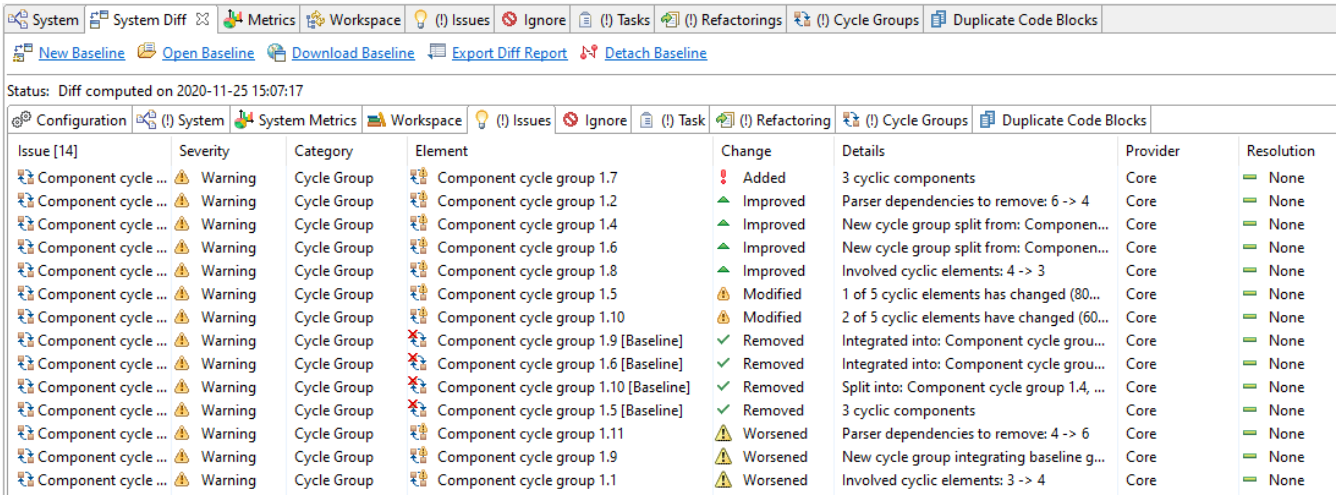


Figure 14.1. System Diff View (Issues)

NOTE

Since the cycle group names are not stable, the matching is done based on the contained elements. If up to 40% of the cyclic elements have changed, the cycle groups are matched, otherwise they are treated as different groups and are reported as removed from the baseline and added in the current system.

A similar approach is taken for duplicate code blocks, where the individual occurrences are matched against each other, tolerating extension and shortening of blocks.

The following types of changes in cycle groups are detected:

1. **Added:** Cycle group was not present in baseline.

2. **Removed:** Cycle group existed in baseline but is no longer present. If two or more baseline cycle groups are integrated into a single cycle group or a baseline cycle group is split into several smaller cycle groups, this is indicated in the "details" column.

3. **Improved:** Cycle group consists now of fewer elements and/or has fewer parser dependencies to remove.

   If a cycle group is the result of splitting a baseline cycle group, this is indicated in the "details" column. Cyclic elements of this group are reported as added.

4. **Worsened:** Cycle group consists now of more elements and/or has more parser dependencies to remove.

   If a cycle group is the result of integrating two or more baseline cycle groups, this is indicated in the "details" column.

5. **Modified:** If up to 40% of the cyclic elements have changed but the number of cyclic elements in the cycle group is the same it is identified as the same group and marked as modified.

If a worsened cycle group is opened in the Cycle view, the added elements are highlighted. See “Highlighting Added Cyclic Elements” for details.

The tab "Cycle Groups" provides additional details about the added/removed cyclic elements. Filter options on the top-right corner of the tab allow to hide unmodified cyclic elements or issues with resolution.

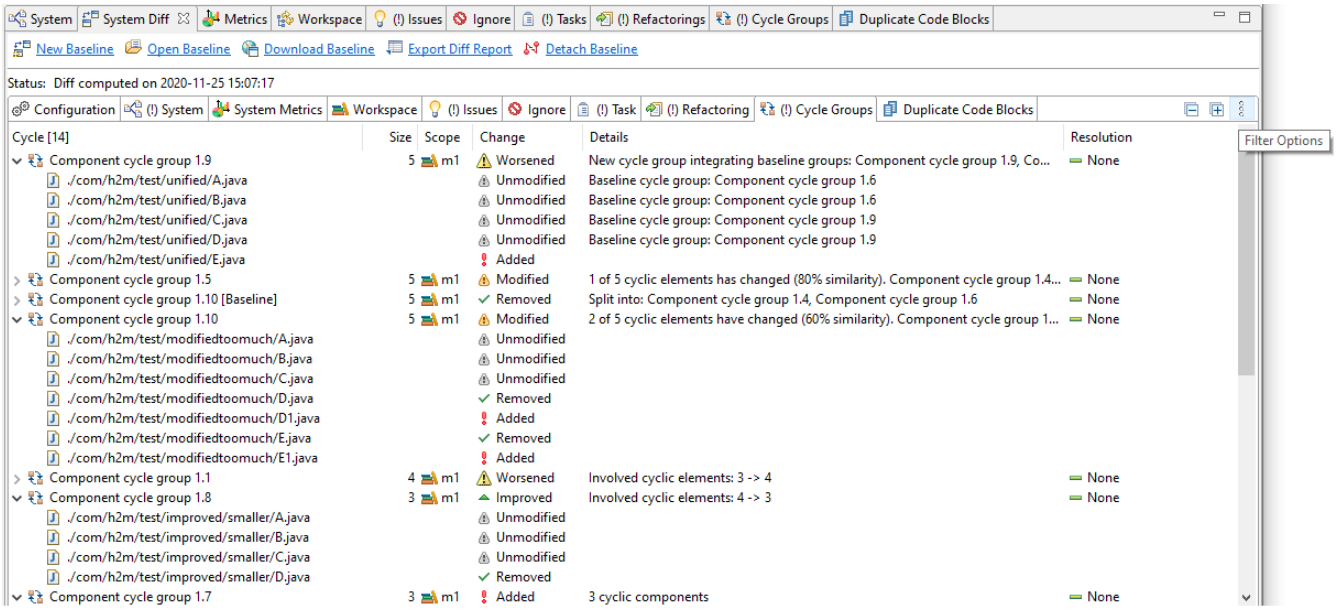


Figure 14.2. System Diff View (Cycle Groups)

Current Limitations

The following changes only indirectly affect the Sonargraph issues, but will be treated as changes by the diff detector. The issues in the baseline report will be reported as removed and the issues from the current system as added, despite the fact that the issues are logically the same:

1. If an element (e.g. type, method) or one of its parents (e.g. namespace, module, root) is renamed, its fully qualified name changes and thus its issues are reported as changed.

2. If a script or an architecture file is renamed, the origin of the issues generated by those resources is changed.

3. If artifacts in architectures are renamed, the resulting issues cannot be matched.

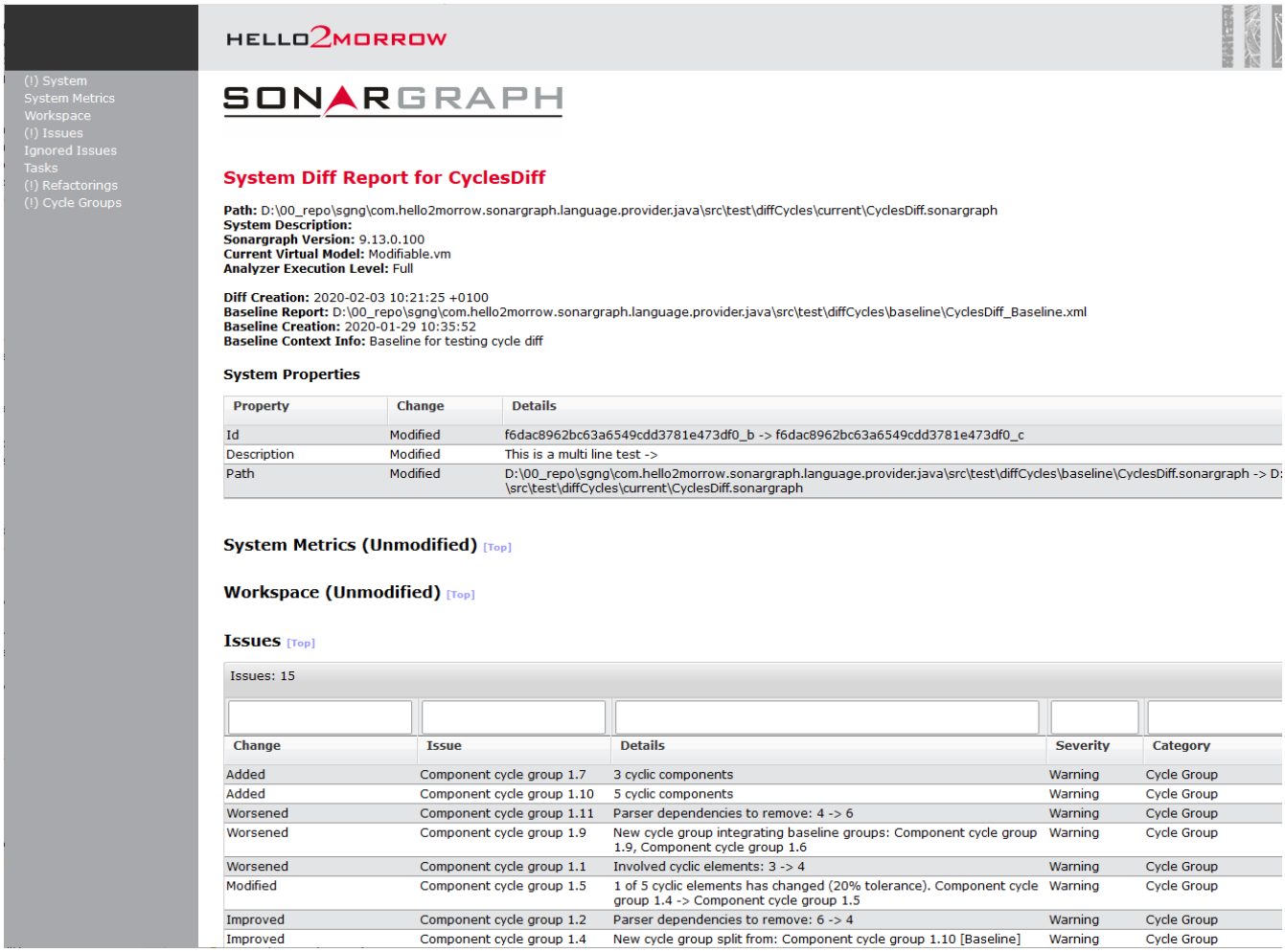
NOTE

As with every modification: Frequent and small changes are easier to review than big-bang refactorings.

This feature has been introduced with the Sonargraph release 9.13 and we will continue improving the precision of the results and integrating it into other views in upcoming releases. Feedback is always welcome and can be sent to `<support@hello2morrow.com>`.

Export Diff Report to HTML

The System Diff view allows exporting the current info to HTML as the following screenshot illustrates. Sections that contain changes are marked with an exclamation mark "(!)" in the navigation area on the top left.



HELLO2MORROW

(!) System
System Metrics
Workspace
(!) Issues
Ignored Issues
Tasks
(!) Refactorings
(!) Cycle Groups

SONARGRAPH

System Diff Report for CyclesDiff

Path: D:\00_repo\sgng\com.hello2morrow.sonargraph.language.provider.java\src\test\diffCycles\current\CyclesDiff.sonargraph
System Description:
Sonargraph Version: 9.13.0.100
Current Virtual Model: Modifiable.vm
Analyzer Execution Level: Full

Diff Creation: 2020-02-03 10:21:25 +0100
Baseline Report: D:\00_repo\sgng\com.hello2morrow.sonargraph.language.provider.java\src\test\diffCycles\baseline\CyclesDiff_Baseline.xml
Baseline Creation: 2020-01-29 10:35:52
Baseline Context Info: Baseline for testing cycle diff

System Properties

Property	Change	Details
Id	Modified	f6dac8962bc63a6549cdd3781e473df0_b -> f6dac8962bc63a6549cdd3781e473df0_c
Description	Modified	This is a multi line test ->
Path	Modified	D:\00_repo\sgng\com.hello2morrow.sonargraph.language.provider.java\src\test\diffCycles\baseline\CyclesDiff.sonargraph -> D:\src\test\diffCycles\current\CyclesDiff.sonargraph

System Metrics (Unmodified) [Top]

Workspace (Unmodified) [Top]

Issues [Top]

Issues: 15

Change	Issue	Details	Severity	Category
Added	Component cycle group 1.7	3 cyclic components	Warning	Cycle Group
Added	Component cycle group 1.10	5 cyclic components	Warning	Cycle Group
Worsened	Component cycle group 1.11	Parser dependencies to remove: 4 -> 6	Warning	Cycle Group
Worsened	Component cycle group 1.9	New cycle group integrating baseline groups: Component cycle group 1.9, Component cycle group 1.6	Warning	Cycle Group
Worsened	Component cycle group 1.1	Involved cyclic elements: 3 -> 4	Warning	Cycle Group
Modified	Component cycle group 1.5	1 of 5 cyclic elements has changed (20% tolerance). Component cycle group 1.4 -> Component cycle group 1.5	Warning	Cycle Group
Improved	Component cycle group 1.2	Parser dependencies to remove: 6 -> 4	Warning	Cycle Group
Improved	Component cycle group 1.4	New cycle group split from: Component cycle group 1.10 [Baseline]	Warning	Cycle Group

Figure 14.3. HTML Diff Report

Chapter 15. Defining Quality Gates

Starting with version 10.3 Sonargraph provides the option to define quality gates (commercial license required). A "quality gate" consists of a set of conditions. If one or more conditions are not met, this is flagged by a "Quality Gate Condition Failed" issue. Conditions define the expectations for the current system's state as well as for an expected quality trend w.r.t a defined baseline. Some sample conditions:

- "Less than 10 error issues without resolution."
- "No threshold violations for metric 'Core:SourceFile:TotalLines'."
- "No additional error issues."
- "Number of architecture violations must be reduced by at least 10%."
- "An increase in the Average Component Dependency (ACD) must be lower than 5%."

Sonargraph allows the definition of any number of quality gates and validates those that are "activated", similar to architectures and scripts. Activated quality gates are validated by the "Quality Gate" analyzer. A quality gate consists of two sections, the section for conditions based on the current system state and the section for conditions based on the system diff with respect to a baseline (see Chapter 14, *Examining Changes*). The Quality Gate view shows the conditions at the top and the table in the lower half lists the matched issues / elements for the selected condition.

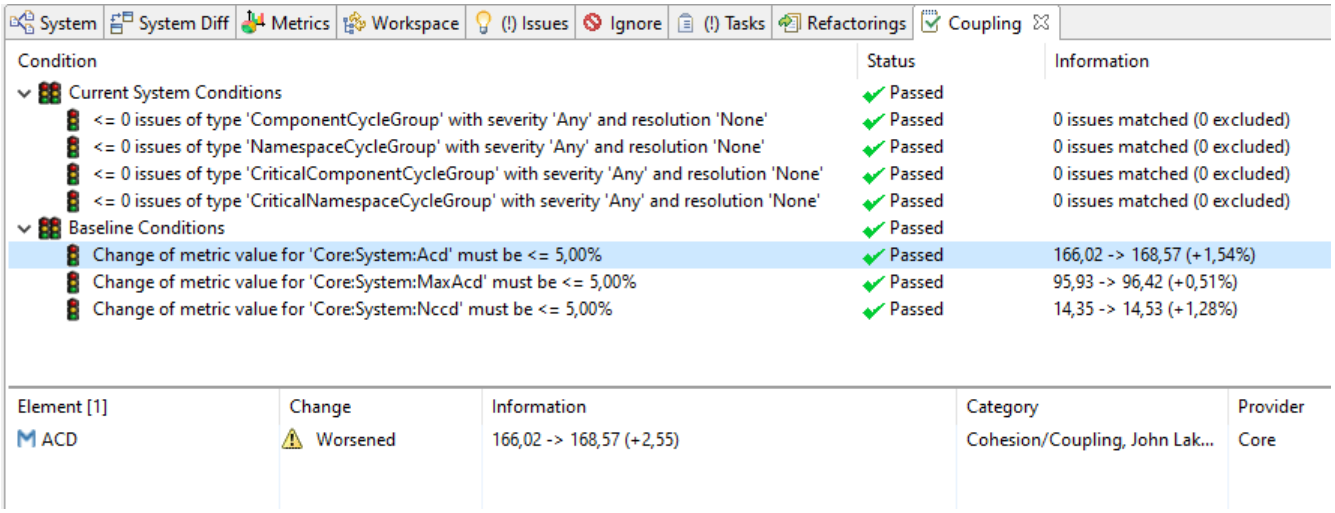


Figure 15.1. Quality Gate View

TIP

An unlimited number quality gates is supported, so there is no reason to put too many conditions into a single quality gate.

NOTE

Conditions for the current system's state can only be defined on issues. If you want to set a condition for a certain metric value, define a threshold for that metric and then check on the existence of threshold violations.

15.1. Creating Quality Gates

A new quality gate can be created via the main menu "File" → "New" → "New Quality Gate..." or by selecting the "Quality Gates" folder in the "Files" view and opening the context menu.

Define Conditions for the Current System's State

A condition for the current system's state can be created via the context menu of the "Current System Conditions" node. If a metric id is specified, only threshold violations for that metric are matched.

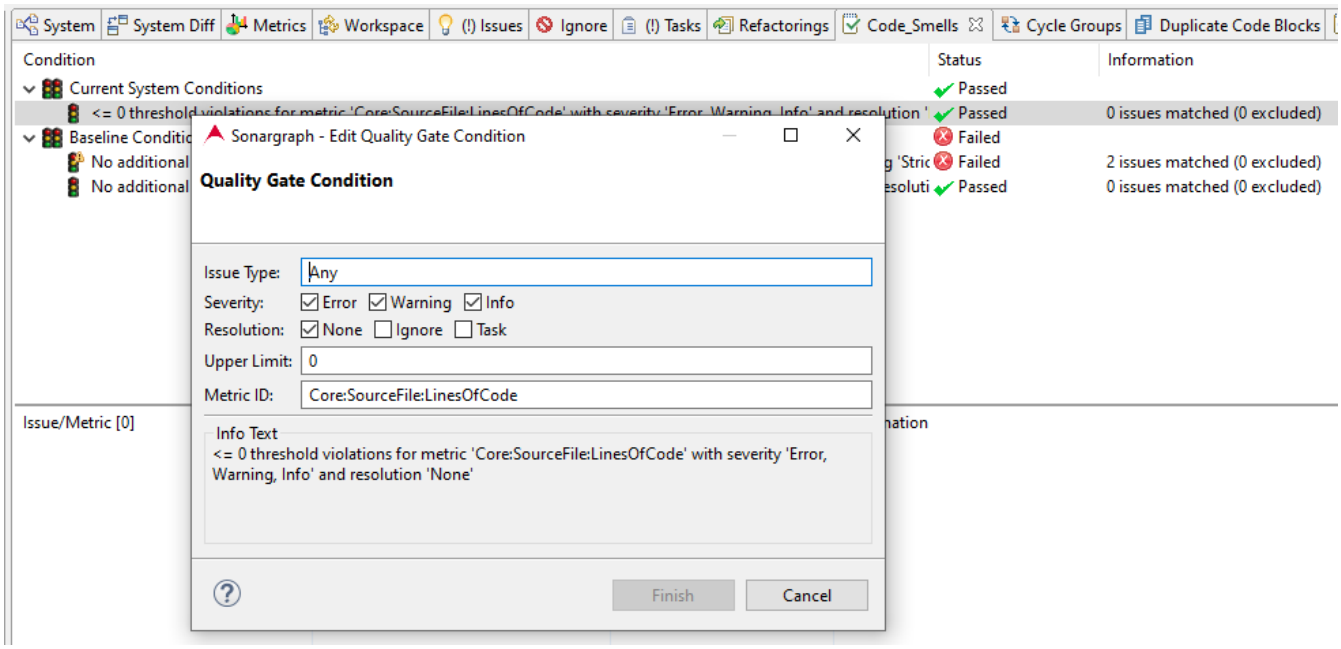


Figure 15.2. Current System Condition Dialog

TIP

To match threshold violations, simply specify a metric id and use the wildcard "any" as issue type.

Define Conditions With Respect to a Baseline

Conditions with respect to a baseline can either be defined based on issues or metric values. The "Baseline Issue Dialog" looks similar to the "Current System Condition" dialog and adds additional input fields for threshold violations.

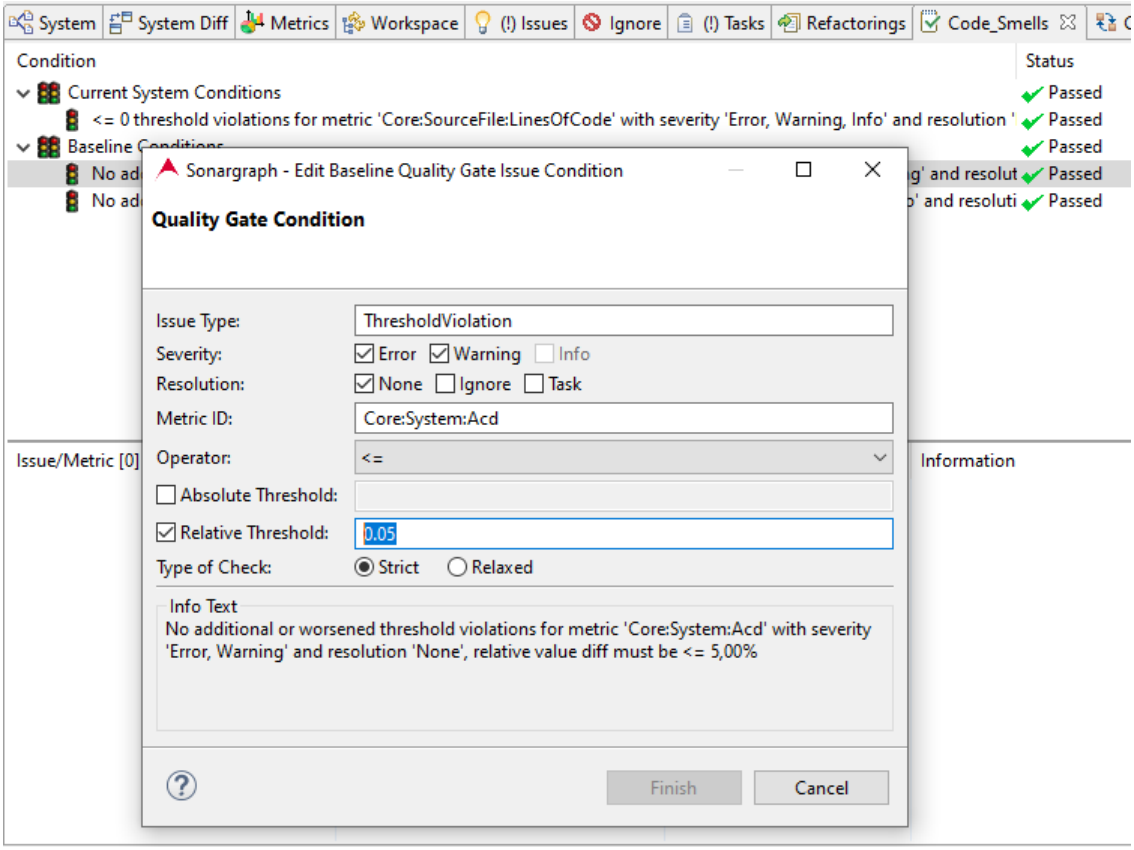


Figure 15.3. Baseline Issue Condition Dialog

TIP

If small changes to existing threshold violations should be tolerated, define an absolute or relative threshold for the metric value difference.

Some issues support a "relaxed" check, meaning that already existing issues that got slightly worse are tolerated. The following table provides the details about the effect of "relaxed" and "strict", an "X" means that the condition will fail for this change, "-" means that the change is tolerated:

Change	"Relaxed"	"Strict" (default)
Any added issue of severity error or warning.	X	X
Any issue that changed severity from warning to error.	X	X
Any issue whose resolution got removed.	X	X
Cycle group with more involved elements.	X	X
Cycle group with more parser dependencies to remove.	-	X
Threshold violation with a worsened metric value, if no diff threshold is defined.	-	X
Threshold violation with a worsened metric value, if the value diff is below the defined threshold.	-	X
Duplicate code blocks with more occurrences.	X	X
Duplicates code blocks with more involved lines.	-	X

Table 15.1. Effect of "Relaxed" and "Strict"

Define Baseline Metric Conditions

Values for some metrics will grow as more code is added to the system. An example are coupling metrics. It is nevertheless useful to monitor how much a metric value changes, as a big increase is usually an indicator for a bad design decision. Changes in metric values can be monitored via "Baseline Metric Conditions". The dialog allows the configuration of an absolute and/or relative threshold:

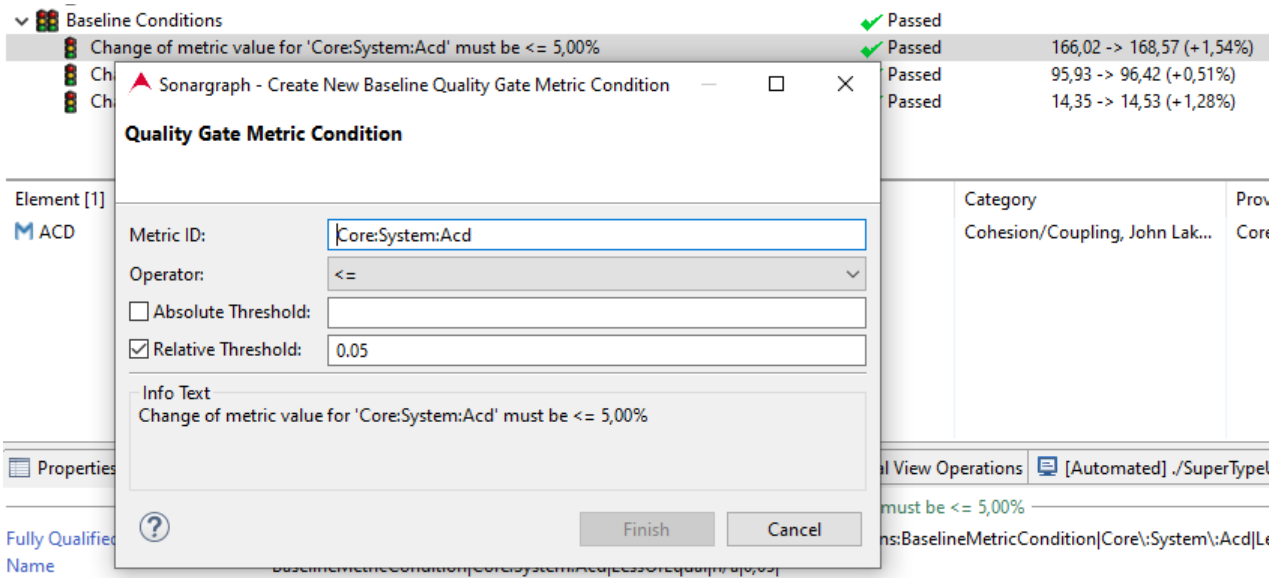


Figure 15.4. Baseline Metric Condition Dialog

NOTE

Only metrics on 'System' level are supported because only those values are currently part of the 'System Diff'.

Define Quality Gate Exclude Filters

For both sections of the quality gate it is possible to define "Exclude Filters". Issues matched by an exclude filter will no longer affect the outcome of any issue condition contained in the same section, i.e. either conditions for the current system's state or baseline conditions.

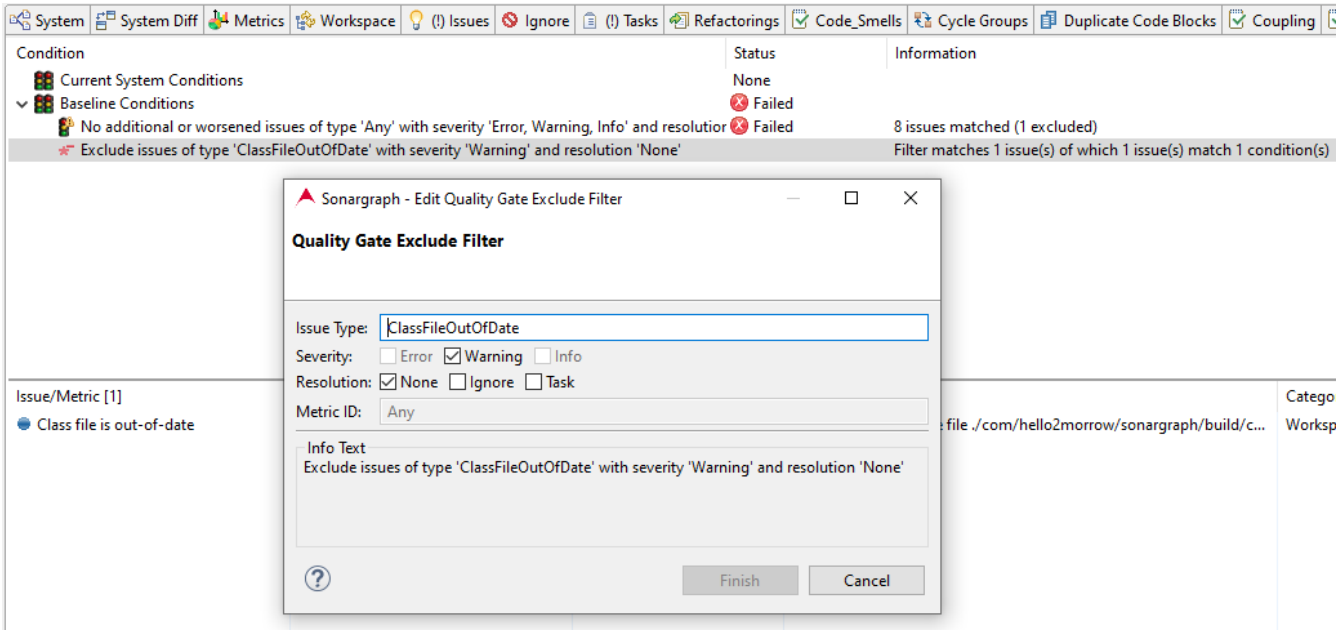


Figure 15.5. Exclude Filter Dialog

TIP

The column "Information" provides details about how many issues a filter matches and how many conditions are affected.

15.2. Using Quality Gates in the Continuous Integration (CI) Build

Sonargraph-Build can be used to enforce quality gates. All that is needed is a failSet configuration including the issue type "QualityGateIssue". The example below shows the XML configuration file for the shell integration of Sonargraph-Build. The same options exist for the other integrations (Ant, Maven, Gradle):

```
<sonargraphBuild
...
 <failSet>
     <failOnEmptyWorkspace>false</failOnEmptyWorkspace>
     <includes>
         <include>
             <issueType>QualityGateIssue</issueType>
         </include>
     </includes>
 </failSet>
</sonargraphBuild>
```

The result of the quality gate check is printed to the console (slightly formatted here):

```
...
Checking active quality gate(s)...
[Failed] Quality Gate 'No_Additional_CycleGroups'
    [Failed] Baseline Conditions:
        [Failed] Condition "Change of metric value for 'Core:System:CyclicComponents' must be
            <= 1,00 (absolute)" [0 -> 2 (+2)]
        [Failed] Condition "Change of metric value for 'Java:System:CyclicityPackages' must be
            <= 1,00 (absolute)" [0 -> 4 (+4)]

[Failed] Quality Gate 'No_Threshold_Violations'
    [Failed] Current System Conditions:
        [Failed] Condition "<= 0 threshold violations for metric 'Core:Type:SourceElementCount'
            with severity 'Any' and resolution 'None'" [5 issues matched (0 excluded)]
        [Passed] Condition "<= 0 issues of type 'ThresholdViolation'
            with severity 'Error' and resolution 'None'" [0 issues matched (0 excluded)]
    [Passed] Baseline Conditions:
        [Passed] Condition "No additional or worsened threshold violations for metric
            'Core:Type:SourceElementCount' with severity 'Any' and resolution 'None'"
            [0 issues matched (0 excluded, 0 tolerated)]
Check of quality gates failed.
Quality Gate Summary: 2 of 2 failed.
...
```

15.3. Current Quality Gate Limitations

As the Quality Gate feature is pretty new, there are still some things on our roadmap that will be implemented in the following releases:

- The Quality Gate analyzer is the last analyzer executed, therefore changes in quality gate issues are not part of the system diff.

Chapter 16. Extending the Static Analysis

Sonargraph presents the possibility to write Groovy scripts that will be run over the current *software system* in order to get specific results.

Scripts support the following use cases (among others): Create and calculate custom metrics, identify specific elements, list dependencies to methods, create issues for detected anti-patterns.

To get an idea of the Script API's power, it is recommended to examine the existing scripts contained in the provided quality models. For the core and each of the supported languages (Java, C#, C++) a script named VisitorExample.scr exists that illustrates the available Script API.

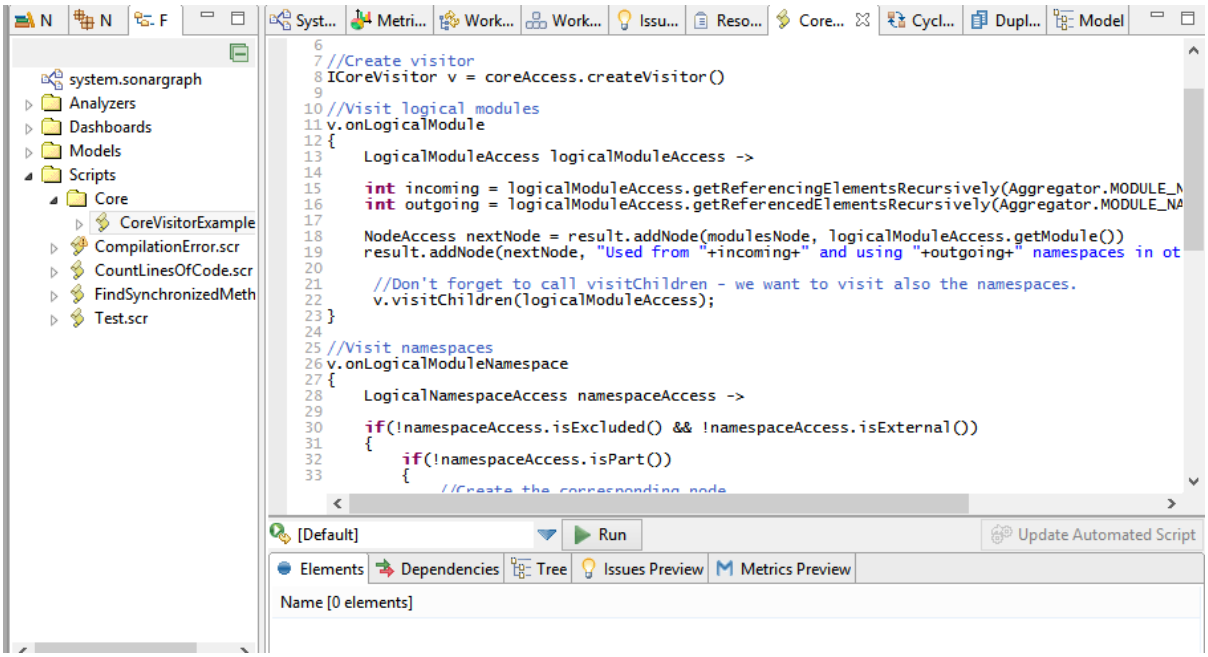


Figure 16.1. Script View

The Script view has two main sections:

- **Script editing area:** In this area you can write Groovy based scripts to retrieve information of your system in ways that would not be possible otherwise.
- **Results area:** Shows the results from the executed script which can be *software system* elements, dependencies between the different components of the system, a tree structure of elements, a list of issues, or metrics created by the script.

The execution of the current script can be triggered by clicking the button "Run" below the text edit area.

16.1. Interaction with Auxiliary Views

The Script view offers interaction with the Markers Auxiliary view which lists all markers of the script file. Typically those markers indicate compilation errors.

16.2. Groovy Scripts from Quality Model

When creating a new *system* , an existing quality model can be used, which usually contains some scripts. (See Section 6.4, “Quality Model”)

Figure 16.2. Quality Model

16.3. Creating a new Groovy Script

A new Groovy script can be created by "New" → "Other" → "Script" , or by selecting a Groovy script directory in the Files view (See Section 8.7, “Managing the System Files”) and choosing "New Script..." in the context menu. The "New Script Wizard" will open.

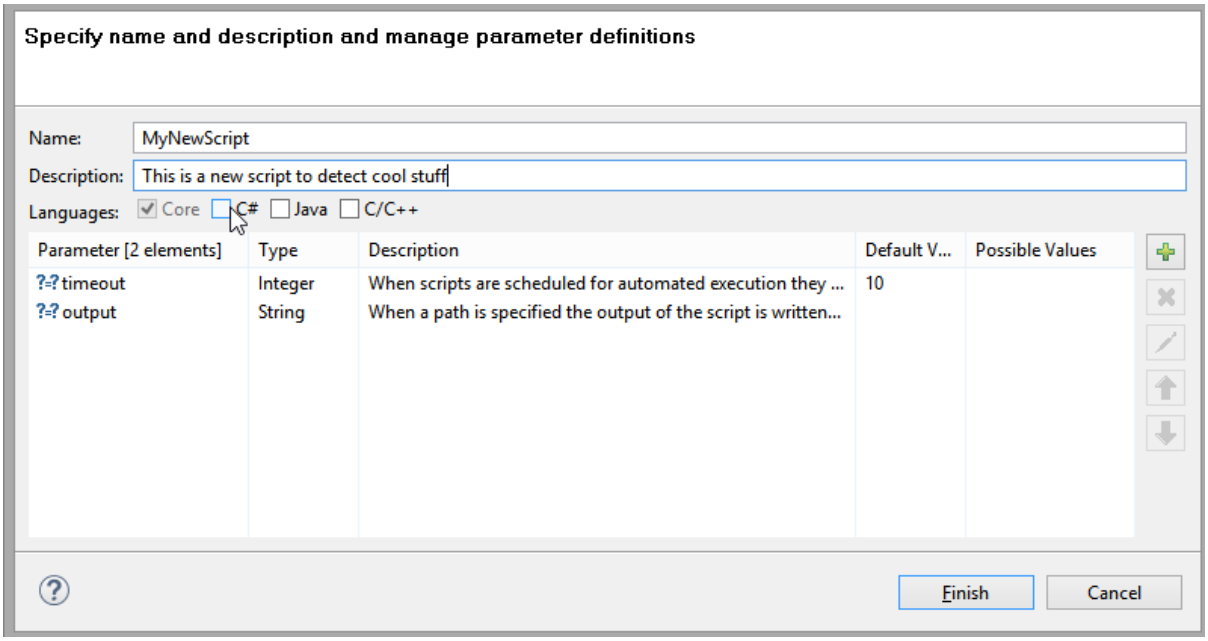


Figure 16.3. New Script

On the main page the following metadata of a Groovy script can be edited:

- the name of the Groovy script (must be unique in its directory)
- a description for the Groovy script
- a timeout value in seconds: whenever the Groovy script takes more time to run, it is stopped automatically
- the output file path where the textual output produced by println-Statements within the script is written.
- a list of APIs the Groovy script can use: "Core" contains functionality available to all languages, selecting any of the other languages offers additional functionality. Obviously, relying on a language specific API makes the script language dependent.

16.3.1. Default Parameters in a Script

Every Groovy script has a binding with some predefined parameters

- **out** the output stream of a Groovy script. Use **`out.println "message"`** or **`println "message"`** in the script. The output will appear in the Console view and in the output file (in case an output parameter has been specified).
- **result** of type Result Access (see JavaDoc) for adding the results of a Groovy script.
- **coreAccess** of type CoreAccess (see JavaDoc) for all Groovy scripts.
- **javaAccess** of type JavaAccess (see JavaDoc) for Groovy scripts using the Java API.
- **cppAccess** of type CppAccess (see JavaDoc) for Groovy scripts using the C++ API.
- **csharpAccess** of type CSharpAccess (see JavaDoc) for Groovy scripts using the C# API.

16.3.2. Adding Parameters

User defined parameters may be added to a Groovy script. In the Groovy script they can be referenced by their name, preceded by "parameter".

There are three different types of parameters:

- **String parameter** a default value can be given, a list of candidates (allowed values) can be given
- **Integer parameter** a default value can be given, a list of candidates (allowed values) can be given
- **Boolean parameter** allowed values are "true" or "false" (case insensitive)

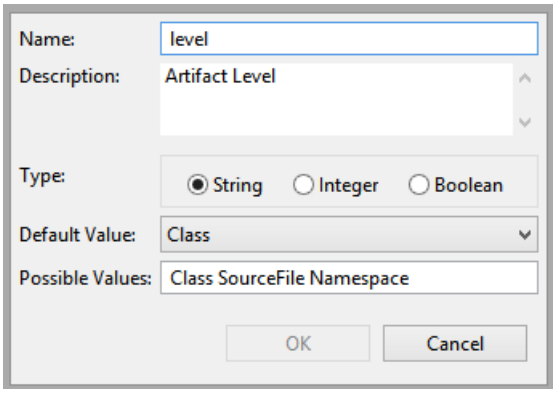


Figure 16.4. Parameter Definition

Now if you defined a parameter "level", it can be referenced in your script with name "parameterLevel":

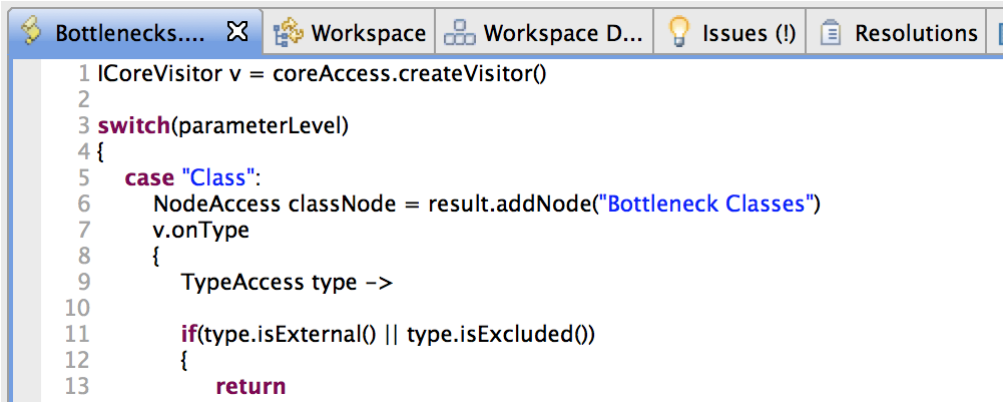


Figure 16.5. Parameter Usage in Script

16.3.3. Creating Run Configurations

Run Configurations allow a parameterized execution of a script. Right-click on a Groovy script in the Files view and select "New Run Configuration...".

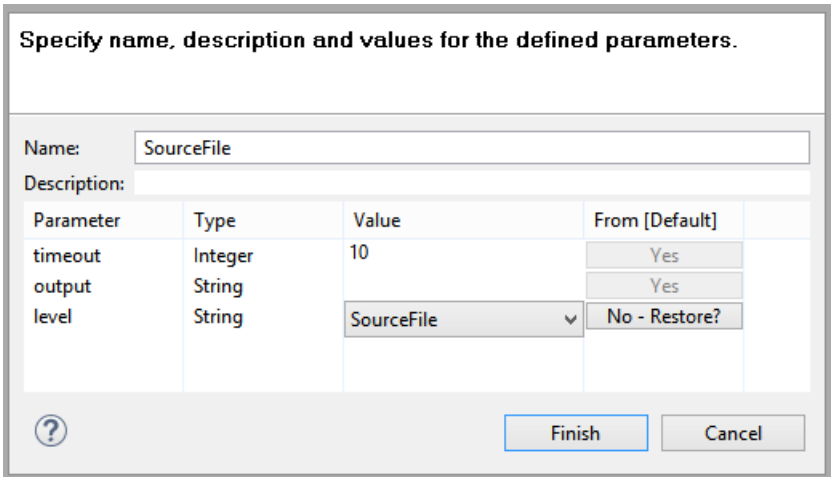


Figure 16.6. Create Run Configuration

A Run Configuration consists of

- a name
- a description
- a list of parameters and values that are inherited from the script's default Run Configuration.

Run configurations are used in two places: When a Groovy script is run manually, and when a Groovy script is run automatically. They are saved in the same directory as the script as <scriptname>#<runconfigname>.rcfg

16.4. Editing a Groovy Script

To edit the metadata (description, API use, parameters, run configuration) of a Groovy script, select the script and choose "Edit Script..." from the context menu, or press **F2** . To edit the Groovy script source code, open it in the Script view.

16.4.1. Auto Completion

To start auto completion in the Script view, place the cursor at the position you want autocompletion for and press CTRL+SPACE .

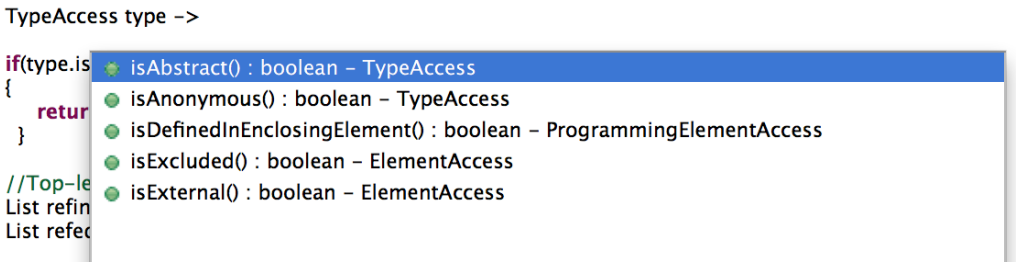


Figure 16.7. Auto Completion

Delayed Auto Completion

The first request for the auto completion might take several seconds to complete since some initialization needs to be done behind the scenes.

16.4.2. Compiling a Groovy Script

After editing a script the "Run" button changes to "Compile". Press "Compile" first, and after successful compilation the button will change its caption to "Run". If the Groovy script wasn't compiled successfully, there will be some Markers applied to the Groovy Script.

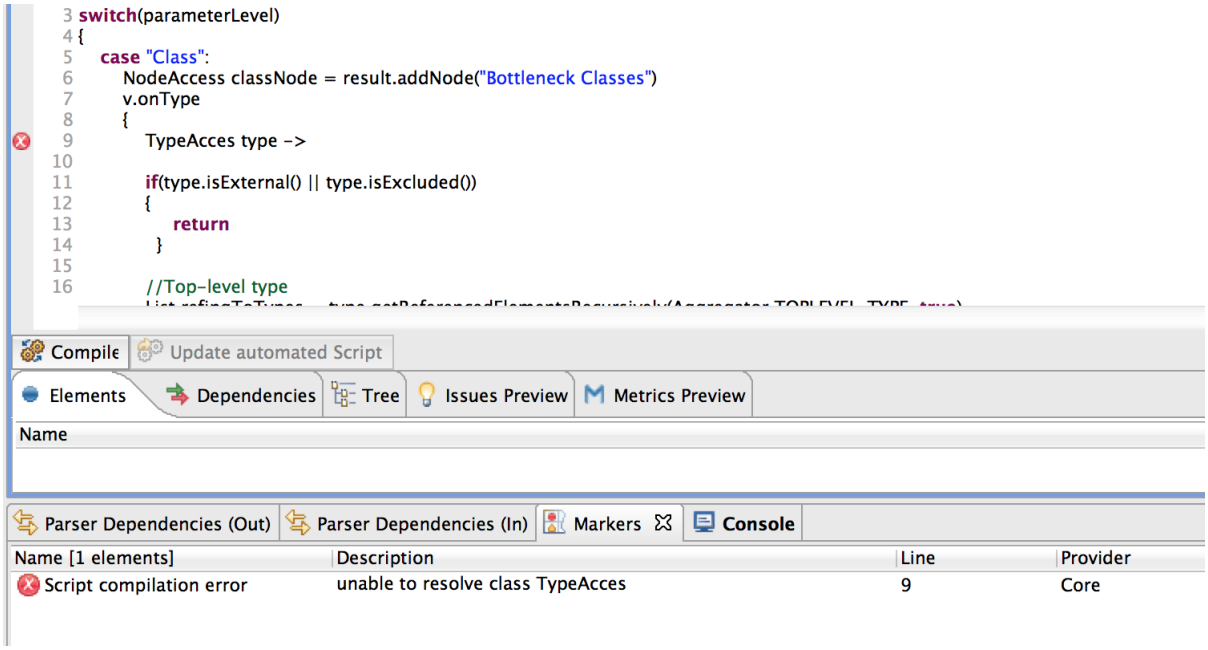


Figure 16.8. Script View Marker

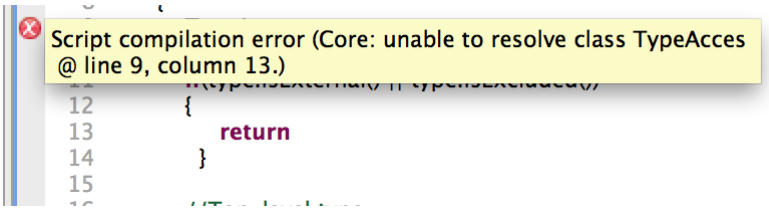


Figure 16.9. Script View Marker Tooltip

Update Automated Script

If this Groovy script is configured to be run automatically, the button "Update automated Script" will be active after a change, successful compilation and saving the script.

16.5. Producing Results with Groovy Scripts

Press the "Run" button to run a Groovy script manually. The combo box allows to change the Run Configuration to be used.

After a script was executed, the results of the scripts appear in five different tabs. The tabs that really hold results are marked with an exclamation mark. The class ResultAccess (see JavaDoc) provides methods to add different types of results.

The "Elements" and "Dependencies" tab hold a list of elements/dependencies, which were added by the script with

```
result.addElement()
```

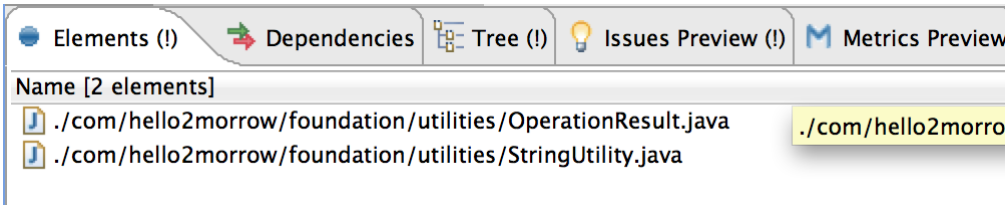


Figure 16.10. Script View Elements Tab

The "Tree" tab holds structure of nodes, which were added by the script with

`result.addNode()`

A node can have child nodes, or child elements.

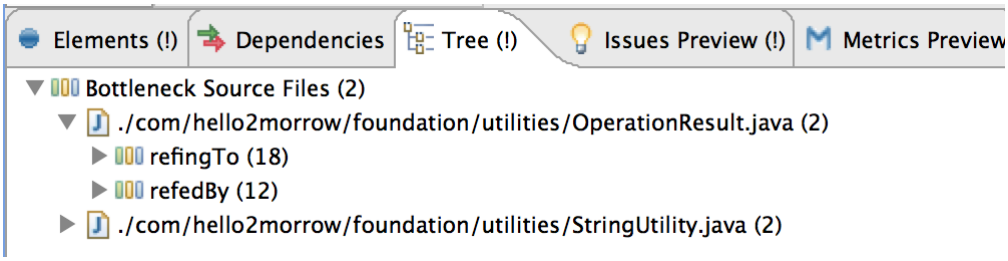


Figure 16.11. Script View Tree Tab

The "Issues Preview" tab shows a list of issues, which were added by the script with one of

`result.addInfoIssue()`

`result.addWarningIssue()`

`result.addErrorIssue()`

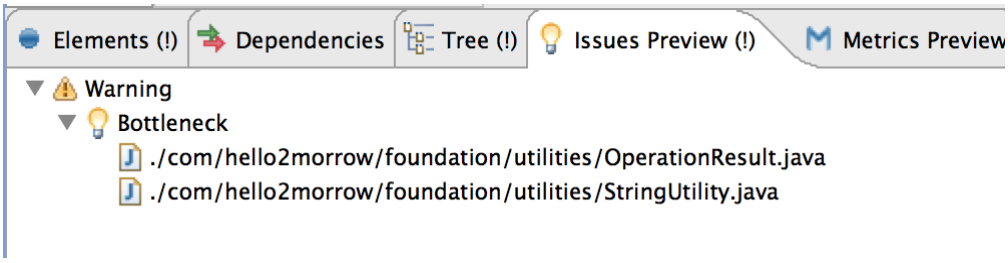


Figure 16.12. Script View Issues Preview

The "Metrics Preview" tab shows a list of metrics, which were added by the script with

```
MetricIdAccess id = coreAccess.getOrCreateMetricId(
    "SupertypeUsesSubtype",
    "Supertype uses subtype",
    "A super type must not know its subtypes",
    false /*non-float*/);
result.addMetricValue(id, coreAccess, warnings)
```

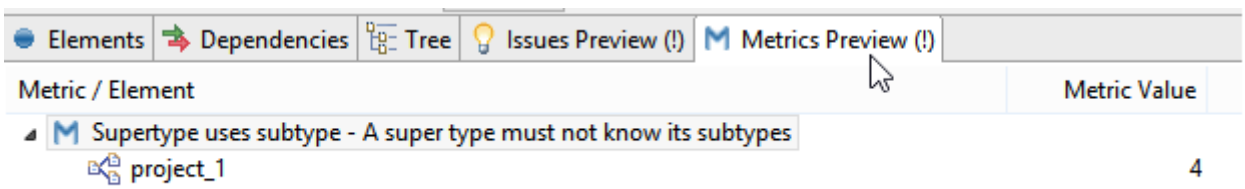


Figure 16.13. Script View Metrics Preview

16.6. Running a Groovy Script Automatically

It is possible to run Groovy scripts automatically whenever the workspace is refreshed. Go to "System" → "Configure..." → "Automated Scripts" and add the Groovy script + Run Configuration.

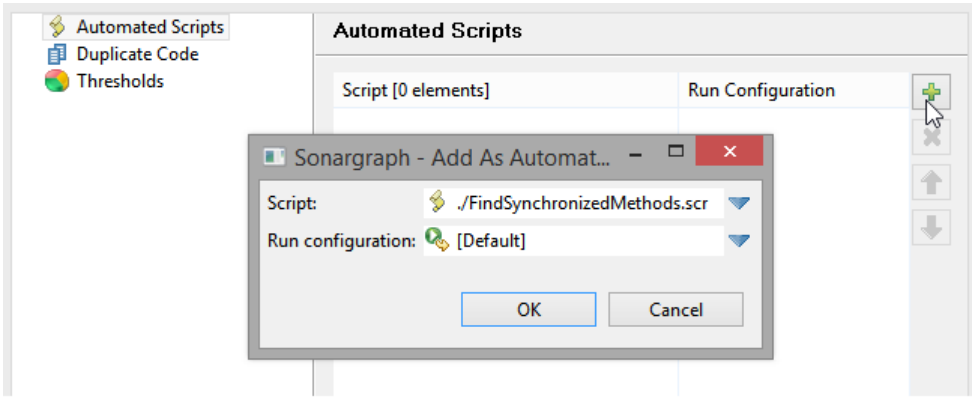


Figure 16.14. Script Runner

Run a Script With Different Run Configurations

It is possible to add the same Groovy script multiple times with different run configurations to the list of automated scripts.

Metrics and Issues

When a script running automatically creates a metric, this metric is displayed in the Metrics view. Executing the same script manually lets the metric show up in the "Metrics" tab of the Script view, but not in the Metrics view.

The same applies for any issues created during the script execution.

16.7. Managing Groovy Scripts

The Files view shows the organization of Groovy scripts. To add a new Groovy script directory, select an existing one (or the root directory "Scripts") in Files view, and choose "New Script Directory..." from the context menu.

To delete a script directory, select it in Files view, and press **DEL** , or select "Delete Script Directory" from the context menu. All contained Groovy scripts and Groovy script directories will be deleted recursively.

Single or multiple selected Groovy scripts can be deleted via **DEL** , or via the context menu.

Automated Scripts

If any of the deleted Groovy scripts was configured to be run automatically, it will be automatically removed from the list of automated scripts.

16.8. Groovy Script Best Practices

This chapter provides hints for improving the performance of Groovy scripts. This becomes more important the more scripts are configured to be executed automatically and thus run on every "refresh".

16.8.1. Only Visit What is Needed

The Script API uses the "Visitor" pattern to traverse the information of a software system. The pattern is very popular and explanations are easy to find.

Use the Right "visit" Method

Choosing the "visit" method that matches the script's purpose leads to fewer methods being called by the visitor and faster execution:

1. CoreAccess.visitLogicalModuleNamespaces() : Visits logical module namespaces and contained elements. See Section 5.4, “Logical Models” for details.

2. CoreAccess.visitLogicalSystemNamespaces() : Visits logical system namespaces and contained elements. See Section 5.4, “Logical Models” for details.

3. CoreAccess.visitParserModel() : Visits all elements of the parser model, i.e. no logical system or module namespaces.

4. CoreAccess.visitModel() : Visits all elements of the model. Most powerful, but obviously the most detailed and slow execution.

Only Visit Interesting Parts of the Model

If you are not interested in visiting externals or certain root directories, stop the visitor traversing that part of the model. The easiest way to exclude external elements from the analysis:

```
visitor.onExternal
{
    //We are not interested in external
    return;
}
```

Similarly, if you only want to investigate dependencies to external elements, you can stop the visitor from traversing the internal model:

```
visitor.onModule
{
    //We are not interested in internal
    return;
}
```

If you want to check only a specific module named "Test", you can do the following:

```
v.onExternal
{
    return;
}

visitor.onModule
{
    ModuleAccess module ->
    if (module.getName().equals("Test"))
    {
        //only visit children of this module
        visitor.visitChildren(module);
    }
}

visitor.onType
{
    TypeAccess type ->

    //Prints out only types of module "Test"
    println "Type $type";
}
```

The same approach should be used to limit the visiting of other model elements (e.g. namespace, component, type, method, field).

16.8.2. Find Text in Code

If you want to create metrics or issues based on text contained in source files, the visitor offers the method onSourceFile() and the class SourceFileAccess that provides access to individual lines.

Combined with regular expressions this is a very powerful method to identify anything in the code that is not contained in the model, e.g. FIXME or TODO in comments.

The following is an excerpt from the script FindFixmeAndTodoInComments.scr contained in the "Core" quality model:

```
def todoPattern = ~/\/\/\s?TODO.?\b/;
ICoreVisitor visitor = coreAccess.createVisitor();

visitor.onSourceFile
{
    ISourceFileAccess source ->
    if(source.isExcluded())
    {
        return;
    }

    List<ISourceLineAccess> lines = source.getSourceLines();
        ...
}
```

TIP

The compilation of the regular expression pattern is an expensive operation and should be done in the "global" section of a script, not within a visit() method.

TIP

Limit the number of scripts using SourceFileAccess. Sonargraph does not keep file contents in memory, thus visiting source files and traversing individual lines causes the actual files being opened. This is a costly operation and slows down the execution.

If you notice that various scripts contain source matching and this is time consuming, think about minimizing file operations by violating the "Single Responsibility Principle" and merge the functionality of several scripts into one.

Chapter 17. Using Additional Plugins

Sonargraph offers a plugin infrastructure, so that it is possible to extend *Sonargraph's* internal model and to create additional issues. Plugins can contribute to the internal model during the 'create model' and 'create dependencies' phases and create issues during the 'analysis' phase. Currently the following plugins are available:

- Spring Microservices
- Swagger
- Spotbugs
- PMD
- Issues Importer

17.1. Plugin Configuration

Configuration of a *Sonargraph* plugin is system specific, and stored in *Sonargraph's* system file folder 'Plugins' as file '<pluginId>.xml'. For every plugin there will be an initial default configuration in memory with the 'Enabled' parameter set to 'false'. The configurations can be edited either by double clicking the corresponding configuration or by using the menu 'System->Configure...' and locating the corresponding plugin property page manually. Changing the default configuration will create the corresponding file on disk containing the settings.

17.2. Spring Microservices Plugin

The 'Spring Microservices' plugin for Java exposes web resources of SpringBoot applications and dependencies between them. It finds exposed web service end points by looking at annotations like org.springframework.web.bind.annotation.RequestMapping. If a method is annotated with one of those annotations the web resource will be added as a child to the method in the Navigation view.

Currently, clients using the Spring FeignClient annotations are detected as 'Web Call' elements. More client frameworks will be added in the future. For each of those methods, a 'Web Call' child element is created, and a corresponding 'Web Resource' element is tried to be found in the workspace. The web resources could have been created by another plugin. If no matching 'Web Resource' is found, an 'External Web Resource' element is created as child element of the plugin's external node.

SpringBoot offers various ways of configuration. The plugin currently expects a standard directory layout for SpringBoot modules, with configuration files (application.properties, application.yml, bootstrap.properties, bootstrap.yml) contained in the module's 'src/main/resources/' directory. It can also analyze configuration files contained in SpringBoot applications annotated with org.springframework.cloud.config.server.EnableConfigServer. Currently, configuration loaded from classpath is supported (spring.cloud.config.server.native.search-locations = classpath:/shared).

Please contact us if you have a use case and need some support!

The following screenshots have been created for the *Piggy Metrics* application.

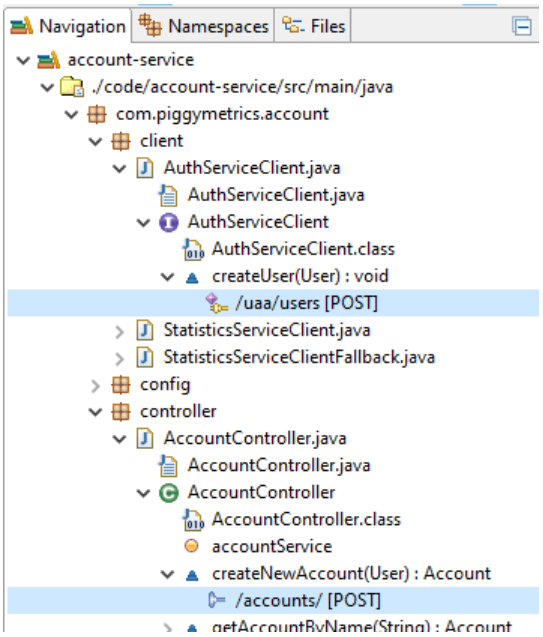


Figure 17.1. Spring Microservices Plugin Web Call (above) and Web Resource (below)

Dependencies between web calls and web resources are treated like any other dependency and can cause architecture violations.

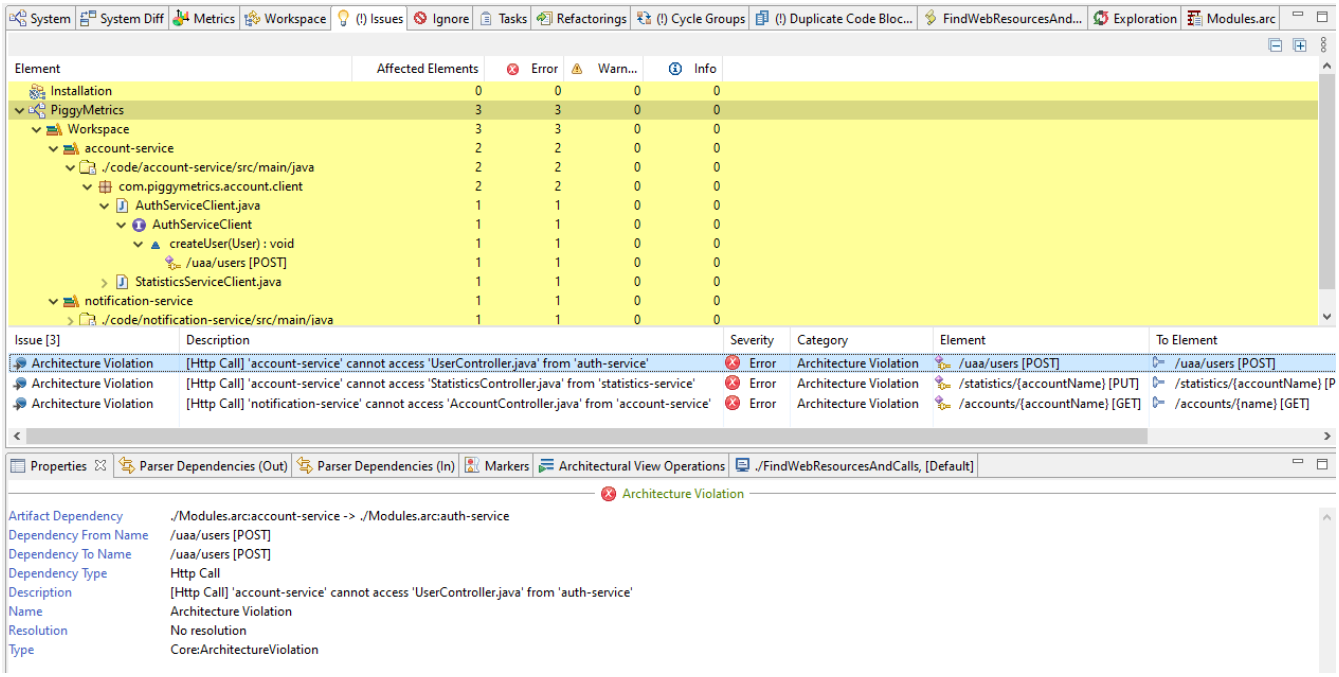


Figure 17.2. Architecture Violations for Dependencies between Plugin Elements

Related topics:

- Chapter 18, *Investigating Microservice Dependencies*
- Section 17.3, “Swagger Plugin”

17.3. Swagger Plugin

The 'Swagger' plugin for Java exposes web resources and dependencies between them. It finds exposed web service end points by looking at the javax.ws.rs.Path annotation. If a method is annotated with this annotation the web resource will be added as a child to the method in the Navigation view.

The much more difficult part is to find out who is calling those web end points. Right now the plugin detects calls generated by Swagger for the OkHttpClient framework. The generated client code and its class files need to be added to the Sonargraph workspace. The plugin scans the generated Java code for web calls, creates a 'Web Call' element as a child element of the originating method, and tries to resolve the web resource within the scope of the project. The web resources could have been created by another plugin. If no web resource is found the called end point will show up under the "External (Web)" node in the navigation view.

With the help of the plugin you can visualize the dependencies between your web/micro-services and also define an architectural model that would enforce restrictions on those dependencies. To achieve that just create a big Java project containing the code of all your web/micro-services. The Swagger plugin will then automatically add web/micro-service dependencies to the Sonargraph model.

Please contact us if you have a use case and need some support!

Related topics:

- Chapter 18, *Investigating Microservice Dependencies*
- Section 17.2, “Spring Microservices Plugin”

17.4. SpotBugs Plugin

SpotBugs (successor of FindBugs) looks for 'bugs' in Java code.

17.5. PMD Plugin

PMD finds 'violations' (common programming flaws) in Java code.

The user can specify custom rule sets for the PMD plugin as comma separated entries of the 'Rule Sets' field on the corresponding property page. If no rule set is specified for the PMD plugin an internal default rule set is used.

NOTE: The path of a custom rule set needs to start with either './' or '../'. Those relative paths are resolved relative to the parent directory containing the Sonargraph system folder. E.g.: If the system is called 'MySystem' those relative paths are resolved relative to the parent directory of the directory 'MySystem.sonargraph'.

17.6. Issues Importer Plugin

The plugin allows you to import issues generated by other tools into Sonargraph.

In the corresponding property page you can configure the following:

- A comma separated list of csv files. The file names should be relative to the Sonargraph system directory. Each line in these files should have five columns separated by semicolons. The columns are line number, column number, affected file (absolute path), error code as an integer and the error message itself.
- The ranges of error codes that would lead to an error, a warning or and info level issue. Ranges are described as a comma separated list of integer ranges, e.g. '1-500, 600-999'

Chapter 18. Investigating Microservice Dependencies

A lot of applications have been developed around 'Microservices'. One big advantage of Microservices is their loose coupling via HTTP(S), which can turn into a disadvantage because dependencies between a large number of services are hard to track.

Sonargraph exposes the dependencies via its 'Spring Microservices' and 'Swagger' plugins. All detected 'Web Resources' and 'Web Calls' can be listed with the script 'Core/FindWebResourcesAndCalls.xml' which can be imported from the built-in quality model. Executing the script, multi-selecting all found elements in the 'Elements Tab' of the Script View and opening the Exploration View via the context menu creates a nice dependency overview between microservices as shown in the following screenshot for the *Piggy Metrics* application.

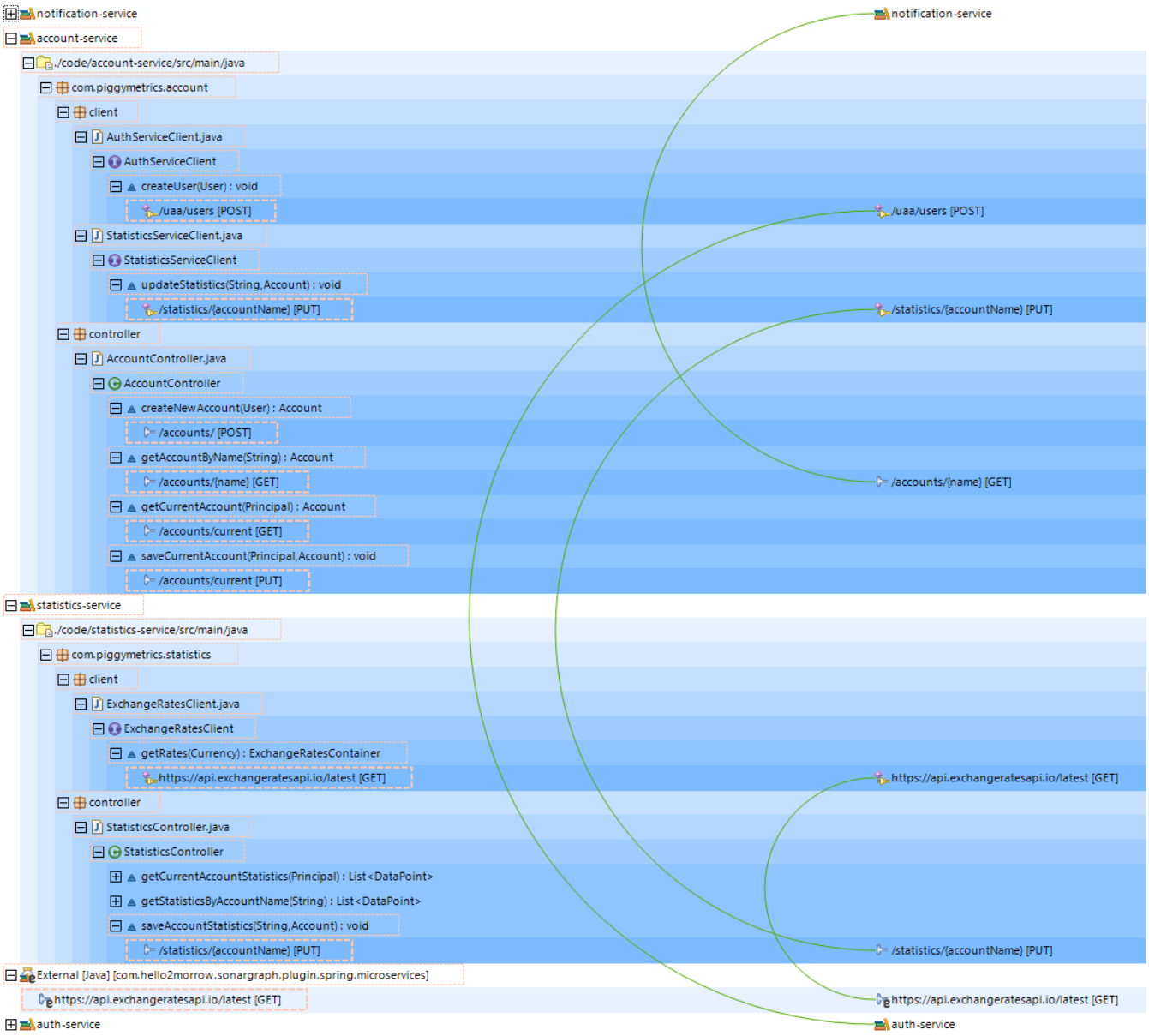


Figure 18.1. Exploring Microservice Dependencies

NOTE

With the help of the plugins and the script you cannot only visualize the dependencies between your microservices, but also define an architectural model that would enforce restrictions on those dependencies. To achieve that just create a big Java project containing the code of all your microservices.

Currently, calls and resources are detected for the above mentioned Java frameworks (see the detailed sections for implementation details) as a starting point. Dependencies are resolved between web calls and web resources detected by a plugin for any of the supported languages.

If you have the need for the support of a specific framework, please get in contact with us via `<support@hello2morrow.com>`!

Related topics:

- Chapter 17, *Using Additional Plugins*
- Section 17.2, “Spring Microservices Plugin”
- Section 17.3, “Swagger Plugin”
- Chapter 16, *Extending the Static Analysis*

Chapter 19. Build Server Integration

Several integrations exist to run the same Sonargraph quality checks on your build server. Sonargraph-Build can be downloaded from our web site: *https://www.hello2morrow.com/products/downloads* Integrations are available to start Sonargraph using Ant, Maven, Gradle or Shell scripts. Plugins are available to visualize the results in *SonarQube* and *Jenkins* . More details about configuration options can be found in the user manual of Sonargraph-Build.

If you want to analyze a Java system with Ant on the build server, chances are high that the workspace definition contains class root directories of the development environment and that those directories are not available on the build server. The following section describes how workspace profiles can be used to solve this problem: Section 8.8.3, “Creating Workspace Profiles for Build Environments”

Related topics:

- Section 8.8.3, “Creating Workspace Profiles for Build Environments”

Chapter 20. IDE Integration

The purpose of the IDE integrations of *Sonargraph* is to run the quality checks continuously during development. This helps to prevent new problems being introduced into the shared code base: It is not needed to wait for the build server to report any problems, but instead the IDE integrations of Sonargraph run quality checks in the background, whenever the IDE compiles Java code. Problem and task markers are created for issues and resolutions and support the developer to fix the problems.

NOTE

The IDE must be started at least with a Java 21 runtime for the integration to work.

To ease navigating between Sonargraph and the Eclipse IDE, a remote selection mechanism has been introduced with version 11.1. This enables quick navigation to the right spot to fix something, when analyzing the code base with Sonargraph. And also the other way, when the advanced visualization mechanisms of Sonargraph need to be used to get a better understanding while coding. More details are provided in Section 20.3, “Collaboration between Sonargraph and IDE”.

Currently, the IDE integrations only support Java systems.

20.1. Eclipse Plugin

To install the Sonargraph Eclipse plugin, run Eclipse and open menu "Help" → "Install New Software...". Add this update site as a new location: *https://eclipse.hello2morrow.com/SonargraphEclipse.site*

After successful installation and a restart of Eclipse the additional menu entry "Sonargraph" should be visible. If not, check the Eclipse "Error View" for any errors related to the plugin and get in contact with <support@hello2morrow.com>.

NOTE

Installing Sonargraph Eclipse plugin on Eclipse Oxygen with an already installed Groovy plugin may lead to some Eclipse editors or views showing errors after a restart of Eclipse, due to Groovy plugin's compiler resolver being broken when there are multiple Groovy compiler bundles for the same Groovy compiler version. In this case it may help to delete the Groovy compiler bundle introduced by Sonargraph Eclipse plugin from plugins folder of your Eclipse Oxygen installation. If you need further assistance please get in contact with <support@hello2morrow.com>.

TIP

Occasionally, Eclipse gets confused after installing plugins. If Eclipse fails to startup, configure the "-clean" startup option in the eclipse.ini file within you Eclipse installation. This will clear any cached data. If the slightly longer startup time bothers you, remove the option again. More details are available here: *https://help.eclipse.org/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fmisc%2Fruntime-options.html*

Activation

You need to have a valid license or activation code in order to use the plugin. More details can be found in Chapter 3, *Licensing*.

Open the dialog via the menu "Sonargraph" → "Manage License..." and supply either the activation code or license file and hit "Request".

The Sonargraph icon in the Eclipse toolbar indicates the current status of the plugin and its tooltip provides additional information. This makes it easy to spot, if the plugin is still analyzing, if there are any issues, etc.

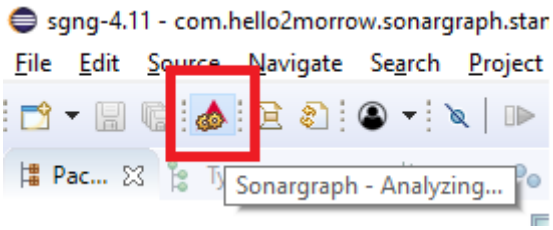


Figure 20.1. Status Icon

The following sections describe common interactions and usage of the plugin.

NOTE

The IDE must be started with a Java 21 (or higher) runtime for the integration to work.

We tested the plugin successfully with Eclipse 4.32.0. If you notice any compatibility problems during installation, please send us the Eclipse error log or a screenshot of the error and details about your Eclipse installation to `<support@hello2morrow.com>`.

20.1.1. Assigning a System

The next step after the successful installation and activation of the plugin is to open a Software System that has been previously created using *Sonargraph*. Open the menu "Sonargraph" → "Open System..." and select the Sonargraph system.

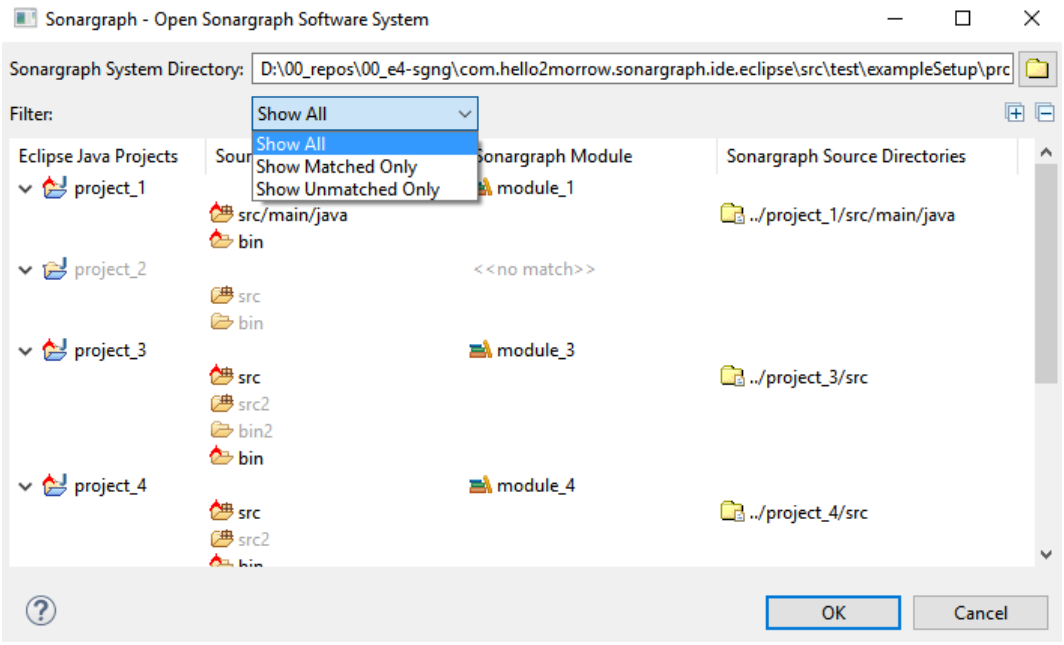


Figure 20.2. Open Sonargraph System

The red decorators and black font indicate, which Eclipse projects could be mapped to Sonargraph modules and which source and class directories are part of the Sonargraph workspace. The mapping is done based on matching source root directories. Projects and directories that are not part of the Sonargraph analysis are indicated by a gray font.

20.1.2. Displaying Issues and Tasks

NOTE

The plugin currently always applies the default virtual model "Modifiable.vm".

NOTE

The number of Sonargraph issues and resolution markers might differ from the number of issues and resolutions displayed in the Sonargraph application for the following reasons:

- Individual markers are created and attached to source files for each duplicate code block occurrence and each component involved in a component cycle group. This makes it easier for the developer to spot a problem while editing a source file, but results in a higher number of markers.
- No markers are created for ignored issues, because the developer cannot resolve them in the IDE.
- Markers are only generated for elements that are part of the currently monitored workspace. If a Sonargraph module cannot be mapped to an Eclipse project, no issues and resolutions for elements contained in that module are shown.

Detected issues are shown in the standard Eclipse Problems and Tasks views. The view options allow to group problems by "Type" as shown in the screenshot.

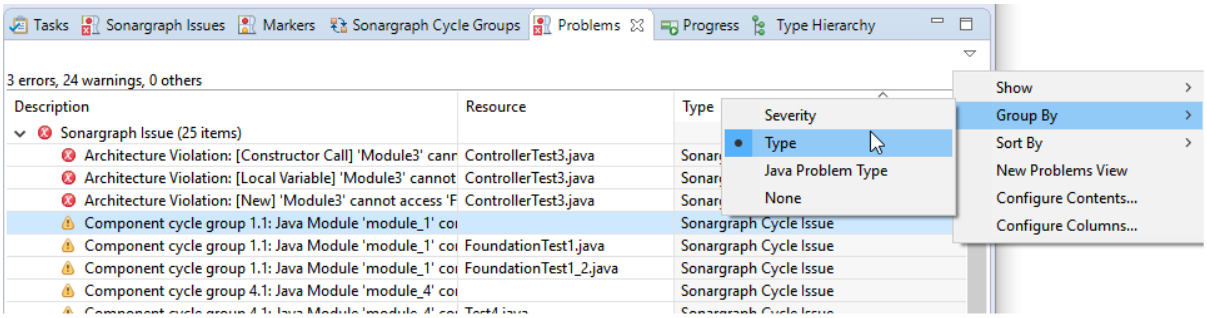


Figure 20.3. Show Issues in Problems View

It is also possible to configure a new Problems view via the Problems view's view menu and exclusively show the Sonargraph issues by selecting "Configure Contents..." and filtering for the Sonargraph issues as shown below in the screenshot. This configuration dialog can be opened via Problems view's view menu "Configure Contents...".

TIP

The same grouping and filtering options are applicable on the standard Eclipse Tasks and Markers views.

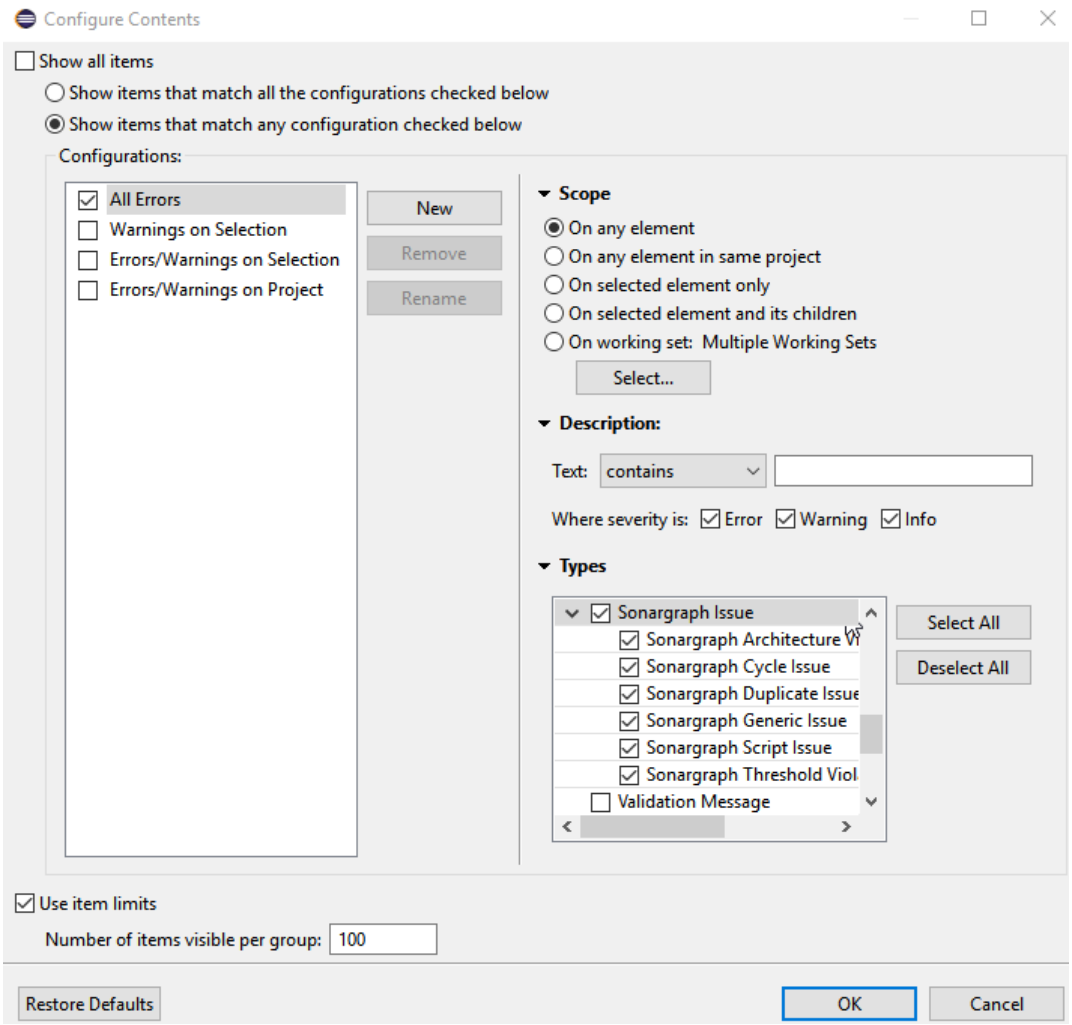


Figure 20.4. Problems View Configuration for Sonargraph Issues

Examining Cycle Group Issues

Sonargraph calculates logical namespace cycle groups, i.e. physical namespaces are merged on module or system level. The Sonargraph Cycle Groups view can be opened via the main Sonargraph menu or via the context menu of a Sonargraph Cycle Issue marker.

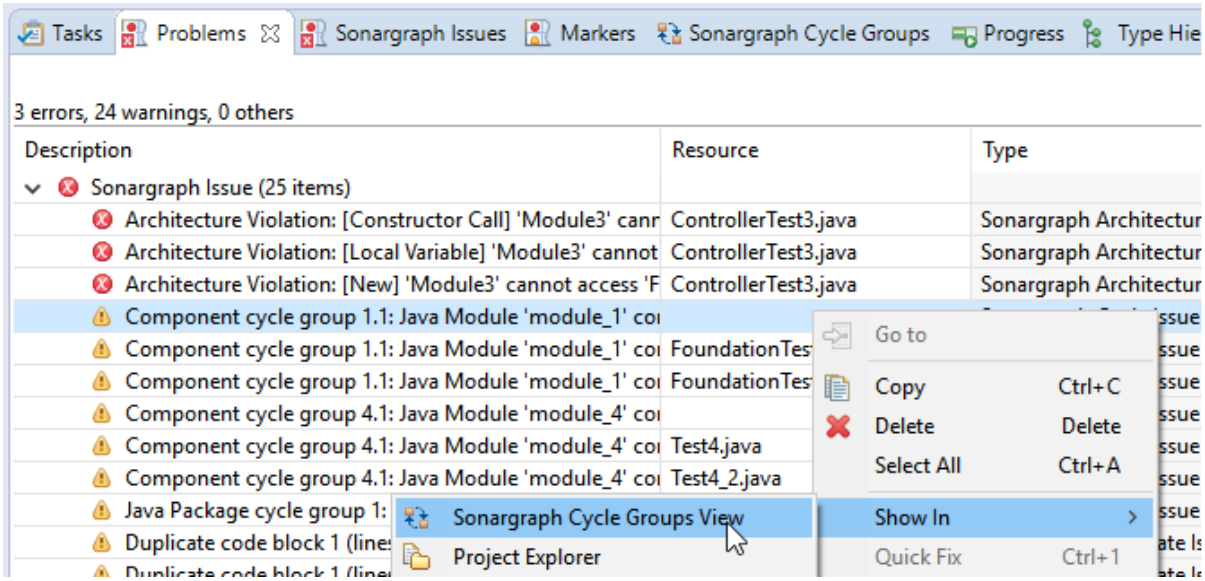


Figure 20.5. Context Menu To Open Sonargraph Cycle Groups View

More detailed cycle group analysis (including possibilities to break them up) should be done with the Sonargraph-Architect application as described in Section 8.10, “Analyzing Cycles”.

20.1.3. Suspending / Resuming Quality Monitoring

The Sonargraph quality checks are executed as an additional "builder" in the background whenever the project is built. If this is too time consuming or you are currently not interested in the Sonargraph checks, the plugin can be disabled quickly via the menu "Sonargraph" → "Suspend Analysis". This is equivalent to closing the system. Once the checks should be resumed, simply select "Sonargraph" → "Resume Analysis". This is equivalent of opening the system from snapshot and doing a refresh.

NOTE

If the class path of a monitored Eclipse project is modified or a monitored project is closed, the monitoring is automatically suspended. Resume the monitoring, once you are finished with the workspace modifications.

20.1.4. Setting Analyzer Execution Level

The Sonargraph Analyzer Execution Level can be set via the menu "Sonargraph" → "Analyzer Execution Level".

20.1.5. Getting Back In Sync with Manual Refresh

If you updated Sonargraph system files in parallel using the Sonargraph-Architect application, you can choose "Sonargraph" → "Refresh System Files" to just update those resources.

If you notice that some markers are not properly updated, or that the Sonargraph analysis has not picked up the latest changes, please use the menu "Sonargraph" → "Reparse System" to bring the Sonargraph model back in sync with the Eclipse workspace.

Since Eclipse caches resources sometimes, you might see Sonargraph "Class file out of date" issues on Eclipse startup. A "refresh" of the Eclipse workspace followed by a build should solve it. If the changes are not picked up by Sonargraph, trigger a manual "reparse" as described above.

TIP

If you notice any problem using the plugin, we a grateful to receive your feedback! The easiest way is to use the menu "Sonargraph" → "Send Feedback".

TIP

We think *assertions* are really helpful to ensure proper program execution and we are using them a lot in Sonargraph. You can enable assertions for Eclipse by adding the -ea VM argument at the end of your eclipse.ini configuration file.

An error dialog will show up if an assertion error happens. Please take the opportunity to let us know about the error! We will do our best to fix it as soon as possible.

20.1.6. Examining Changes

Similar to Sonargraph application, the Eclipse plugin allows to track changes of issues with respect to a baseline. The Sonargraph menu allows the following interactions:

- New Baseline: Create and apply a new baseline, i.e. at the beginning of a feature development, to ensure that no new issues are introduced.
- Open Baseline: Open an existing baseline, i.e. an XML report generated at the end of the previous release.
- Activate System Baseline: Switch to the baseline that is configured in the software system. Obviously, this menu is only enabled if there is a baseline configured and it is currently not activated.
- Detach Baseline: Disconnect the current Sonargraph system from the baseline to see all existing issues.
- Export HTML Report: Create and open an HTML report focussed on the differences w.r.t the baseline.

Sonargraph issues are converted to Eclipse problem markers. If a baseline is applied, all unmodified issues are assigned the severity "info". This sets them clearly apart from the added or changed issues which keep their original severity. The following screenshot shows an Eclipse Problems view that has been configured to focus on Sonargraph issues only:

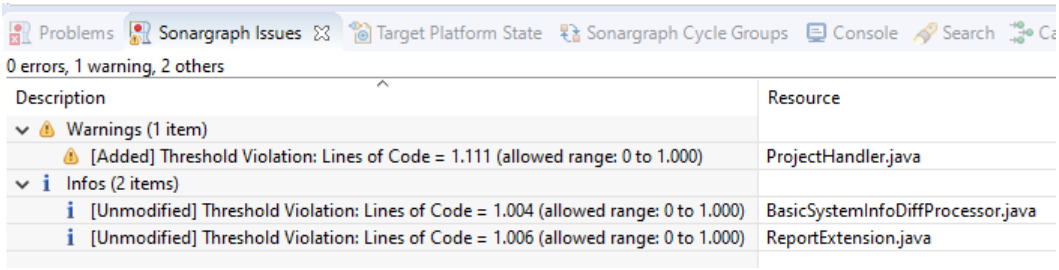


Figure 20.6. Sonargraph Issues in Eclipse with Baseline Applied

There are no markers created for resolved issues. If you are interested which issues have been resolved, you need to create the HTML report.

Sonargraph tasks and refactorings are also converted to Eclipse task markers. The change info (Added, Unmodified, etc.) is prepended to the tasks description.

Current Limitations

Not all functionality related to the system diff has been implemented yet in the Eclipse plugin and the following list summarizes the current limitations, which will be resolved in future releases:

1. The "Sonargraph Cycle Groups" view does not support the system diff and always shows the complete list of cycle groups. To see diff information about cycle groups, you currently have to generate the HTML report.
2. The "Sonargraph Refactorings" view does not support the system diff and always shows the complete list of refactorings. To see diff information about refactorings, you currently have to generate the HTML report.

Related topics:

- Chapter 14, *Examining Changes*

20.1.7. Execute Refactorings in Eclipse

The list of Sonargraph refactorings definitions is shown in the Sonargraph Refactorings View. The view can be opened from the Eclipse main menu "Window" → "Show View" → "Other...". Select the folder "Sonargraph" and select "Sonargraph Refactorings".

The "Sonargraph Refactorings" view offers filter options in the top right corner. Refactorings can be filtered by status, priority, assignee and description.

NOTE

Refactorings defined in Sonargraph might affect a lot of resources. We recommend committing all pending changes to your version control system before executing the refactorings, so you have a safe fallback.

NOTE

Execute the refactorings in the order of their definition. Otherwise subsequent refactorings might no longer be applicable.

The Sonargraph plugin delegates the refactorings to the refactoring mechanism of Eclipse. Some Sonargraph refactorings cannot be converted into a single Eclipse refactoring. The following refactorings need to be split:

1. Namespace refactorings that effectively merge two packages by moving or renaming a package into an existing target package.
2. Move refactorings that change the source root of a namespace or compilation unit.
3. Move refactorings of a package containing subpackages.
4. Move+Rename refactorings are not supported as an atomic operation in Eclipse and need to be split.

The following steps are executed for each refactoring:

1. If the Sonargraph refactoring needs to be split, a confirmation dialog will inform you about the necessary actions.
2. The standard Eclipse refactoring dialogs are shown that allow you to control the affected resources (e.g. change names in non-Java files) and preview the changes.
3. If you chose to deviate from the planned refactoring, a dialog prompts you to add a comment.

   ## NOTE

   Subsequent Sonargraph refactorings might become obsolete if you deviate from the planned refactoring!

4. The refactoring log containing the list of changed resources is shown in the end. You can copy&paste these details as a protocol e.g. into your task management system.

Related topics:

- Chapter 10, *Simulating Refactorings*
- Section 10.4, “Best Practices”

20.2. IntelliJ Plugin

To install the *Sonargraph* Intellij plugin run Intellij, open the Intellij Settings dialog, go to "Plugins" → "Browse repositories..." → "Manage repositories..." and add a new repository supplying hello2morrow's Intellij plugin repository URL: *http://intellij.hello2morrow.com/sonargraphIntelliJ/updatePlugins.xml*

Once the repository is configured, select *Sonargraph* from the plugin list and click on the green install button in the description area.

After successful installation and a restart of Intellij, the *Sonargraph* entry should appear under the "Other Settings" node in Intellij's setting dialog. If not, check the Event Log view and Intellij's notifications for any errors related to the plugin and get in contact with <support@hello2morrow.com>.

NOTE

Intellij version 2018.2 or newer is required for *Sonargraph* Intellij plugin to run.

Activation

You need to have a valid license or activation code in order to use the plugin. More details can be found in Chapter 3, *Licensing*. Open the dialog via Intellij's settings, then go to "Other Settings" → "Sonargraph" → "Manage License..." and supply either a license file or the activation code, hit "Request" and "Install License" before closing the dialog.

The following sections describe common interactions and usage of the plugin.

NOTE

The IDE must be started with a Java 21 (or higher) runtime for the integrations to work.

We tested the plugin successfully with IntelliJ 24.2. If you notice any compatibility problems during installation, please send us the IntelliJ error log or a screenshot of the error and details about your IntelliJ installation to <support@hello2morrow.com>.

20.2.1. Assigning a System

The next step after the successful installation and activation of the plugin is to open a Software System that has been previously created using *Sonargraph*. On the Intellij settings, go to "Other Settings" → "Sonargraph" and select the Sonargraph system.

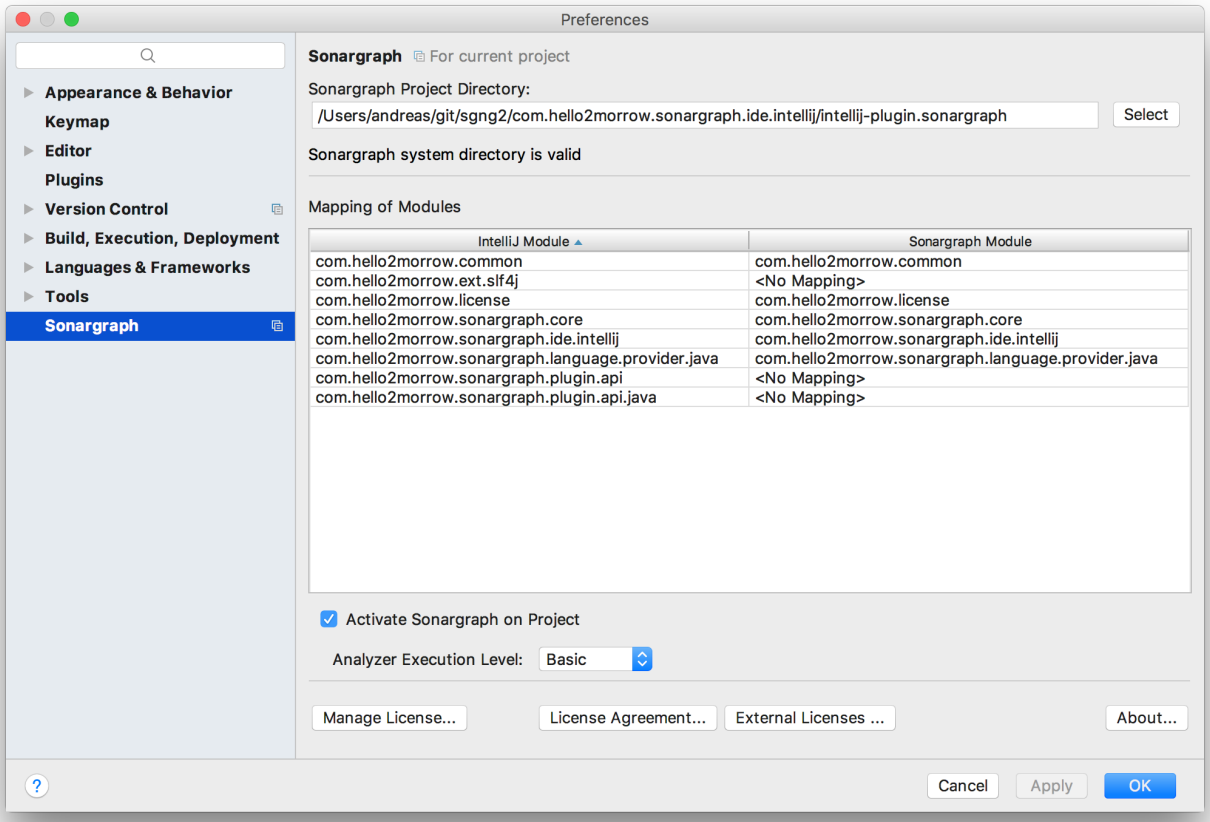


Figure 20.7. Open Sonargraph System

Sonargraph will match its own modules to Intellij's modules based on source root directories. The "Mapping of modules" table shown above indicates the result of the matching process.

Once the system is opened and the matching is completed, use the "Activate Sonargraph on Project" to enable *Sonargraph's* analysis and user interface components in your Intellij IDE.

The "Analyzer Execution Level" can be set to one of "Full", "Advanced", "Basic", or "Minimal". The tooltip shows which Analyzers will be run for each of the levels.

20.2.2. Displaying Issues and Tasks

NOTE

As of now, the plugin always applies the default virtual model "Modifiable.vm".

NOTE

The number of *Sonargraph* issues and resolution markers might differ from the number of issues and resolutions displayed in the *Sonargraph* application for the following reasons:

- Individual markers are created and attached to source files for each duplicate code block occurrence. This makes it easier for the developer to spot a problem while editing a source file, but results in a higher number of markers.
- No markers are created for ignored issues, because the developer cannot resolve them in the IDE.

- Markers are only generated for elements that are part of the currently monitored workspace. If a *Sonargraph* module cannot be mapped to an Intellij project, no issues and resolutions for elements contained in that module are shown.

Detected issues are shown in the standard *Sonargraph* tool window in the Intellij IDE. The tool window has the following tabs: Architecture Violations, Issues, Cycles, Tasks, and Refactorings.

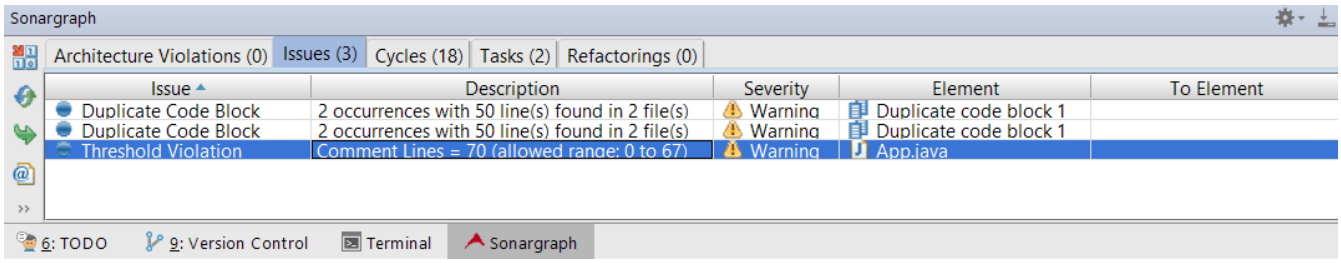


Figure 20.8. Sonargraph Tool Window

20.2.3. Toolbar

Sonargraph's tool window has a toolbar on the left hand side which has four buttons :

- Make Project : Triggers the source code compilation directly from *Sonargraph's* tool window.
- Synchronize : Reloads the information that is currently contained in *Sonargraph's* system files. If changes are detected, the user interface will be updated accordingly.
- Reset : Recreates graphical the components in *Sonargraph's* tool window and synchronizes the information contained in the system files.
- Send Feedback : This button will open *Sonargraph's* feedback dialog. Any information submitted from this dialog will be sent to support@hello2morrow.com
- Scroll to Source : When this toggle button is pushed, any click on an architecture violation, issue, task or compilation unit cycle group whose affected element is a source file will open the java source editor and go to the line containing the marker associated with the clicked issue.
- Toggle Markers : Shows/hides the different markers that the *Sonargraph* Intellij plugin will add for issues/tasks in the IDE source editor.

20.2.4. Getting Back In Sync with Manual Refresh

If you updated *Sonargraph* system files in parallel using the *Sonargraph* Architect application, you can use the "Synchronize" button in the toolbar to get these files in sync.

If you notice that some markers are not properly updated, or that the *Sonargraph* analysis has not picked up the latest changes, please compile your code to bring the *Sonargraph* model back in sync with the Intellij project. You can use the "Make Project" button in the toolbar.

TIP

If you notice any problem using the plugin, we are grateful to receive your feedback! The easiest way is to use the "Send Feedback" toolbar button.

TIP

If an exception happens in our plugin, you will get an error notification from Intellij and if you click on it, you will get the Intellij feedback dialog. Since this dialog only sends feedback to Intellij's bug tracking system, please click on the

"Add Details..." button to get the Sonargraph error feedback dialog, fill in the details and click on the "Ok" button. This way the information gets to us directly and we can address errors quickly.

20.2.5. Examining Changes

Similar to Sonargraph application, the IntelliJ plugin allows to track changes of issues with respect to a baseline. The Sonargraph menu allows the following interactions:

- New Baseline: Create and apply a new baseline, i.e. at the beginning of a feature development, to ensure that no new issues are introduced.
- Open Baseline: Open an existing baseline, i.e. an XML report generated at the end of the previous release.
- Activate System Baseline: Switch to the baseline that is configured in the software system. Obviously, this menu is only enabled if there is a baseline configured and it is currently not activated.
- Detach Baseline: Disconnect the current Sonargraph system from the baseline to see all existing issues.
- Export HTML Report: Create and open an HTML report focussed on the differences w.r.t the baseline.

Sonargraph issues are converted to IntelliJ markers. If a baseline is applied, all unmodified issues are assigned the severity "info". This sets them clearly apart from the added or changed issues which keep their original severity. The following screenshot shows the Sonargraph Tool Window and the possible menu interactions:

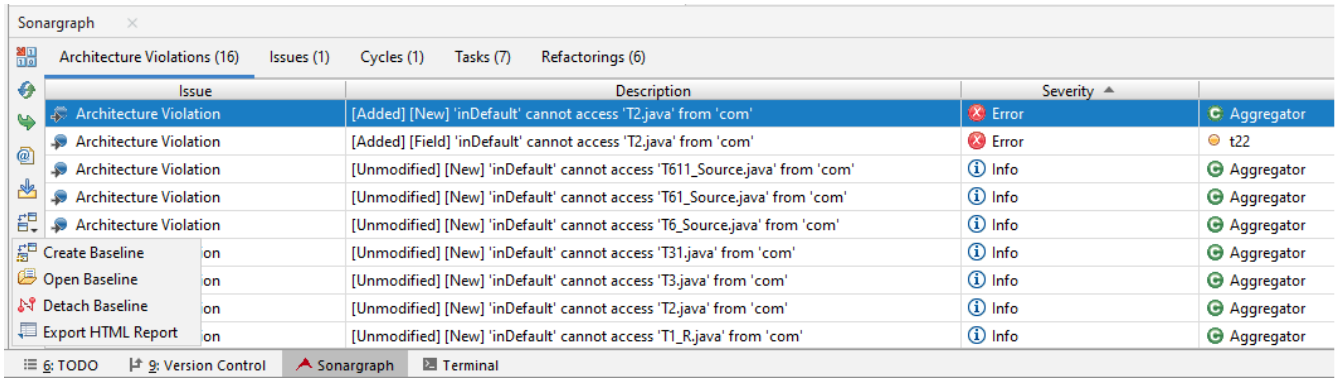


Figure 20.9. Sonargraph Issues in IntelliJ with Baseline Applied

There are no markers created for resolved issues. If you are interested which issues have been resolved, you need to create the HTML report.

Current Limitations

Not all functionality related to the system diff has been implemented yet in the IntelliJ plugin and the following list summarizes the current limitations, which will be addressed in future releases:

1. The "Sonargraph Cycle Groups" view does not support the system diff and always shows the complete list of cycle groups. To see diff information about cycle groups, you currently have to generate the HTML report.

Related topics:

- Chapter 14, *Examining Changes*

20.2.6. Execute Refactorings in IntelliJ

The list of Sonargraph refactorings definitions is shown in the "Refactorings" tab of the *Sonargraph* tool window. A refactoring can be executed via right-click.

NOTE

Refactorings defined in Sonargraph might affect a lot of resources. We recommend committing all pending changes to your version control system before executing the refactorings, so you have a safe fallback.

NOTE

Execute the refactorings in the order of their definition. Otherwise subsequent refactorings might no longer be applicable.

The Sonargraph plugin delegates the refactorings to the refactoring mechanism of IntelliJ. Sonargraph "Move+Rename" refactorings of compilation units cannot be converted into a single IntelliJ refactoring and therefore needs to be split.

The following steps are executed for each refactoring:

1. If the Sonargraph refactoring needs to be split, a confirmation dialog will inform you about the necessary actions.
2. The standard IntelliJ refactoring dialogs and views are shown that allow you to control the affected resources (e.g. change names in non-Java files) and preview the changes.

Related topics:

- Chapter 10, *Simulating Refactorings*
- Section 10.4, “Best Practices”

20.3. Collaboration between Sonargraph and IDE

As of version 11.1, Sonargraph offers a close integration with the Eclipse plugin to make it easier to fix issues right away when analyzing the code base in Sonargraph. And also vice-versa, making it easy to investigate dependencies using Sonargraph's advanced visualizations when coding in the IDE.

NOTE

The integration is currently only implemented for the Eclipse plugin!

Interactions

- **Connect / Disconnect:** If selected, the application listens to incoming selection requests.
- **Send Selection Request:** Information about the current selection is sent.
- **Reveal Selection Request:**

  Sonargraph -> IDE: The matching element is highlighted in the 'Package Explorer' or 'Project Explorer' in Eclipse and if the selection has been within a source file in Sonargraph, the editor is opened automatically in Eclipse and the matching line is selected.

  IDE -> Sonargraph: The matching element(s) are selected in the 'Navigation' view. From there the appropriate visualization can be opened via the context menu.

NOTE

The following preconditions must be fulfilled for the integration to work:

- The same Sonargraph system must be opened in Sonargraph and the IDE.
- The receiving application must be 'connected', i.e. must listen to incoming selection requests.

NOTE

You need to manually trigger a refresh (F5) in Sonargraph after changing code in the IDE.

Configuration

Default ports for listening to selection requests are 42420 (Sonargraph) and 42421 (IDE). This can be changed via a preference page in Sonargraph or via the menu "Sonargraph" → "Configure Remote Selection..." in Eclipse.

NOTE

The configuration is shared between the two applications, so that ports need to be configured only once. Just re-connect in the other application to activate the new configuration.

Sample Use Cases

Fixing a Detected Issue in the IDE

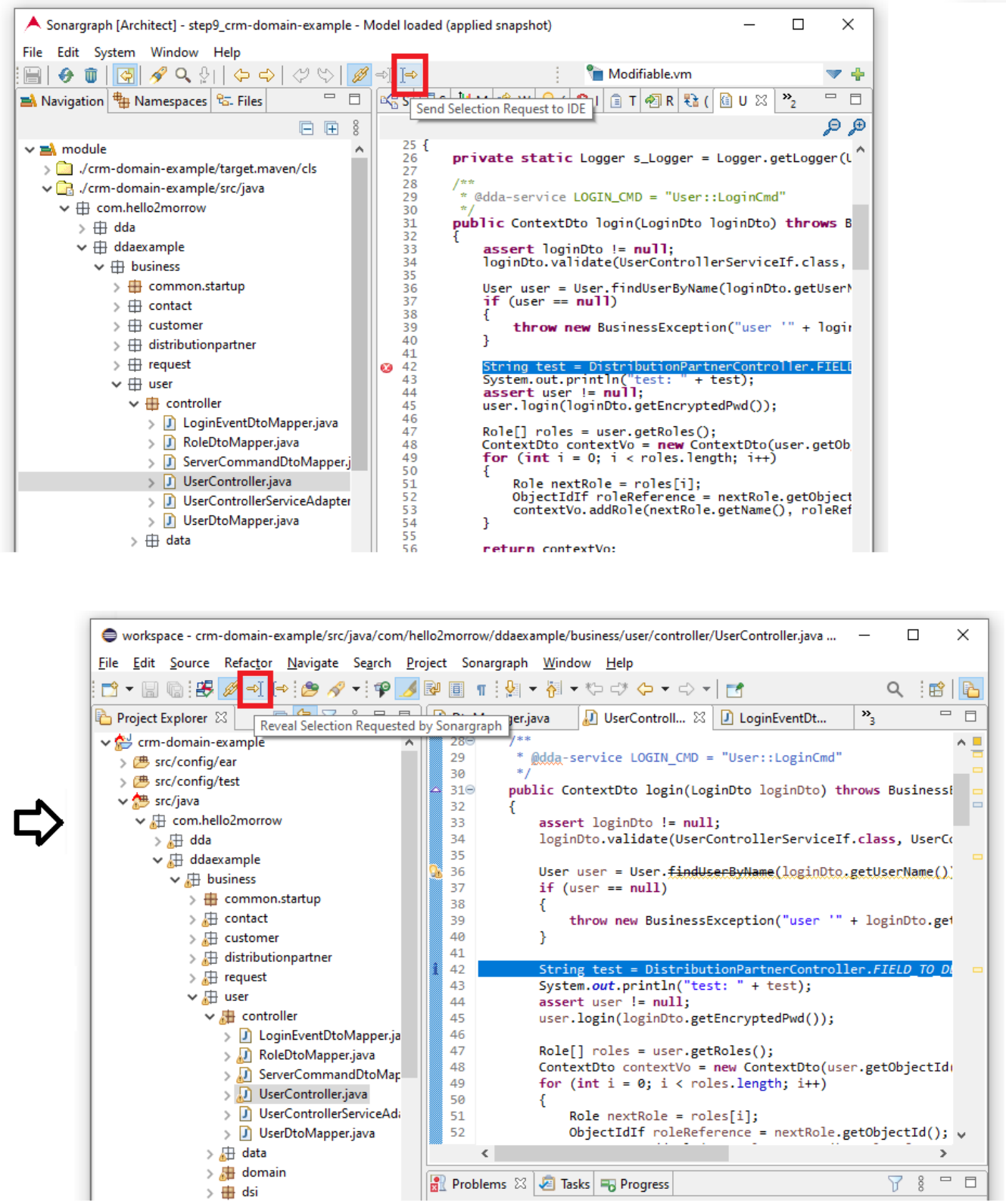


Figure 20.10. Fixing a Detected Issue in the IDE

Selecting Elements for Inspection in Sonargraph

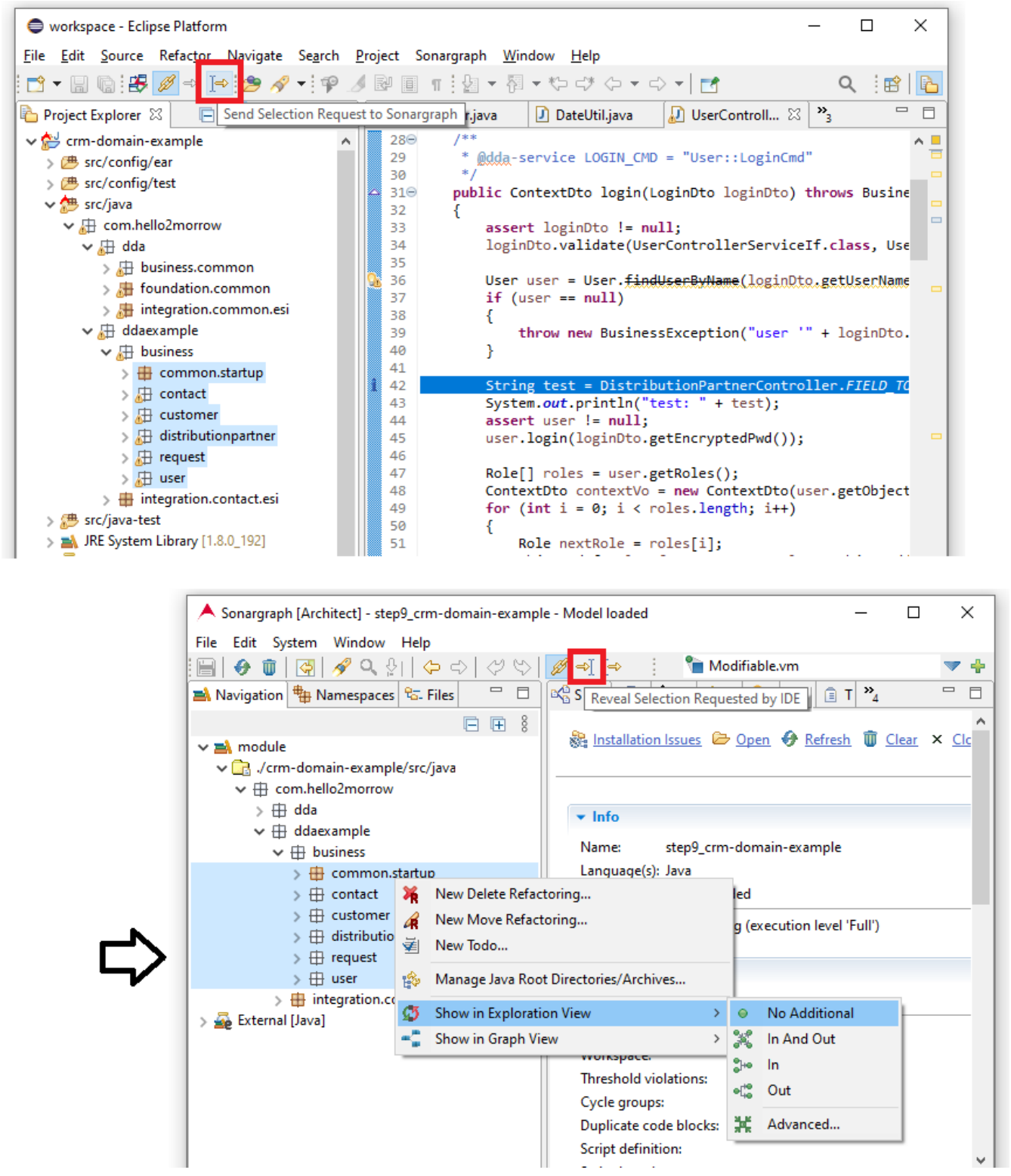


Figure 20.11. Selecting Elements for Inspection in Sonargraph

Chapter 21. Metric Definitions

This chapter contains definitions for the built-in metrics provided by Sonargraph.

21.1. Language Independent Metrics

Architecture Violation Density

Description: Number of architecture violations per 1000 lines of code. This metric is calculated for code that is fully analyzed plus code that is excluded by the 'Issue Filter'.

Categories: Architecture

Architecture Violation Density (Source Elements)

Description: Number of architecture violations per 1000 source elements. This metric is calculated for code that is fully analyzed plus code that is excluded by the 'Issue Filter'.

Categories: Architecture

Code Contained in Files Uncovered by Architecture (%)

Description: Percentage of lines of code contained in files not assigned to any architecture artifact. This metric is calculated for code that is fully analyzed plus code that is excluded by the 'Issue Filter'.

Categories: Architecture

Code Contained in Files with Violations (%)

Description: Percentage of lines of code contained in files with at least one violation. This metric is calculated for code that is fully analyzed plus code that is excluded by the 'Issue Filter'.

Categories: Architecture

Code Contained in Files with Violations or Deprecations (%)

Description: Percentage of lines of code contained in files with at least one violation or deprecation.This metric is calculated for code that is fully analyzed plus code that is excluded by the 'Issue Filter'.

Categories: Architecture

Deprecated parser dependencies

Description: Number of deprecated parser dependencies

Categories: Architecture

Ignored Deprecated Parser Dependencies

Description: Number of parser dependencies in ignored architecture deprecations

Categories: Architecture

Lines of Code in Files with Violations

Description: Lines of code contained in files with at least one violation. This metric is calculated for code that is fully analyzed plus code that is excluded by the 'Issue Filter'.

Categories: Architecture

Lines of Code in Files with Violations or Deprecations (%)

Description: Lines of code contained in files with at least one violation or deprecation.This metric is calculated for code that is fully analyzed plus code that is excluded by the 'Issue Filter'.

Categories: Architecture

Number of Artifacts

Description: Number of architecture artifacts in checked files

Categories: Architecture

Number of Components in Deprecated Artifacts

Description: Number of components that are assigned to deprecated artifact

Categories: Architecture

Number of Components with Violations

Description: Number of components that contain architecture violations

Categories: Architecture

Number of Empty Artifacts

Description: Number of architecture artifacts that are empty in checked files

Categories: Architecture

Number of Ignored Violations (Parser Dependencies)

Description: Number of parser dependencies in ignored architecture violations

Categories: Architecture

Number of Logical Elements in Deprecated Artifacts

Description: Number of logical programming elements that are assigned to deprecated artifact

Categories: Architecture

Number of Unassigned Logical Elements

Description: Number of internal logical elements that are not assigned to any artifact

Categories: Architecture

Number of Unassigned Physical Components

Description: Number of internal physical components that are not assigned to any artifact

Categories: Architecture

Number of Violations (Component Dependencies)

Description: Number of architecture-violating component dependencies

Categories: Architecture

Number of Violations (Parser Dependencies)

Description: Number of architecture-violating parser dependencies

Categories: Architecture

Average Block Nesting Depth

Description: Weighted average of nesting depth.

Categories: Code Analysis

Component Dependencies to Remove (Components)

Description: Number of component dependencies to remove to break up all component cycles.

Categories: Code Analysis, Cycle

Component Rank (Module)

Description: Component Rank is based on Google's page rank algorithm. The total component rank over all components in the selected group adds up to 100. The higher the rank, the more 'important' a component is in a system. Having many incoming dependencies or being referenced by other important components increases rank.

Categories: Code Analysis

Component Rank (System)

Description: Component Rank is based on Google's page rank algorithm. The total component rank over all components in the selected group adds up to 100. The higher the rank, the more 'important' a component is in a system. Having many incoming dependencies or being referenced by other important components increases rank.

Categories: Code Analysis

Issue Density

Description: Calculated as the number of unresolved issues (errors, warnings) * 1000, divided by source element count

Categories: Code Analysis

Max Block Nesting Depth

Description: Nesting depth is a good complexity indicator. Minimum value is zero, each nesting level adds 1.

Categories: Code Analysis

Number of Code Duplicates

Description: Number of duplicated code blocks.

Categories: Code Analysis

Number of Code Duplicates to be Fixed

Description: Number of duplicated code blocks with applied Fix task.

Categories: Code Analysis

Number of Duplicated Code Lines

Description: Number of duplicated lines in duplicated code blocks. The duplicated lines of each code block are calculated as the sum of involved occurrences excluding the largest, which is treated as the reference.

Categories: Code Analysis

Number of Ignored Code Duplicates

Description: Number of ignored duplicated code blocks.

Categories: Code Analysis

Parser Dependencies to Remove (Components)

Description: Number of code lines to change to break up all component cycles.

Categories: Code Analysis, Cycle

Redundant Code (%)

Description: Percentage of redundant code. This also represents the probability that any line is contained in a duplicate. This metric is calculated for fully analyzed code.

Categories: Code Analysis

Redundant Code [Ignored] (%)

Description: Percentage of ignored redundant code. This also represents the probability that any line is contained in an ignored duplicate. This metric is calculated for fully analyzed code.

Categories: Code Analysis

Redundant Code [To Be Fixed] (%)

Description: Percentage of redundant code with an assigned "Fix" task. This also represents the probability that any line is contained in a to-be-fixed duplicate. This metric is calculated for fully analyzed code.

Categories: Code Analysis

Structural Debt Index (Components)

Description: Cumulative structural debt index of component cycles.

Categories: Code Analysis

Biggest Component Cycle Group

Description: Number of components in biggest cycle.

Categories: Cycle

Critically Entangled Lines of Code

Description: Lines of code of source files involved any type of critical cycle (marked as error).

Categories: Cycle

Critically Entangled Lines of Code (%)

Description: Percentage of fully analyzed code contained in source files involved any type of critical cycle (marked as error). This also represents the probability that any line is involved in critically entangled code.

Categories: Cycle

Critically Entangled Lines of Code [Ignored]

Description: Lines of code of source files involved in any type of ignored critical cycle (marked as error).

Categories: Cycle

Critically Entangled Lines of Code [Ignored] (%)

Description: Percentage of fully analyzed code contained in source files involved in any type of ignored critical cycle (marked as error). This also represents the probability that any line is involved in ignored critically entangled code.

Categories: Cycle

Critically Entangled Lines of Code [To Be Fixed]

Description: Lines of code of source files involved any type of to be fixed critical cycle (marked as error).

Categories: Cycle

Critically Entangled Lines of Code [To Be Fixed] (%)

Description: Percentage of fully analyzed code contained in source files involved any type of to be fixed critical cycle (marked as error). This also represents the probability that any line is involved in to be fixed critically entangled code.

Categories: Cycle

Cyclicity (Components)

Description: Cumulated cyclicity of component cycles.

Categories: Cycle

Entangled Lines of Code

Description: Lines of code of source files involved any type of cycle.

Categories: Cycle

Entangled Lines of Code (%)

Description: Percentage of fully analyzed code contained in source files involved in any type of cycle. This also represents the probability that any line is involved in entangled code.

Categories: Cycle

Entangled Lines of Code [Ignored]

Description: Lines of code of source files involved in any type of ignored cycle.

Categories: Cycle

Entangled Lines of Code [Ignored] (%)

Description: Percentage of fully analyzed code contained in source files involved in any type of ignored cycle. This also represents the probability that any line is involved in ignored entangled code.

Categories: Cycle

Entangled Lines of Code [To Be Fixed]

Description: Lines of code of source files involved any type of to be fixed cycle.

Categories: Cycle

Entangled Lines of Code [To Be Fixed] (%)

Description: Percentage of fully analyzed code contained in source files involved any type of to be fixed cycle. This also represents the probability that any line is involved in to be fixed entangled code.

Categories: Cycle

Maximum Lines of Code Involved in a Cycle

Description: Biggest cycle group with respect to the lines of code of involved source files.

Categories: Cycle

Number of Component Cycle Groups

Description: Number of all component cycle groups, warnings and errors.

Categories: Cycle

Number of Critical Component Cycle Groups

Description: Number of component cycle groups marked as errors.

Categories: Cycle

Number of Cyclic Components

Description: Number of cyclic components.

Categories: Cycle

Number of Cyclic Modules

Description: Number of cyclic modules.

Categories: Cycle

Number of Ignored Cyclic Components

Description: Number of ignored cyclic components.

Categories: Cycle

Relative Cyclicity (Components)

Description: Relative component cyclicity in percent.

Categories: Cycle

Relative Entanglement (%)

Description: Computed as the sum of relative cyclicities on component and namespace/directory levels, with each level contributing 50%. If the system contains several languages, the namespace/directory values per language are weighted against the lines of code contributed by the language. High values are an indicator for very large cycle groups.

Categories: Cycle

ACD

Description: Average component dependency according to John Lakos. Average number of components a component depends on directly and indirectly. This metric can be used to characterize the overall average coupling of internal components.

Categories: Cohesion/Coupling, John Lakos

CCD

Description: Cumulative component dependency according to John Lakos. Cumulated depends upon values.

Categories: Cohesion/Coupling, John Lakos

Depends Upon (Module)

Description: Depends upon module level according to DependsOn by John Lakos. Total number of components that a component directly and indirectly depends upon in containing module.

Categories: Cohesion/Coupling, John Lakos

Depends Upon (System)

Description: Depends upon system level according to DependsOn by John Lakos. Total number of components that a component directly and indirectly depends upon in system.

Categories: Cohesion/Coupling, John Lakos

Fan In Maintainability Level (Module)

Description: Percentage of higher-level components in the same module that depend directly or indirectly on this component.

Categories: Cohesion/Coupling

Fan In Visibility (Module)

Description: Percentage of components in the same module that depend directly or indirectly on this component.

Categories: Cohesion/Coupling, MacCormack, Rusnak, Baldwin

Fan In Visibility (System)

Description: Percentage of internal components in the system that depend directly or indirectly on this component.

Categories: Cohesion/Coupling, MacCormack, Rusnak, Baldwin

Fan Out Visibility (Module)

Description: Percentage of components in the same module that this component depends upon.

Categories: Cohesion/Coupling, MacCormack, Rusnak, Baldwin

Fan Out Visibility (System)

Description: Percentage of internal components in the system that this component depends upon.

Categories: Cohesion/Coupling, MacCormack, Rusnak, Baldwin

Highest ACD

Description: Highest module ACD.

Categories: Cohesion/Coupling, John Lakos

LCOM4

Description: Determines the number of components in a class. A component is composed of fields, methods and types defined top level including all their nested programming elements. Constructors, destructors, empty, abstract and overridden methods of classes are not included in the calculation. The metric represents the unrelated portions of code in a class. A value of 1 indicates the highest cohesion possible - which is normally desirable. High values might indicate that a class is a candidate for a refactoring. Consider that utility classes by nature have high LCOM4 values.

Categories: Cohesion/Coupling

Logical Cohesion (Module)

Description: Number of dependencies 'to' and 'from' other top-level logical programming elements in the same namespace on module level.

Categories: Cohesion/Coupling

Logical Cohesion (System)

Description: Number of dependencies 'to' and 'from' other top-level logical programming elements in the same namespace on system level.

Categories: Cohesion/Coupling

Logical Coupling (Module)

Description: Number of dependencies 'to' and 'from' other top-level logical programming elements in other namespaces on module level.

Categories: Cohesion/Coupling

Logical Coupling (System)

Description: Number of dependencies 'to' and 'from' other top-level logical programming elements in other namespaces on system level.

Categories: Cohesion/Coupling

Maintainability Level

Description: This metric estimates maintainability as a percentage. 100% is the best possible value. To do that it looks at the dependency structure between components (source files in most languages). Cyclic dependencies and low level classes with a lot of incoming dependencies have a negative influence on the metric. Keeping good vertical boundaries and not having too many layers will have a positive influence. It is also recommended to have as many components as possible that are independent, i.e. have no incoming dependencies and therefore can be changed without influencing the rest of the system. In Java and C# the metric also considers the value of the relative cyclicity metric for packages/ namespaces. If you have large cycle groups they will have a negative influence on the metric value.

Categories: Cohesion/Coupling

NCCD

Description: Normalized cumulative component dependency according to John Lakos. The ratio between the cumulative component dependency and the cumulative component dependency of a balanced binary tree of the same size. A value greater than 1 indicates a more vertical design. A value less than 1 indicates a more horizontal design.

Categories: Cohesion/Coupling, John Lakos

Physical Cohesion

Description: Number of dependencies 'to' and 'from' other components in the same module.

Categories: Cohesion/Coupling

Physical Coupling

Description: Number of dependencies 'to' and 'from' other components in other modules.

Categories: Cohesion/Coupling

Propagation Cost

Description: Propagation cost metric according to MacCormack, Rusnak and Baldwin. It describes the proportion of software files that are directly or indirectly linked to each other.

Categories: Cohesion/Coupling, MacCormack, Rusnak, Baldwin

Used From (Module)

Description: Number of all depending elements (direct and indirect) + 1 (including self) in containing module.

Categories: Cohesion/Coupling, John Lakos

Used From (System)

Description: Number of all depending elements (direct and indirect) + 1 (including self) in system.

Categories: Cohesion/Coupling, John Lakos

Code Comment Lines

Description: Counts all comment lines excluding header comments and blank comment lines. This includes code of fully analyzed and issue ignoring code.

Categories: Size

Comment Lines

Description: Counts all comment lines excluding blank comment lines. This includes fully analyzed and issue ignoring code.

Categories: Size

Lines of Code

Description: Lines of code excluding blank and comment lines. This includes fully analyzed and issue ignoring code.

Categories: Size

Lines of Fully Analyzed Code

Description: Lines of fully analyzed code excluding blank and comment lines.

Categories: Size

Lines of Fully Analyzed Code in Large Files

Description: Lines of fully analyzed code excluding blank and comment lines in files violating the threshold (default 1000).

Categories: Size

Lines of Fully Analyzed Code in Large Files (%)

Description: Percent of lines of fully analyzed code excluding blank and comment lines in files violating the threshold (default 1000).

Categories: Size

Lines of Fully Analyzed Code in Large Files [Ignored]

Description: Lines of fully analyzed code excluding blank and comment lines in ignored files violating the threshold (default 1000).

Categories: Size

Lines of Fully Analyzed Code in Large Files [Ignored] (%)

Description: Percent of lines of fully analyzed code excluding blank and comment lines in ignored files violating the threshold (default 1000).

Categories: Size

Lines of Fully Analyzed Code in Large Files [To Be Fixed]

Description: Lines of fully analyzed code excluding blank and comment lines in to be fixed files violating the threshold (default 1000).

Categories: Size

Lines of Fully Analyzed Code in Large Files [To Be Fixed] (%)

Description: Percent of lines of fully analyzed code excluding blank and comment lines in to be fixed files violating the threshold (default 1000).

Categories: Size

Lines of Issue-Ignoring Code

Description: Lines of code excluding blank and comment lines for which only architecture violations and parsing problems are reported.

Categories: Size

Number of Components (Full Analysis)

Description: Number of fully analyzed components.

Categories: Size

Number of Components (Ignoring Issues)

Description: Number of components ignoring issues.

Categories: Size

Number of Components/Sources

Description: Number of components or source files

Categories: Size

Number of Excluded Source Files

Description: Number of source files excluded via 'File Filter'. These files are completely excluded from the analysis and do not contribute to any metric.

Categories: Size

Number of Logical Types (Module)

Description: Number of logical types (classes, enums or similar) in container on module level.

Categories: Size

Number of Logical Types (System)

Description: Number of logical types (classes, enums or similar) in container on system level.

Categories: Size

Number of Methods

Description: Number of member functions.

Categories: Size

Number of Modules

Description: Number of modules.

Categories: Size

Number of Parameters

Description: Number of parameters.

Categories: Size

Number of Source Files

Description: Number of source files in fully analyzed and issue ignoring code.

Categories: Size

Number of Source Files (Excluded)

Description: Number of source files in test code (excluded via 'Production Code Filter').

Categories: Size

Number of Source Files (Full Analysis)

Description: Number of source files that are fully analyzed, i.e. not excluded by any workspace filter.

Categories: Size

Number of Source Files (Ignoring Issue)

Description: Number of source files excluded via 'Issue Filter' that no issues (except parser issues and architecture violations) are generated for.

Categories: Size

Number of Statements

Description: Counts all statements. This includes statements of fully analyzed and issue ignoring code.

Categories: Size

Number of Statements in Fully Analyzed Code

Description: Counts all statements in fully analyzed code.

Categories: Size

Number of Types

Description: Number of types (classes, enums or similar) in container.

Categories: Size

Source Element Count

Description: Number of programming elements (i.e. types, fields, methods, functions, ...) plus number of statements. This includes elements of fully analyzed and issue ignoring code.

Categories: Size

Total Lines

Description: Counts all lines including empty and comment lines of source files. This includes files of fully analyzed and issue ignoring code.

Categories: Size

Relational Cohesion (Module)

Description: Relation cohesion according to Craig Larman (adapted). Number of internal namespace dependencies divided by the number of top-level logical programming elements in the same namespace on module level. Higher numbers suggest more cohesion.

Categories: Craig Larman, Cohesion/Coupling

Relational Cohesion (System)

Description: Relation cohesion according to Craig Larman (adapted). Number of internal namespace dependencies divided by the number of top-level logical programming elements in the same namespace on system level. Higher numbers suggest more cohesion.

Categories: Craig Larman, Cohesion/Coupling

Abstractness (Module)

Description: Abstractness according to Robert C. Martin based on module level dependencies. Total number of abstract types divided by the total number of concrete types. The metric has a range of [0,1]. 0 means that the container contains no abstract types. 1 means that the container contains nothing but abstract types.

Categories: Robert C. Martin

Abstractness (System)

Description: Abstractness according to Robert C. Martin based on system level dependencies. Total number of abstract types divided by the total number of concrete types. The metric has a range of [0,1]. 0 means that the container contains no abstract types. 1 means that the container contains nothing but abstract types.

Categories: Robert C. Martin

Distance (Module)

Description: Distance according to Robert C. Martin based on module level dependencies. Abstractness + Instability - 1. The metric has a range of [-1,1]. This is a variation of the original metric definition. A negative sign means 'in the zone of pain' and a positive sign means 'in the zone of uselessness'. A 'good' value should be around 0.

Categories: Robert C. Martin

Distance (System)

Description: Distance according to Robert C. Martin based on system level dependencies. Abstractness + Instability - 1. The metric has a range of [-1,1]. This is a variation of the original metric definition. A negative sign means 'in the zone of pain' and a positive sign means 'in the zone of uselessness'. A 'good' value should be around 0.

Categories: Robert C. Martin

Instability (Module)

Description: Instability according to Robert C. Martin based on module level dependencies. The metric has a range of [0,1]. If there are no outgoing dependencies, then the Instability will be 0 and the measured element is stable. If there are no incoming dependencies, then the Instability will be 1 and the measured element is instable. Stable means that the element is not so easy to be changed. Instable means that it is easier to be changed.

Categories: Robert C. Martin

Instability (System)

Description: Instability according to Robert C. Martin based on system level dependencies. The metric has a range of [0,1]. If there are no outgoing dependencies, then I will be 0 and the measured element is stable. If there are no incoming dependencies, then I will be 1 and the measured element is instable. Stable means that the element is not so easy to be changed. Instable means that it is easier to be changed.

Categories: Robert C. Martin

Number of Incoming Dependencies (Module)

Description: Number of incoming dependencies on module level.

Categories: Robert C. Martin

Number of Incoming Dependencies (System)

Description: Number of incoming dependencies on system level.

Categories: Robert C. Martin

Number of Outgoing Dependencies (Module)

Description: Number of outgoing dependencies on module level.

Categories: Robert C. Martin

Number of Outgoing Dependencies (System)

Description: Number of outgoing dependencies on system level.

Categories: Robert C. Martin

Average Complexity

Description: Weighted average modified extended cyclomatic complexity for fully analyzed code

Categories: Thomas J. McCabe

Average Complexity (Module)

Description: Weighted average modified extended cyclomatic complexity for fully analyzed code on module level

Categories: Thomas J. McCabe

Average Complexity (System)

Description: Weighted average modified extended cyclomatic complexity for fully analyzed code on system level

Categories: Thomas J. McCabe

Cyclomatic Complexity

Description: Cyclomatic complexity according to Thomas J. McCabe. Number of decision points in a method plus one for the method entry.

Categories: Thomas J. McCabe

Extended Cyclomatic Complexity

Description: As cyclomatic complexity adding the number of logical '&&' and '||' operations.

Categories: Thomas J. McCabe

Modified Cyclomatic Complexity

Description: As cyclomatic complexity but switch statements only add 1 independent from the number of cases.

Categories: Thomas J. McCabe

Modified Extended Cyclomatic Complexity

Description: As cyclomatic complexity but switch statements only add 1 independent from the number of cases and adding the number of logical '&&' and '||' operations.

Categories: Thomas J. McCabe

Code Churn (2y)

Description: Number of lines added or removed in the last 2 years

Categories: Change History

Code Churn (30d)

Description: Number of lines added or removed in the last 30 days

Categories: Change History

Code Churn (365d)

Description: Number of lines added or removed in the last 365 days

Categories: Change History

Code Churn (5y)

Description: Number of lines added or removed in the last 5 years

Categories: Change History

Code Churn (90d)

Description: Number of lines added or removed in the last 90 days

Categories: Change History

Code Churn Rate (2y)

Description: Percentage of lines added or removed in the last 2 years based on total lines

Categories: Change History

Code Churn Rate (30d)

Description: Percentage of lines added or removed in the last 30 days based on total lines

Categories: Change History

Code Churn Rate (365d)

Description: Percentage of lines added or removed in the last 365 days based on total lines

Categories: Change History

Code Churn Rate (5y)

Description: Percentage of lines added or removed in the last 5 years based on total lines

Categories: Change History

Code Churn Rate (90d)

Description: Percentage of lines added or removed in the last 90 days based on total lines

Categories: Change History

Days since last commit

Description: Days since this file was last changed (9999 means no changes in the last 5 years)

Categories: Change History

File Changes (2y)

Description: Number of committed file changes in the last 2 years

Categories: Change History

File Changes (30d)

Description: Number of committed file changes in the last 30 days

Categories: Change History

File Changes (365d)

Description: Number of committed file changes in the last 365 days

Categories: Change History

File Changes (5y)

Description: Number of committed file changes in the last 5 years

Categories: Change History

File Changes (90d)

Description: Number of committed file changes in the last 90 days

Categories: Change History

Number of Authors (30d)

Description: Number of developers who have worked on this item in the last 30 days

Categories: Change History

Number of authors (2y)

Description: Number of developers who have worked on this item in the last 2 years

Categories: Change History

Number of authors (365d)

Description: Number of developers who have worked on this item in the last year

Categories: Change History

Number of authors (5y)

Description: Number of developers who have worked on this item in the last 5 years

Categories: Change History

Number of authors (90d)

Description: Number of developers who have worked on this item in the last 90 days

Categories: Change History

Number of Statements in Complex Methods

Description: Counts all statements in fully analyzed code of too complex methods, i.e. that violate the thresholds for max nesting depth (default 4) or for extended modified cyclomatic complexity (default 15).

Categories: Complexity

Number of Statements in Complex Methods (%)

Description: Percentage of statements in fully analyzed code in too complex methods, i.e. that violate the thresholds for max nesting depth (default 4) or for extended modified cyclomatic complexity (default 15).

Categories: Complexity

Number of Statements in Complex Methods [Ignored]

Description: Counts all statements in fully analyzed code of ignored too complex methods, i.e. that violate the thresholds for max nesting depth (default 4) or for extended modified cyclomatic complexity (default 15).

Categories: Complexity

Number of Statements in Complex Methods [Ignored] (%)

Description: Percentage of statements in fully analyzed code in ignored too complex methods, i.e. that violate the thresholds for max nesting depth (default 4) or for extended modified cyclomatic complexity (default 15).

Categories: Complexity

Number of Statements in Complex Methods [To Be Fixed]

Description: Counts all statements in fully analyzed code of to be fixed too complex methods, i.e. that violate the thresholds for max nesting depth (default 4) or for extended modified cyclomatic complexity (default 15).

Categories: Complexity

Number of Statements in Complex Methods [To Be Fixed] (%)

Description: Percentage of statements in fully analyzed code in to be fixed too complex methods, i.e. that violate the thresholds for max nesting depth (default 4) or for extended modified cyclomatic complexity (default 15).

Categories: Complexity

21.2. Java Metrics

Average Java Class Member Visibility (%) (Module)

Description: Average of Java class member visibility in a Java package

Categories: Code Analysis

Average Java Public Visibility (%)

Description: Average of Java public visibility for all Java packages in a Java module

Categories: Code Analysis

Component Dependencies to Remove (Java Packages)

Description: Number of component dependencies to remove to break up all Java package cycle groups.

Categories: Code Analysis, Dependency

Java Member Visibility (%)

Description: Percentage of non-private Java members in a class

Categories: Code Analysis

Java Public Visibility (%) (Module)

Description: Percentage of public Java types in a Java package

Categories: Code Analysis

Parser Dependencies to Remove (Java Packages)

Description: Number of code lines to change to break up all Java package cycle groups).

Categories: Code Analysis, Dependency

Structural Debt Index (Java Packages)

Description: Cumulative structural debt index of all Java package cycle groups.

Categories: Code Analysis

Biggest Java Package Cycle Group

Description: Biggest Java package cycle group.

Categories: Cycle

Cyclicity (Java Packages)

Description: Cumulated cyclicity of Java package cycle groups.

Categories: Cycle

Number of Critical Java Package Cycle Groups

Description: Number of Java package cycle groups marked as errors.

Categories: Cycle

Number of Cyclic Java Packages

Description: Number of cyclic Java packages.

Categories: Cycle

Number of Ignored Cyclic Java Packages

Description: Number of ignored cyclic Java packages.

Categories: Cycle

Number of all Java Package Cycle Groups

Description: Number of all Java package cycle groups, errors and warnings

Categories: Cycle

Relative Cyclicity (Java Packages)

Description: Relative Java package cyclicity in percent.

Categories: Cycle

Byte Code Instructions

Description: Number of Java byte code instructions.

Categories: Size

Number of Java Packages

Description: Number of Java packages containing types in fully analyzed and issue ignoring code.

Categories: Size

Number of Java Packages (Full Analysis)

Description: Number of Java packages containing fully analyzed types.

Categories: Size

21.3. C# Metrics

Component Dependencies to Remove (C# Directories)

Description: Number of component dependencies to remove to break up all C# directory cycle groups.

Categories: Code Analysis, Cycle

Component Dependencies to Remove (C# Namespaces)

Description: Number of component dependencies to remove to break up all C# namespace cycle groups.

Categories: Code Analysis, Dependency

Parser Dependencies to Remove (C# Directories)

Description: Number of code lines to change to break up all C# directory cycle groups.

Categories: Code Analysis, Cycle

Parser Dependencies to Remove (C# Namespaces)

Description: Number of code lines to change to break up all C# namespace cycle groups.

Categories: Code Analysis, Dependency

Structural Debt Index (C# Directories)

Description: Cumulative structural debt index of all C# directory cycle groups.

Categories: Code Analysis

Structural Debt Index (C# Namespaces)

Description: Cumulative structural debt index of all C# namespace cycle groups.

Categories: Code Analysis

Biggest C# Directory Cycle Group

Description: Biggest C# directory cycle group.

Categories: Cycle

Biggest C# Namespace Cycle Group

Description: Biggest C# namespace cycle group.

Categories: Cycle

Cyclicity (C# Directories)

Description: Cumulated cyclicity of C# directory cycle groups.

Categories: Cycle

Cyclicity (C# Namespaces)

Description: Cumulated cyclicity of C# namespace cycle groups.

Categories: Cycle

Number of Critical C# Directory Cycle Groups

Description: Number of C# directory cycle groups marked as errors.

Categories: Cycle

Number of Critical C# Namespace Cycle Groups

Description: Number of C# namespace cycle groups marked as errors.

Categories: Cycle

Number of Cyclic C# Directories

Description: Number of cyclic C# directories.

Categories: Cycle

Number of Cyclic C# Namespaces

Description: Number of cyclic C# namespaces.

Categories: Cycle

Number of Ignored Cyclic C# Directories

Description: Number of ignored cyclic C# directories.

Categories: Cycle

Number of Ignored Cyclic C# Namespaces

Description: Number of ignored cyclic C# namespaces.

Categories: Cycle

Number of all C# Directory Cycle Groups

Description: Number of all C# directory cycle groups, errors and warnings.

Categories: Cycle

Number of all C# Namespace Cycle Groups

Description: Number of C# namespace cycle groups, errors and warnings.

Categories: Cycle

Relative Cyclicity (C# Directories)

Description: Relative C# directory cyclicity in percent.

Categories: Cycle

Relative Cyclicity (C# Namespaces)

Description: Relative C# namespace cyclicity in percent.

Categories: Cycle

Number of C# Directories

Description: Number of C# directories containing components in fully analyzed and issue ignoring code.

Categories: Size

Number of C# Directories (Full Analysis)

Description: Number of C# directories containing fully analyzed components.

Categories: Size

Number of C# Namespaces

Description: Number of C# namespaces containing types in fully analyzed and issue ignoring code.

Categories: Size

Number of C# Namespaces (Full Analysis)

Description: Number of C# namespaces containing fully analyzed types.

Categories: Size

21.4. C,C++ Metrics

Component Dependencies to Remove (C++ Namespaces)

Description: Number of component dependencies to remove to break up all C++ namespace cycle groups.

Categories: Code Analysis, Cycle

Component Dependencies to Remove (C,C++ Directories)

Description: Number of component dependencies to remove to break up all C,C++ directory cycle groups.

Categories: Code Analysis, Cycle

Parser Dependencies to Remove (C++ Namespaces)

Description: Number of code lines to change to break up all C++ namespace cycle groups.

Categories: Code Analysis, Cycle

Parser Dependencies to Remove (C,C++ Directories)

Description: Number of code lines to change to break up all C,C++ directory cycle groups.

Categories: Code Analysis, Cycle

Structural Debt Index (C++ Namespaces)

Description: Cumulative structural debt index of all C++ namespace cycle groups.

Categories: Code Analysis

Structural Debt Index (C,C++ Directories)

Description: Cumulative structural debt index of all C,C++ directory cycle groups.

Categories: Code Analysis

Biggest C++ Namespace Cycle Group

Description: Biggest C++ namespace cycle group

Categories: Cycle

Biggest C,C++ Directory Cycle Group

Description: Biggest C,C++ directory cycle group.

Categories: Cycle

Cyclicity (C++ Namespaces)

Description: Cumulated cyclicity of C++ namespace cycle groups.

Categories: Cycle

Cyclicity (C,C++ Directories)

Description: Cumulated cyclicity of C,C++ directory cycle groups.

Categories: Cycle

Number of Critical C++ Namespace Cycle Groups

Description: Number of C++ namespace cycle groups marked as errors.

Categories: Cycle

Number of Critical C,C++ Directory Cycle Groups

Description: Number of C,C++ directory cycle groups marked as errors.

Categories: Cycle

Number of Cyclic C++ Namespaces

Description: Number of cyclic C++ namespaces.

Categories: Cycle

Number of Cyclic C,C++ Directories

Description: Number of cyclic C,C++ directories.

Categories: Cycle

Number of Ignored Cyclic C++ Namespaces

Description: Number of ignored cyclic C++ namespaces.

Categories: Cycle

Number of Ignored Cyclic C,C++ Directories

Description: Number of ignored cyclic C,C++ directories.

Categories: Cycle

Number of all C++ Namespace Cycle Groups

Description: Number of all C++ namespace cycle groups, errors and warnings.

Categories: Cycle

Number of all C,C++ Directory Cycle Groups

Description: Number of all C,C++ directory cycle groups, errors and warnings.

Categories: Cycle

Relative Cyclicity (C++ Namespaces)

Description: Relative C++ namespace cyclicity in percent.

Categories: Cycle

Relative Cyclicity (C,C++ Directories)

Description: Relative C,C++ directory cyclicity in percent.

Categories: Cycle

Number of C++ Namespaces

Description: Number of C++ namespaces containing types in fully analyzed and issue ignoring code.

Categories: Size

Number of C++ Namespaces (Full Analysis)

Description: Number of C++ namespaces containing fully analyzed types.

Categories: Size

Number of C,C++ Directories

Description: Number of C,C++ directories containing components in fully analyzed and issue ignoring code.

Categories: Size

Number of C,C++ Directories (Full Analysis)

Description: Number of C,C++ directories containing fully analyzed components.

Categories: Size

21.5. Python Metrics

Component Dependencies to Remove (Python Packages)

Description: Number of component dependencies to remove to break up all Python package cycle groups.

Categories: Code Analysis, Dependency

Parser Dependencies to Remove (Python Packages)

Description: Number of code lines to change to break up all Python package cycle groups.

Categories: Code Analysis, Dependency

Structural Debt Index (Python Packages)

Description: Cumulative structural debt index of all Python package cycle groups.

Categories: Code Analysis

Biggest Python Package Cycle Group

Description: Biggest Python package cycle group.

Categories: Cycle

Cyclicity (Python Packages)

Description: Cumulated cyclicity of Python package cycle groups.

Categories: Cycle

Number of Critical Python Package Cycle Groups

Description: Number of Python package cycle groups marked as errors.

Categories: Cycle

Number of Cyclic Python Packages

Description: Number of cyclic Python packages.

Categories: Cycle

Number of Ignored Cyclic Python Packages

Description: Number of ignored cyclic Python packages.

Categories: Cycle

Number of all Python Package Cycle Groups

Description: Number of all Python package cycle groups, errors and warnmings.

Categories: Cycle

Relative Cyclicity (Python Packages)

Description: Relative Python package cyclicity in percent.

Categories: Cycle

Number of Python Packages

Description: Number of Python packages containing types in fully analyzed and issue ignoring code.

Categories: Size

Number of Python Packages (Full Analysis)

Description: Number of Python packages containing fully analyzed types.

Categories: Size

Chapter 22. How to Resolve Issues

This section summarizes issues and how they can be resolved.

22.1. Language Independent Issues

Root path does not exist

Indicates that the path supplied for a Root Directory Path cannot be found on disk. A valid path on disk must be supplied.

Duplicate Code

See Section 8.13, “Detecting Duplicate Code” for more information about how to investigate code duplicates and the configuration of the analyzer.

22.2. Java Specific Issues

Class file is out-of-date

Indicates that the class file is older than the corresponding source file. Source needs to be re-compiled.

22.3. C# Specific Issues

C# Parsing Errors

Parsing errors in the C# parser log are usually caused by referenced assemblies that cannot be found locally. Try to build the solution in your favored IDE before analyzing it with Sonargraph.

Project File (.csproj) Processing Failed

Indicates a fatal error during the processing of a C# project file. Depending on the configuration, Sonargraph can use MSBuild to extract information about macro definitions, target frameworks, project references, assembly references, source files, etc. to enable a precise analysis. As Visual Studio and MSBuild are constantly evolving, our integration with MSBuild might be incomplete. You can help us improve the integration by sending us the log file via "Help" → "Send Feedback..." (don't forget to tick "Attach log file") or by sending an email to <support@hello2morrow.com>.

22.4. C/C++ Specific Issues

C/C++ Parsing Errors

Indicates that the EDG parser failed to process a source file. Check that the code compiles in your standard IDE or in your build environment.

Check that the compiler definitions are correct. See Section 4.7, “C/C++ Compiler Definitions” for details.

Report C/C++ Parsing Problems

If you need further support, report the parsing problem to us via the menu "Help" → "Report C/C++ Parsing Problem..." . All files relevant for the problem analysis will be zipped and sent to us for further inspection. Please consider adding some context info.

NOTE

The diagnostic files contain expanded source code. If your company guidelines do not allow to share this information with us, deselect the corresponding checkboxes.

Chapter 23. FAQ

This section summarizes common problems and their solutions.

23.1. Out Of Memory Exceptions

In case of OutOfMemoryExceptions increase the memory made available to *Sonargraph* by opening the file Sonargraph.ini and increase the value for the -Xmx parameter.

23.2. Groovy Template

The configuration of *Sonargraph* is very flexible due to usage of Groovy Templates. Per default, all environment variables are available. The following script illustrates the usage of the variable INCLUDE:

```
<%
def elements = INCLUDE.split(";");
for (element in elements)
{
   println "--sys_include=" + element;
}
%>
```

23.3. MSBuild Error (MSB4019) during Analysis of Visual Studio C# Project

Sonargraph uses MSBuild to retrieve configuration info for C# projects. This specific error can be resolved by removing the following lines from the .csproj file as described on *stackoverflow*:

```
<PropertyGroup>
   <VisualStudioVersion Condition="'$(VisualStudioVersion)' == ''">10.0</VisualStudioVersion>
   <VSToolsPath Condition="'$(VSToolsPath)' == ''">
      $(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)</VSToolsPath>
 </PropertyGroup>
```

Chapter 24. References

Articles about various software quality topics can be found at http://blog.hello2morrow.com/

Our whitepapers and presentations are available at https://www.hello2morrow.com/products/whitepapers

The following list contains books that influenced us a lot prior and during the development of Sonargraph.

[ACM] McCabe, T. J. "A Complexity Measure." IEEE Trans. Software Eng. SE-2, 4, 308-320, Dec. 1976

[ASD] Agile Software Development, Robert C. Martin, Prentice Hall 2003

[AUP] Applying UML And Patterns, Craig Larman, Prentice Hall 2002

[EOT] Erfolgsschlüssel Objekttechnologie, Betrand Meyer, Hanser 1995

[JLS] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, "The Java Language Specification", Addison-Wesley 2005

[LSD] Large-Scale C++ Software Design, John Lakos, Addison-Wesley 1996

[PAP] Robert C. Martin, "Design Principles and Patterns", Objectmentor 2000

[PPR] Jones T.C., "Programming Productivity", New York, McGraw-Hill 1986

[SEE] Boehm, B. W., "Software Engineering Economics", Englewood Cliffs, N. J.: Prentice-Hall 1981

[SOM] Everald E. Mills, "Software Metrics", SEI Curriculum Module SEI-CM-12-1.1 1988

[TOS] Testing Object-Oriented Systems, Beizer, Addison-Wesley 2000

Chapter 25. Trademark Attributions, Library License Texts, and Source Code

Eclipse is a trademark of Eclipse Foundation, Inc.

IntelliJ is a trademark of JetBrains s.r.o.

Java and all Java-based trademarks are trademarks of Oracle Corporation in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Microsoft, and Windows are trademarks of Microsoft Corporation in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Chapter 26. Legal Notice

Glossary

A

Architecture Aspect — Describes a part of the architecture. Via the "apply" directive an aspect defined in its own architecture file can be reused.

Architecture Model — Consists of a top-level checked architecture file and all recursively applied architecture files.

C

Component — We follow the definition of John Lakos in "Large Scale C++ Software Design": “A component is the smallest unit of physical design.” This is a source file in Java and C# and a source file plus included header files in C/C++. For more details, see Chapter 5, *Getting Familiar with the Sonargraph System Model* .

L

Logical Namespace — Unifies physical namespaces contained in different root directories. More details are given in Section 5.4, “Logical Models” .

M

Module — Represents usually a deployable unit (e.g. an OSGi bundle, JAR file, C# assembly) containing components.

P

Programming Element — The abstract term that represents a type, method, routine, etc. within the different languages. More details are given in Chapter 5, *Getting Familiar with the Sonargraph System Model*

S

(Software) System — Represents the scope of analysis and contains all required resources, i.e. the workspace definition, virtual models, analyzer configurations and Groovy scripts and the analyzed source code.

W

Workspace Profile — Transforms the existing root directories. This is useful for the integration of *Sonargraph* in the build server. More details are given in Section 8.8.3, “Creating Workspace Profiles for Build Environments”.

V

Virtual Model — Represents a sandbox where virtual sets of resolutions (Fix, Ignore, TODO, Delete refactoring, Move/Rename refactoring) can be applied to the system.

Appendix A. Walk Through Tutorial (Java)

This tutorial provides a very concise introduction about the functionality of Sonargraph-Architect. At least an evaluation license is required that can be requested via our web site *http://www.hello2morrow.com* to complete the tutorial.

A little example project is used for illustration - the code itself is by no means meant to be an example for good quality or design. The example project is available via our website *https://www.hello2morrow.com/products/downloads* .

You will learn how to create a system in Sonargraph, define a workspace, examine a system, customize the analysis via scripts, define an architecture and check for compliance in the user interface of the rich client application, use the build integration using Ant and Maven and the Eclipse IDE integration. At the end of a section (except the basic first setup) the current stage of the analysis is referenced, that allows to verify that you reached similar results.

This tutorial is intentionally kept as short as possible. For more detailed information about certain functionality, links are provided that will steer you to the corresponding chapters of the user manual.

A.1. Workspace Definition

The following steps describe the basic setup:

1. Install *Sonargraph-Architect*.
2. On startup, specify a license file.
3. Create a new System using the menu "File" -> "New" -> "System" -> "New System...". Specify a name and location.
4. Create a new Java module using the menu: "File" -> "New" -> "Module" -> "New Java Module...". Specify a name.
5. Right-click on the created module in the Workspace view and select "Manage Java Root Directories...".
6. Specify the root folder of the crm-domain-example project.
7. Detect the root directories and drag&drop them to the module from right to left.
8. Parse / refresh the software system.
9. Check that there are no issues related to the workspace of the system in the Issues view.
10. If Maven is installed on your machine, build the crm-domain-example project via 'mvn clean compile' and refresh the system in Sonargraph. Do you understand, why now workspace issues appear? Open the project in Eclipse, build it there and refresh the system in Sonargraph to remove those issues.
11. Close and re-open the system using the menu "File" -> "Open Recently Used" -> "[System Name]" to see how fast the snapshot loading works.

Related topics:

- Chapter 4, *Initial Configuration*
- Chapter 3, *Licensing*
- Chapter 6, *Creating a System*
- Section 7.1.6, “Creating a Java Module Manually”

A.2. Basic Analysis

The following steps describe the first analysis of a code base based on the existing dependencies, detected cyclic dependencies, detected duplicate blocks and further metrics:

1. Select packages in the Navigation View and open them via the context menu in the Graph view and Exploration view to see/ examine their existing dependencies. Experiment with the different options for creating the representations. Experiment also with the options provided on the opened views to focus on different dependency types, etc.

2. Examine the detected issues on the Issues view.

3. Customize the duplicate code analyzer via the menu "System" -> "Configure..." -> "Duplicate Code" and specify a smaller minimum block length (e.g. 20 lines), so that duplicates are detected. Use the Duplicate Blocks view to open a side-by-side diff for closer inspection.

4. Check the detected package cycles in the Cycle Groups view, open one of them in the Cycle view for closer inspection.

5. Right-click anywhere on the white area in the Cycle view to show the cyclic elements in the Exploration view for further investigation.

6. Right-click anywhere on the white area in the Cycle view to open the Cycle Breakup view. Compute a break-up set for the cycle.

7. Right-click on the proposed dependency to remove and create a new "Delete Refactoring" via the context menu. The created task is now visible in the Tasks and Refactorings views.

8. Create a resolution for another issue in the Issues view. Check the filter options of the view in the top right corner to show only issues of a certain category.

9. Switch the Virtual Model on the application's tool bar (top right of the application) and select the "Parser" model in the combo box. Since the resolutions are specific to the Virtual Model, the issues re-appear in the Issues view.

10. Select the Metrics view and examine the values for the system. Use the combo box at the top-left of the view to examine metric values on other levels (Java Package, Source File, Routine, etc.). Multi-select metrics and see if you can detect a correlation. Define a threshold and check the value distribution on the histogram and pie charts.

11. Check where JUnit is used: Open the Exploration view for the JUnit package that is contained in the "External [Java]" node in the Navigation view.

12. Specify an exclude filter on the Workspace view by right-clicking on one of the "Filter" nodes shown above the root directories. Check the context help (F1) and the related chapter in the user manual to get more information about the different filter types. Experiment with the wildcards and filter types to exclude test related classes. Check that the number of issues in the Issues view changes. Use the filter that keeps the test related classes in the Sonargraph model but marks them as "excluded".

13. Irrelevant issues can be either ignored or filtered. Click on the little downwards-pointing white triangle in the top-right corner of the Issues view and open the filter dialog. Deselect the issue type "Dependency To Excluded Internal Component".

14. **Ctrl+H** opens the search dialog. Enter **test* to identify test classes (they should all be marked as excluded).

    Note: If all internal types are filtered out with references to JUnit, there will be no more dependencies being shown in the representation views (Exploration view, Graph view) to these external types.

15. Create a new script by selecting the "Scripts" folder on the Files view and opening the context menu.

16. Modify the default content to check for all types being excluded.

End of Step 1 (step1_crm-domain-example.sonargraph).

Related topics:

- Section 8.5, “Navigating through the System Components”
- Section 8.11, “Exploring the System”
- Chapter 9, *Handling Detected Issues*
- Section 8.10, “Analyzing Cycles”

- Section 9.1, “Using Virtual Models for Resolutions”
- Section 8.13, “Detecting Duplicate Code”
- Section 8.15, “Examining Metrics Results”
- Section 8.8.1, “Definition of Filters, Modules and Root Directories”
- Section 8.12, “Searching Elements”

A.3. Advanced Analysis

The following steps describe how the scripting functionality can be used for advanced analysis of a code base:

1. Select menu "File" -> "Import Quality Model" and choose some scripts from the Java quality model, e.g. "Java/DesignPatterns/Singleton.xml" and "Java/BadSmells/FindDeadCode.xml".
2. Open the Files view and open the scripts in the Script view. Run them and examine the results.
3. Check the script content and examine the usage of the visitor pattern.
4. Click **F1** to open the context help. Select the JavaDoc for the Script API. Detach the Help view for better usability and examine the available functionality.
5. Write a script that finds all "deprecated" methods and classes. Check "Java/BadSmells/FindDeadCode.xml" for the logic to examine dependencies to annotations.
6. Create issues for the found elements.
7. Use the Exploration view to verify your result.
8. Make the annotation class a script parameter.
9. Add the script to the automatically executed scripts via "System" -> "Configure..." -> "Script Runner".
10. Check that the created issues show up in the Issues view.
11. Modify the issue text in the script. Note that the button "Update Automated Script" is now enabled in the Script view. Transfer the modified content to the automated script and check in the Issues view that the description is changed.
12. Check the CoreAccess.find*() methods in the JavaDoc. Modify the script to search for the @Deprecated annotation and find all incoming dependencies. You can copy from the script UsageOfSystemOutPrintln, contained in the Java quality model.
13. Compare the execution times of the two different approaches (visitor vs. search). Think about the pros and cons of each.
14. Open the Sonargraph system "step2_crm-domain-example.sonargraph" and examine the script "FindMethodWithAnnotationValue". Think about annotation values in your own projects that you want to check.

End of Step 2 (step2_crm-domain-example.sonargraph).

Related topics:

- Chapter 16, *Extending the Static Analysis*

A.4. Architecture: Artifacts, Aspects Files and Standard Connections

The following steps describe how a basic architecture check can be implemented using the Domain Specific Language (DSL):

1. Create a new architecture file "layers" by right-clicking on the folder "Architecture" in the Files view. Keep the default options and make this architecture file "checked".

2. Create artifacts "Business", "Integration", "Foundation" with the standard include patterns. Check the context help (**F1**) for more information.

3. Right-click on the file in the Architecture Files view and open the Exploration view for the architecture model. You should see red arcs representing violations.

4. Connect artifact "Business" with "Integration".

5. Make "Foundation" public, so that it is implicitly accessible by the other artifacts.

6. Save the changes and check in the Architecture view that the correct components are matched for the artifacts. Verify that the architecture-based Exploration view does not show any violations.

7. Create new checked architecture file "application". Create artifacts "Startup", "Application" (**/ddaexample/**), Framework (**/dda/**) with the standard include patterns. Connect the artifacts as indicated by the existing dependencies (use the Exploration view).

8. Use the "apply" statement for "Application" and "Framework" to create the layering defined previously in "layers.arc".

9. Add the "optional" keyword to the "Foundation" artifact in layers.arc to get rid of the "Empty artifact" warning.

10. Create the checked architecture file "component". Create artifacts "Controller", "Data", "Domain", etc that correspond with the packages in the code.

11. Create the connections as the dependencies in the code indicate. Again, use the Exploration view to check for the existing dependencies.

12. Check for architecture violations in the Issues view. Drill-down to the code and examine the root cause for the violations.

13. Remove "layers.arc" from the list of checked architecture files. It is now implicitly used, since the layering is checked and applied in the file "application.arc".

14. An architecture can also be interactively modeled using the Architectural view. Create a new "Architectural View" and create the layering with it.

15. Export the information to an Architecture DSL file and observe the differences to the manually written architecture file.

End of Step 3 (step3_crm-domain-example.sonargraph).

Related topics:

- Chapter 11, *Defining an Architecture*
- Section 11.1, “Models, Components and Artifacts”
- Section 11.3, “Reusing Architecture Aspects”
- Chapter 13, *Interactive Restructuring and Code Organization*

A.5. Architecture: Explicit Interfaces and Connectors

The following steps describe how the access between artifacts can be more sophisticated and restrictive:

1. Create checked architecture file "business".

2. Create artifacts for all domain aspects ("User", "Contact", etc), check the package structure and create the correct number of artifacts.

3. Create a dummy artifact that contains all code that is not matched by the other artifacts.

4. Connect the artifacts with simple "connect" statements.
5. Create default interface for the artifact "Service" in "components.arc" to restrict what is accessible from the outside. Include all, except "**/*DtoVal" types.
6. In "application.arc", create a default connector for "Startup" to restrict access to the outside. Only SetupFactories should be allowed to access types outside of its artifact.
7. Check for found architecture violations.
8. Introduce some dummy references in the code (use the Eclipse project) to produce more architecture violations.
9. "Refresh" in Sonargraph-Architect and check that the new violations appear.

End of Step 4 (step4_crm-domain-example.sonargraph).

Related topics:

- Chapter 11, *Defining an Architecture*
- Section 11.2, “Interfaces and Connectors”

A.6. Architecture: Advanced Connections

The following steps describe how the access between nested artifacts can be more sophisticated and how duplication can be avoided with connection schemes:

1. Apply the component structure of "component.arc" to all artifacts contained in "business.arc".
2. Create more detailed connections between the different components using the nested artifacts explicitly. Check those dependencies in the Exploration view. Note that the artifacts in "component.arc" must be "exposed", so that their default interfaces are visible.
3. Note that the components are always connected in the same way. Create a connection scheme in "component.arc". Check via the context help for more information about connection schemes.
4. Remove "component.arc" from the checked architecture files.
5. Use the new connection scheme and remove all duplication in "business.arc".
6. Remove the obsolete dummy element in "component.arc".
7. Check that you still see the same architecture violations.

End of Step 5 (step5_crm-domain-example.sonargraph).

Related topics:

- Chapter 11, *Defining an Architecture*
- Section 11.8, “Connecting Complex Artifacts”
- Section 11.9, “Introducing Connection Schemes”

A.7. Architecture: Advanced Aspect Files

The following steps describe how the information of aspect files can be changed, so that it fits the context where the aspects are applied:

1. We want to create the structure defined in "business.arc" for the "Business" artifact in application.arc. This can be achieved by "extending" the Business artifact and applying "business.arc":

```
extend Business
{
    apply "./business.arc"
}
```

2. Check the context help and modify the architecture.
3. Use the same mechanism in the "Framework" artifact and simply apply the "component.arc" to generate the same structure there.
4. Remove "business.arc" from the checked architecture files. Only "application.arc" should be left as checked architecture file.
5. Verify in the Architecture View that all artifacts are there and the correct components are matched.
6. Experiment with the workspace filters or include / exclude patterns to adjust the matching.

End of Step 6 (step6_crm-domain-example.sonargraph).

Related topics:

- Chapter 11, *Defining an Architecture*
- Section 11.4, “Extending Aspect Based Artifacts”

A.8. Architecture: Referencing external Artifacts in Aspect Files

Aspect files are a good way to apply the same structure to different parts of the architecture. This section demonstrates how to deal with connections from aspects to artifacts that are outside of the scope of the aspect itself. Let's say, we want to control the access to java.lang.reflect and only allow access to it from the artifact "DataServiceInterface" defined in "component.arc". It is shown why a first naive approach is not working, and how the goal can be achieved using the "require" feature.

1. Add the following artifact to this file:

```
artifact Reflection
{
    strong include "External [Java]/[Unknown]/java/lang/reflect/**"
}
```

2. Save the changes and check the assigned elements to the generated instances of the artifact "Reflection". Since there is only a single java.lang.reflect package, the contained types are matched once for the first instance of a "component", and then there are none left to be matched for further "Reflection" instances in other components. This is clearly not what is needed. We want only a single instance of the "Reflection" artifact.

   "strong include" matchers are disabled in aspects, to avoid accidental misuse and the above mentioned strange matching results. This is the reason why no matches are shown for any or the created "Reflection" artifacts.

3. We need only a single instance of the "Reflection" artifact in the application architecture. The above approach is not working, and it is now demonstrated how it can be achieved with the "require" feature. You need to define the "Reflection" artifact in its own file, let's say "reflectionAccess.arc". You can remove "strong" from the include matcher.
4. Open the architecture file "application.arc" and add the following statement at the bottom:

   ```
   apply "./reflectionAccess.arc"
   ```

   This creates a single "Reflection" artifact at the top-level of the architecture check.

5. Go back to "component.arc" and add the following statement at the top of the file:

```
require "./reflectionAccess.arc"
```

This now allows using artifacts contained in that file, but does not instantiate them again. Do not forget to define the allowed connection between "DataServiceInterface" and "Reflection".

6. Save the changes and check again for the generated "Reflection" instances in the Architecture view. There is now only a single instance and when you open the Exploration view for "application.arc" you can check which dependencies to "Reflection" are allowed and which represent violations.

This concludes the architecture modeling. It is recommended to read the remaining chapters describing the architecture model, e.g. Section 11.10, “Artifact Classes”, for even faster architecture modeling.

End of Step 7 (step7_crm-domain-example.sonargraph).

Related topics:

- Chapter 11, *Defining an Architecture*
- Section 11.11, “How to Organize your Code”

A.9. Headless Check with Sonargraph-Build

The following steps describe how *Sonargraph-Build* can be used on the build server to generate a report and let the build fail on specific issue types:

1. Download and extract *Sonargraph-Build* from the web site *https://www.hello2morrow.com/products/downloads* .
2. If you are interested in Ant: Examine the Ant file "crm-domain-example/build/build.xml" and adjust the properties. Run the target "dist".

   If you are interested in Maven: Examine the Maven file "crm-domain-example/pom.xml" and adjust the properties. Run the goal "package" to create the JAR.
3. Create a workspace profile that uses the created JAR as a target directory.
4. Open the *Sonargraph-Build* user manual and check how the workspace profile can be specified as a parameter (only available for command-line and Ant integration).
5. Run the build and adjust the failset: Check for specific issue types only, specific severities, etc.
6. Let the build fail on architecture violations.
7. Check the details of those architecture violations in the generated HTML report.

End of Step 8 (step8_crm-domain-example.sonargraph). The provided example uses the Maven plugin. Note: Workspace profiles can be used with the command-line and Ant integrations. Maven and Gradle provide the option to override the Sonargraph workspace and use the source and class roots as present in the Maven and Gradle build.

Related topics:

- Section 8.8.3, “Creating Workspace Profiles for Build Environments”
- Chapter 19, *Build Server Integration*

A.10. Check at Development Time with Sonargraph Eclipse Integration

The following steps describe how to use the Sonargraph Eclipse plugin to execute the quality checks at development time. Start the Eclipse IDE and import the example project.

1. Install the plugin. Section 20.1, “Eclipse Plugin” provides details about the plugin's update site.
2. Use the "Sonargraph" menu to activate the plugin by assigning a license file or activation code.
3. Assign the Sonargraph system file. Check that the Sonargraph markers appear on the project and root directories.
4. Check that the Sonargraph issues are listed in the problems view. Update the code by introducing / removing violations and check that the markers get updated after saving the changes. ("Build Automatically" must be enabled in Eclipse.)
5. Create a custom "Problems" view and configure it to only show the Sonargraph issues.
6. Switch back to the Sonargraph-Architect application. Open the package cycle from the Cycle Groups view and open then the Cycle Breakup view. The dependency to breakup is now much easier to identify, because of the existing architecture definition.
7. Create a delete refactoring for the proposed breakup and save the changes.
8. Create a move/rename refactoring for a compilation unit or package of your choice and save the changes.
9. Refresh the system files in Eclipse using the corresponding Sonargraph menu entry. A new task marker should be visible in the Tasks view and also at the position within the file. Move the violating line up and down and save frequently. The task marker moves accordingly. Comment out the offending lines and see the marker disappear.
10. Open the Sonargraph Refactorings view in Eclipse and execute the refactoring via the context menu.
11. Refresh the system in Sonargraph-Architect and notice that the status of the refactoring changed to "Potentially Done". The architect can now review the changes and delete the task.

Related topics:

- Section 20.1, “Eclipse Plugin”
- Section 20.1.1, “Assigning a System”
- Section 20.1.2, “Displaying Issues and Tasks”
- Chapter 10, *Simulating Refactorings*
- Section 10.4, “Best Practices”
- Section 20.1.7, “Execute Refactorings in Eclipse”

Appendix B. Tutorial - Java

This is a step-by-step tutorial illustrating the analysis of the Open Source project *Apache Cassandra*. It will be demonstrated how to setup the workspace and quickly get an overview of the state of software quality. Some issues are reported by *Sonargraph* right away without further configuration and it will be shown how to analyze cyclic dependencies and duplicate code blocks. *Sonargraph* also allows to easily analyze the dependency structure in more detail. Next it is illustrated how the GroovyScript API can be used to monitor virtually anything that can be detected via static code analysis. The last chapter shows how you can share the results of the analysis.

This tutorial is intentionally kept as short as possible. For more detailed information about certain functionality, links are provided that will steer you to the corresponding chapters of the user manual.

B.1. Setup the Software System

Apache Cassandra is built using *Apache Ant* therefore the Sonargraph Workspace needs to be set up manually, i.e. the definition of *modules* and the location of source and class files. For systems built with Maven or that are represented by an Eclipse or IntelliJ Workspace, the Sonargraph Workspace can be set up automatically. For more information, see Chapter 6, *Creating a System* and Section 8.8, “Managing the Workspace” .

B.1.1. Create a new Software System

The initial system is created using "File" → "New" . Select the entry "Manual System" and continue. In the following wizard page, provide the name of the *software system* and specify where the system's files will be stored. For more information about the file structure, see Chapter 5, *Getting Familiar with the Sonargraph System Model* .

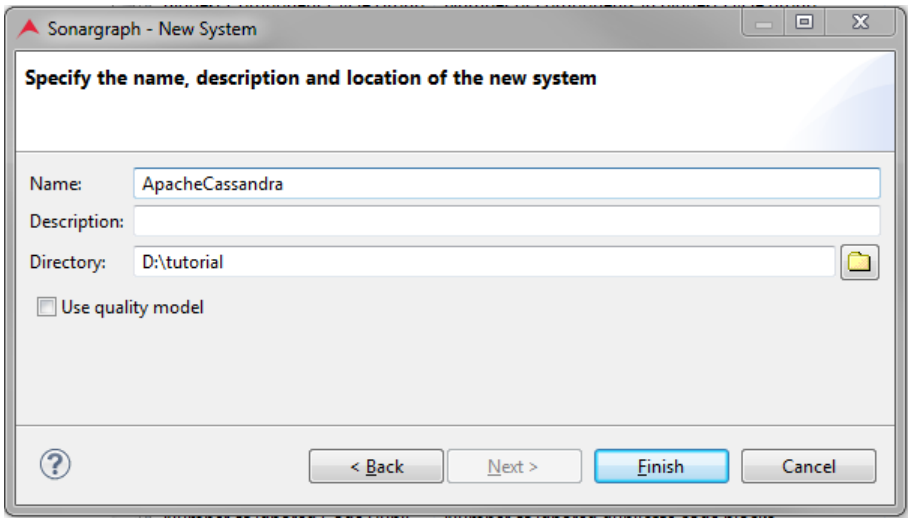


Figure B.1. New Manual System Wizard

Using a Quality Model is explained in Section B.6, “Share Results” , leave this option unchecked for now.

B.1.2. Define the Workspace

As a next step, we need to create one or several modules. A module is the container for source and class root directories and usually represents a Maven module or an Eclipse / IntelliJ project. We will start creating a single module, and refine the workspace later.

A module is created by selecting "File" → "New" and selecting the wizard entry "Manual Java Module". Define the module's name and optionally provide a description.

As a next step, we let *Sonargraph* search for directories containing source and class files. Right-click on the created module in the Navigation view. Select the context menu entry "Manage Java Source/Class Root Directories/Archives..." and specify the root directory of the *Apache Cassandra* project on your disk.

Start the detection and wait until it completes. Now you can move the found directories via drag&drop from the right to the left. We omit directories containing examples or test code.

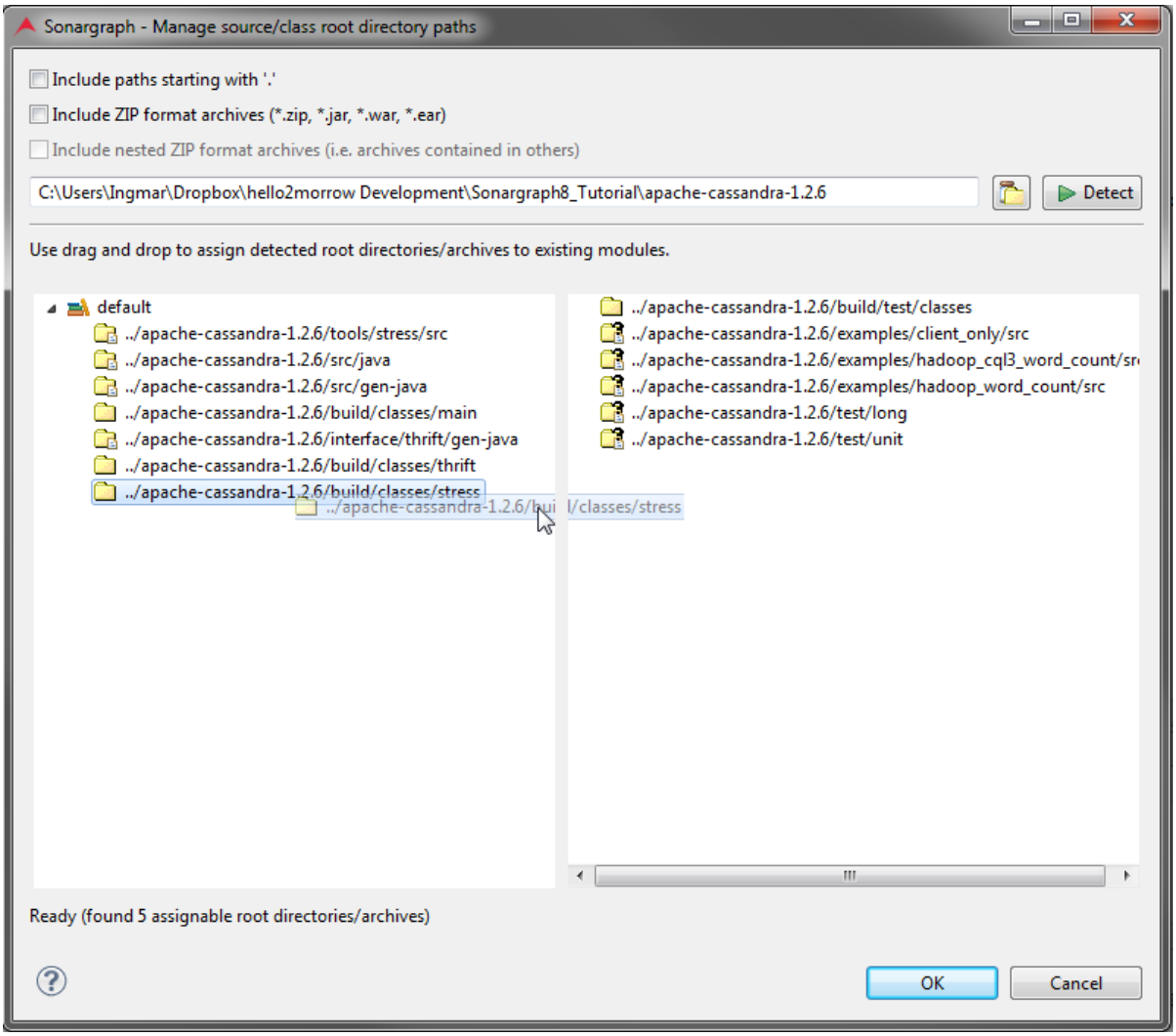


Figure B.2. Root Directories Dialog

The workspace configuration can be examined on the Workspace view. We can see that it is probably best to create additional modules "thrift" and "stress" using the same approach as previously. Now you can use drag&drop in the Workspace view to move directories from the "default" module and rename "default" to "main" using either the context menu or the **F2** shortcut.

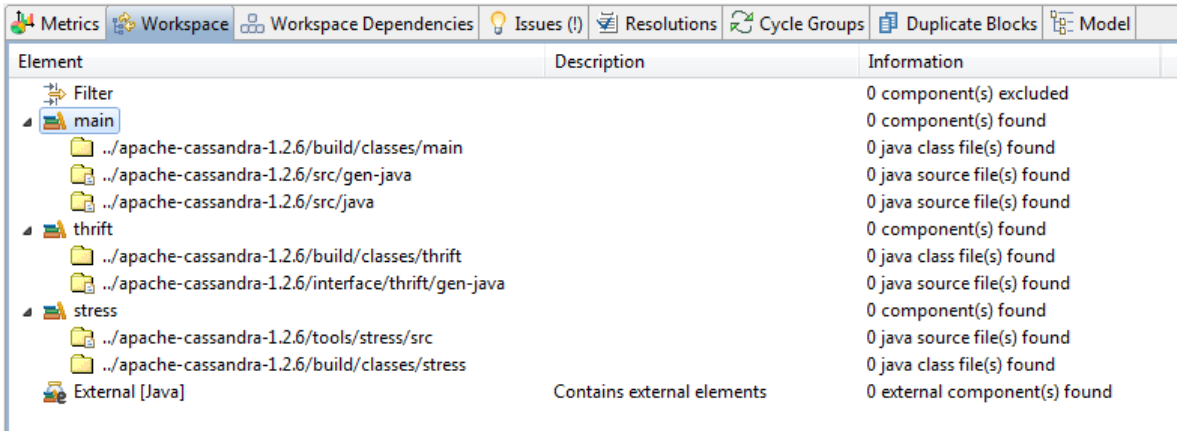


Figure B.3. Workspace View

NOTE

The order of the root directories matters: In case there are classes with the same fully qualified names, the first one found wins. You can change the order of root directories of manual Java modules within the Workspace view using drag&drop.

B.1.3. Define Module Dependencies

Managing module dependencies is especially important when using frameworks like OSGi where you could have a module X and a module Y each one of them containing a type with the fully qualified name a.b.c.Type. Such conflicts are resolvable by defining manual module dependencies. These module dependencies control the type resolution when creating parser dependencies trying to locate the 'to' (Java) type.

If the modules do not define any dependencies all types are visible in all modules. Once there are type resolution conflicts which would show up as 'Ambiguous Target Type' issues manual module dependencies can be used to decide which type(s) should be accessed from which module. If module Z accesses the type a.b.c.Type defined in module X and module Y the conflict is resolved by simply defining a manual module dependency between module Z and the correct target module.

NOTE

If you know how modules are supposed to use each other, define the workspace dependencies explicitly.

B.1.4. Parse the Workspace

To parse the workspace, either chose the menu item "System" → "Refresh" or use the shortcut **F5** . After the parsing has completed, the detected classes are displayed with their package structure in the Navigation view and the Workspace view shows how many items are found in a directory.

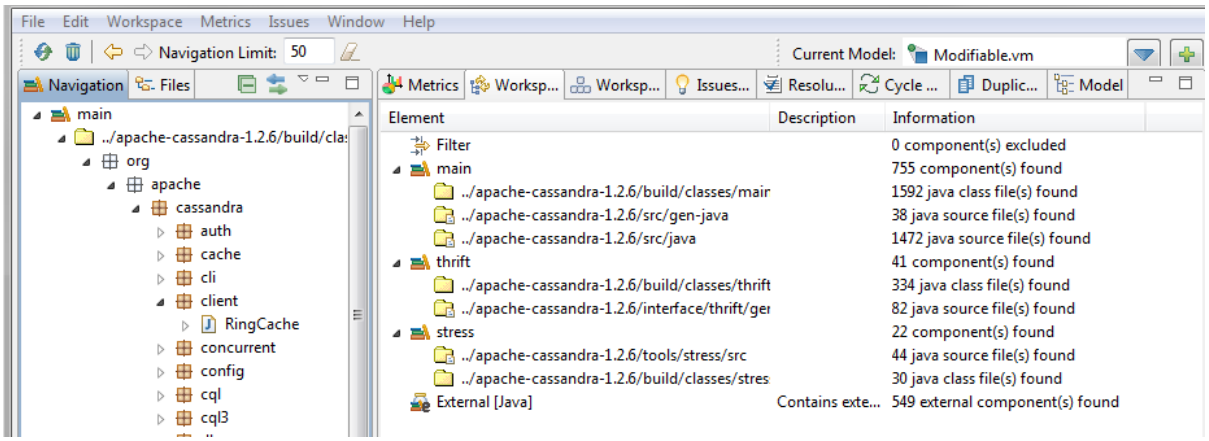


Figure B.4. Workspace View After Parsing

B.2. Initial Analysis

After having parsed the workspace as described in the previous section, basic information about the number of processed source and class files is provided on the workspace view. This section explains how the results of the metric calculation can point out problematic areas.

B.2.1. Detect Problems Using Standard Metrics

The Metrics view is separated into two general areas: The "System Level" and the "Element Level". The "Element Level" tab allows to focus on different levels, e.g. modules, root directories, packages, types, routines, etc. as shown in the following screenshot.

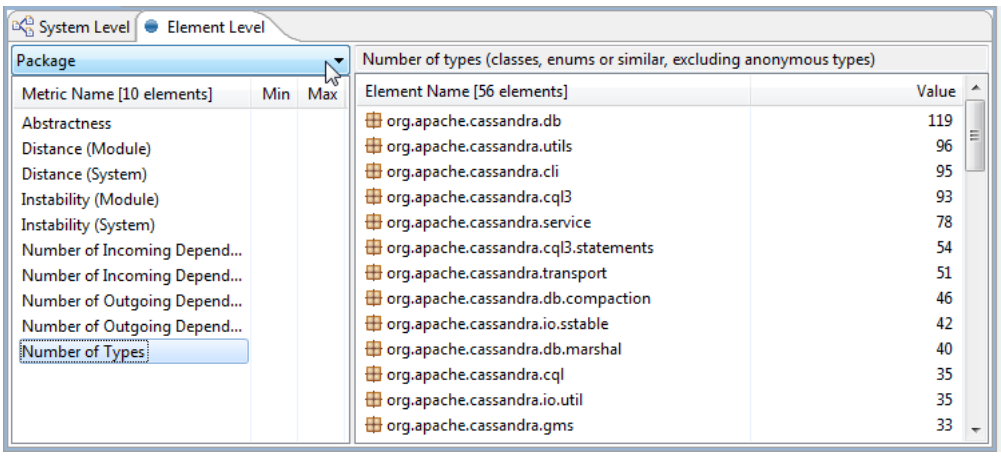


Figure B.5. Metrics View

B.2.2. Adjust Metric Thresholds

The Metrics view allows defining lower and upper-level thresholds for metrics. Issues are created for those elements violating these thresholds and they are clearly marked in the table. If metric thresholds are specified, those values will be saved into a file located at: `<Sonargraph System directory>/Analyzers/MetricThresholds.xml` . Thresholds are defined either via the context menu of a metric or the menu entry "Metrics" → "New Threshold..." .

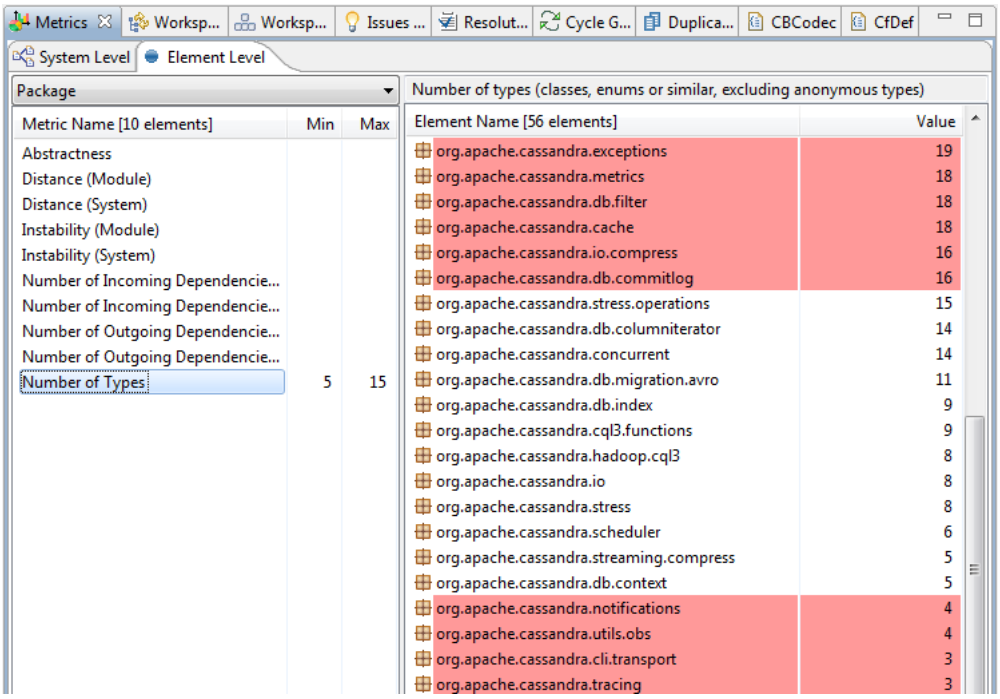


Figure B.6. Metrics View Highlighting Thresholds Violations

B.3. Problem Analysis

Sonargraph lists all problems found in the Issue view. At this stage the view lists issues of type Cycle Group, Duplicate Block and Threshold Violation. If some issues types should be filtered this can be achieved using the filter option as shown in the screenshot.

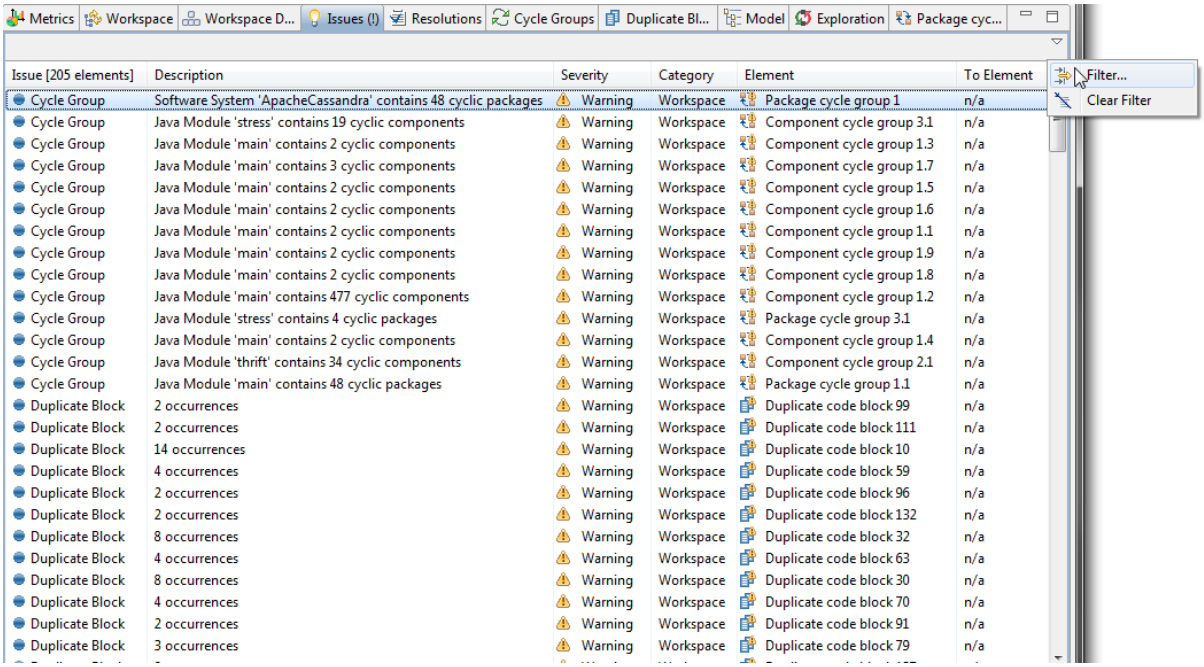


Figure B.7. Filter Issues

Check Section 9.2, “Examining Issues” for more details about filtering.

B.3.1. Examine Cycles

Cycles between any elements should be avoided as they have a negative impact on various properties of the *software system*, e.g. testability, maintainability, understandability, to name a few. Cycles can be examined in more detail by opening the Cycle Groups view. This view additionally shows the involved source files for component cycle groups.

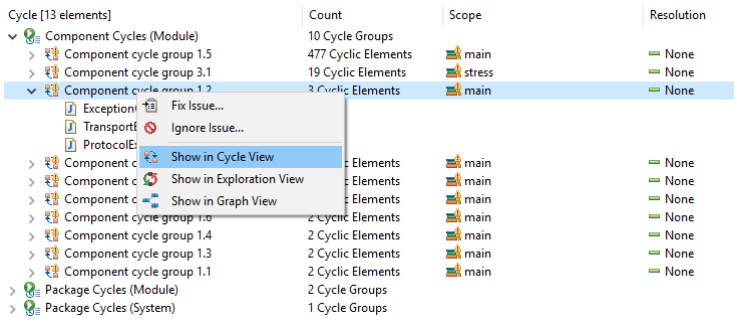


Figure B.8. Cycle Groups View

To analyze individual cycle groups, open the Cycle Group view via the context menu. This view shows the involved elements and when selecting an element or a dependency, details to the dependency are shown in the Parser Dependencies views. Drill-down to the source code is supported via double-click on a dependency.

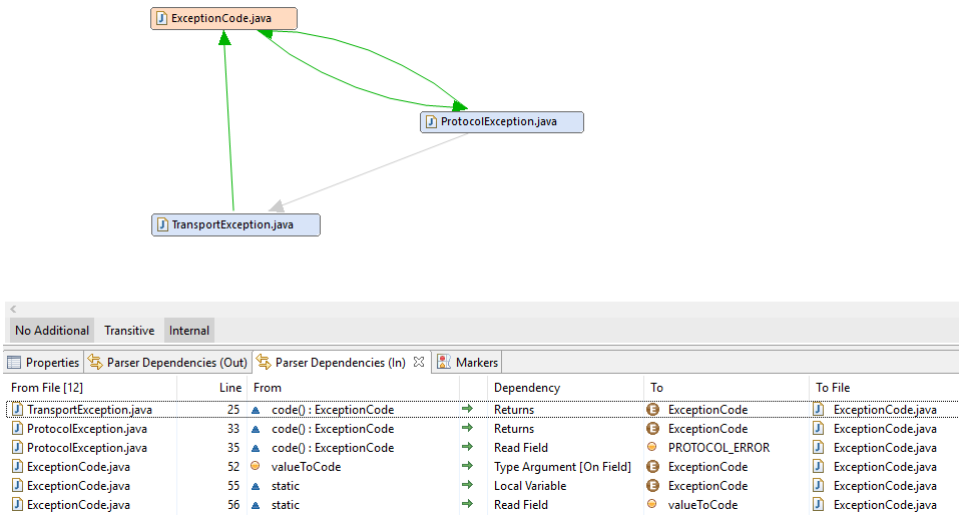


Figure B.9. Cycle View

B.3.2. Examine Duplicate Code

Duplicate code is another type of issue shown in the Issues view. Details of duplicates are shown in the Duplicate Code Blocks view that can be opened via the context menu. This view shows more details about individual duplicates, i.e. the block length, tolerance, and involved files.

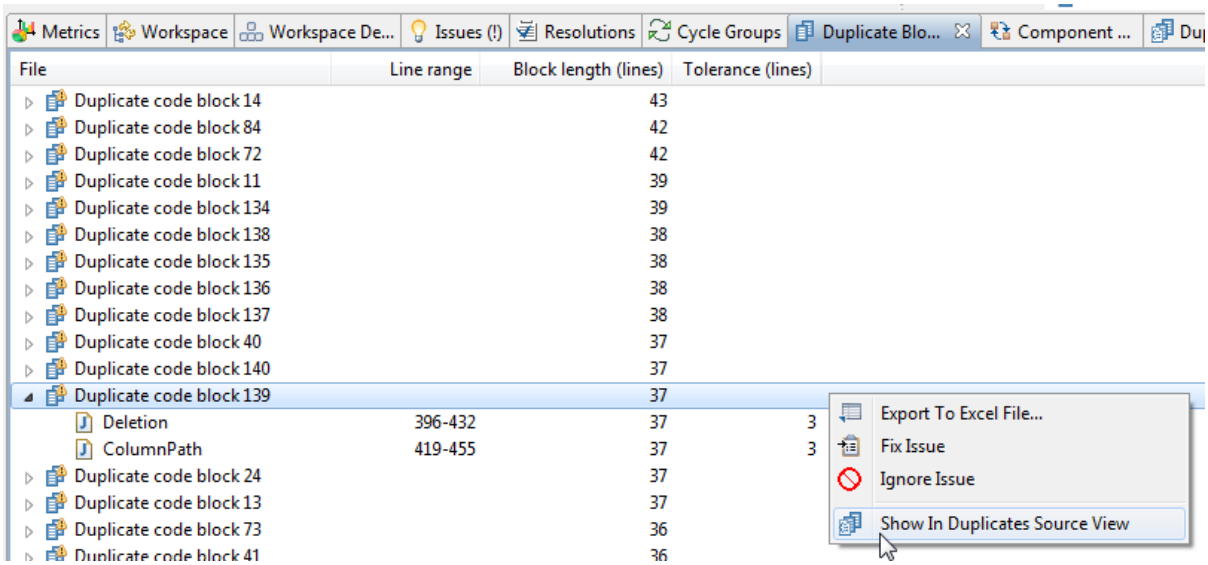


Figure B.10. Duplicate Code Blocks View

The Duplicate Source view highlights the duplicate block and marks the lines within a block that are different.

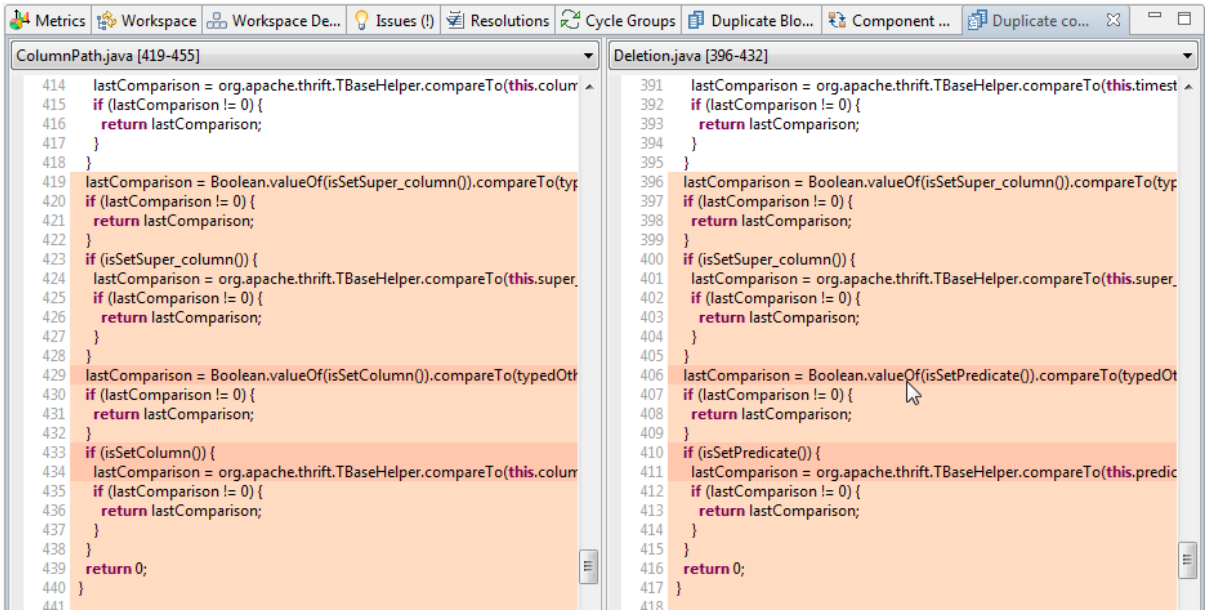


Figure B.11. Duplicate Source View

For more details about configuring the duplicate code analysis, check out Section 8.13.1, “Configuration of Duplicate Code Blocks Computation”.

B.3.3. Handle Issues

Sonargraph allows to handle issues in two different ways via the context menu of the Issue view:

- Ignore: Signifies that the issue will be dealt with later.
- Fix: Signifies that the issue needs to be fixed. An assignee can be specified.

Additionally a TODO-resolution can be defined for any element.

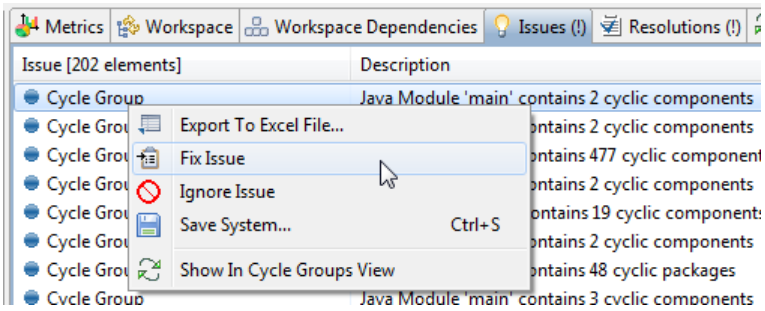


Figure B.12. Add TODO Issue

The details about created resolutions are shown in the Resolutions view, including how many elements are matched. The lower section displays the matched elements and allows the drill-down into a detailed view via the context menu.

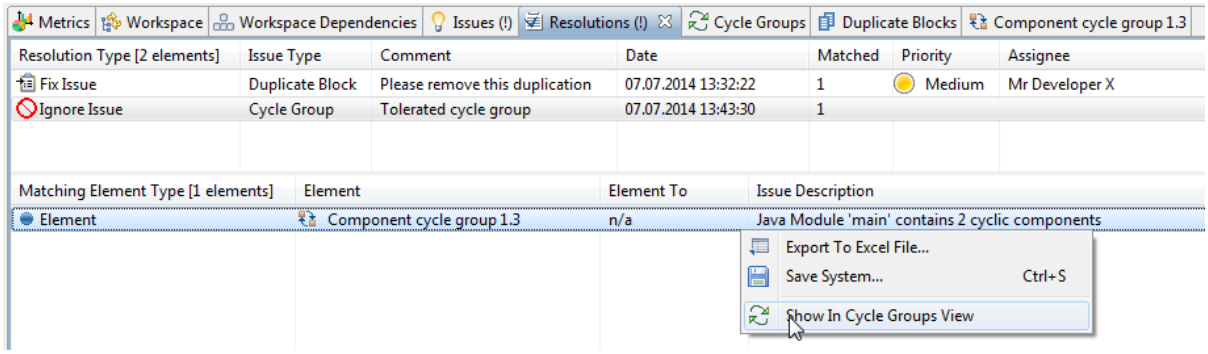


Figure B.13. Resolutions View

More information about issues, resolutions and quality models can be found in Chapter 9, *Handling Detected Issues*.

B.4. Detailed Dependency Analysis

Sonargraph provides different views to analyze dependencies between elements. The most important are the Exploration, Graph and Dependencies views.

B.4.1. Explore Dependencies

The Exploration view can be opened for an arbitrary selection of elements in the Navigation view or via the context menu within other views.

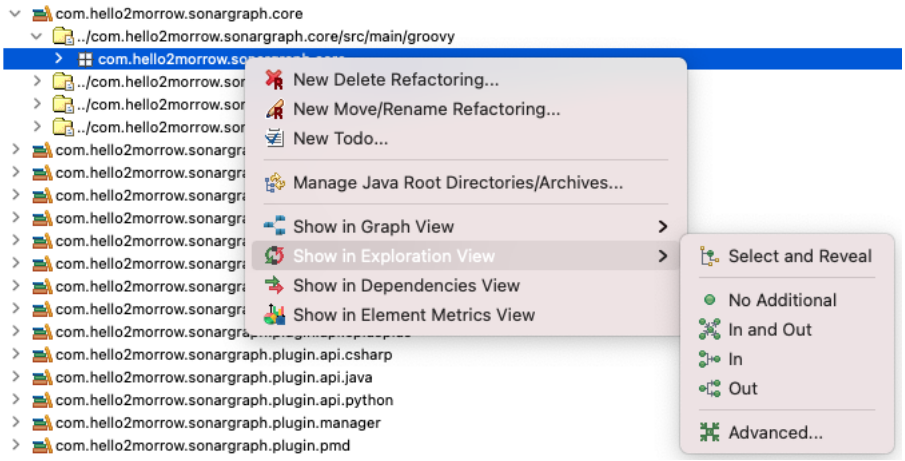


Figure B.14. Open in Exploration View

The Exploration view orders the displayed elements, with elements on top having more outgoing dependencies and elements on the bottom having more incoming dependencies. Clicking on the plus-sign of an element opens nested elements, allowing to drill down to fields. Arcs represent detected dependencies, which can be analyzed in more detail in the Parser Dependencies views and also in the Source view.

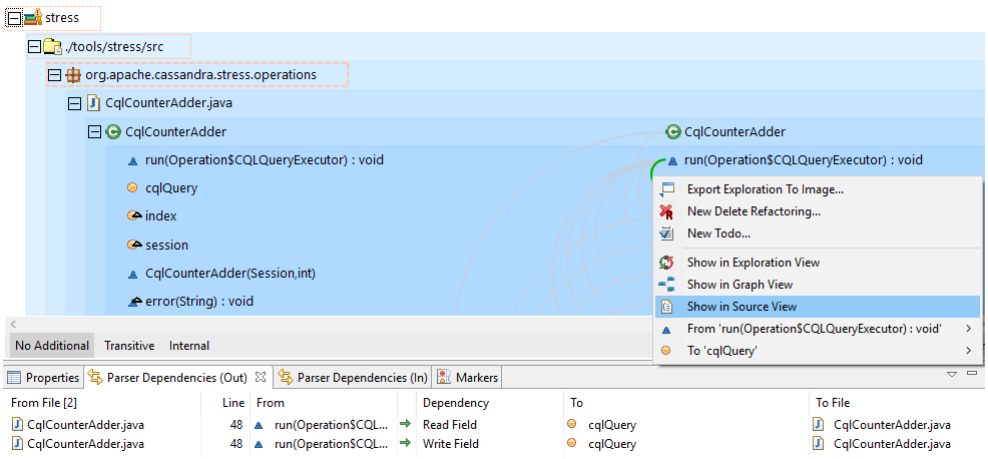


Figure B.15. Exploration View Drilldown

This view offers interactions such as focus and unfocus which can be used to explore the dependencies of an arbitrary selection of elements inside the Exploration view and also provides highlighting, marking and zooming which can be helpful in the analysis of the content that is being displayed. Further details are explained in Section 8.11.2.1, “Focus Modes” and Section 8.11.2.1, “Applying Focus”.

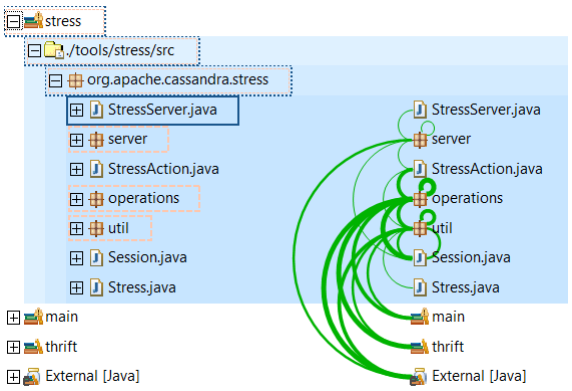


Figure B.16. Exploration View Interactions

B.4.2. Check how Elements are Connected via Graph View

The Graph view can also be opened for an arbitrary selection of elements from the context menu. It shows the selected elements in a levelized graph. For more details about the advantages see Section 8.11.2.2, “Levels”. Again, existing parser dependencies can be analyzed using the Parser Dependencies Views.

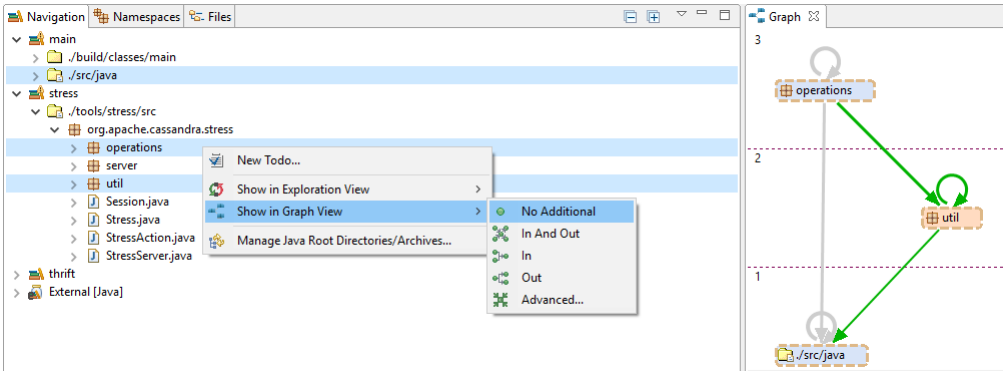


Figure B.17. Graph View

Like the Exploration view, the Graph view also offers focus and unfocus interactions to check the dependencies of an arbitrary selection of elements inside the view. It also offers highlighting and zooming to help in the analysis of the currently displayed content. Additionally, it will automatically group elements that form cycles to make the graphs more comprehensible. By right-clicking on a Cycle Group, it is possible to open the Cycle view to observe the detail of the elements that make part of it. Further details are explained in Section 8.10.2, “Inspecting Cyclic Elements”.

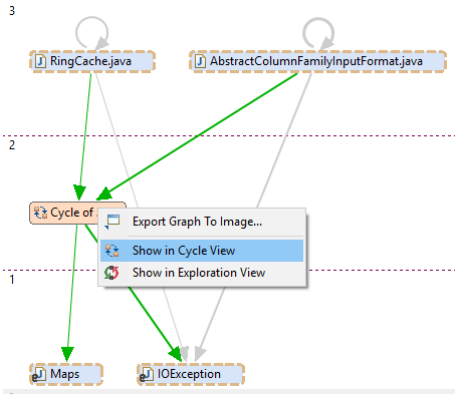


Figure B.18. Graph View Interactions

B.4.3. Check how Elements are Connected via the Dependencies View

If you rather like tabular representations, existing dependencies of elements can be examined in the Dependencies view which is again available via the context menu, but only for a single selection. The Dependencies view is separated into three parts, each allowing to drill down and analyze a specific dependency in more detail. More features of this view are explained in Section 8.11.4, “Tabular System Exploration”.

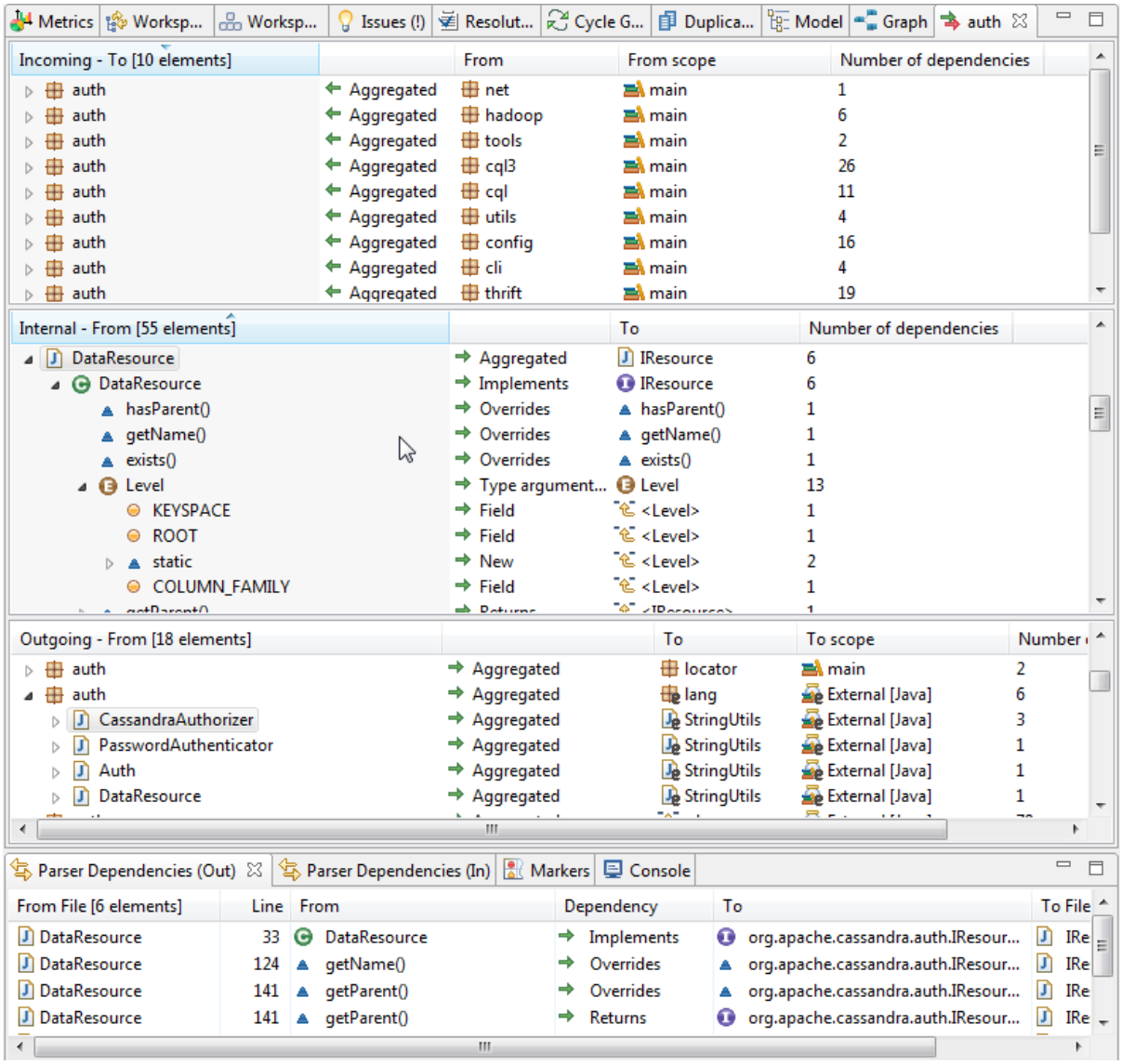


Figure B.19. Dependencies View

B.4.4. Search for Elements

In case you are interested in seeing the dependencies of a particular type, but want to spare the effort of navigating to it via the Navigation view, the Search dialog provides this shortcut. See (Section 8.12, “Searching Elements”)

B.5. Advanced Analysis With Scripts

Sonargraph provides an API to access its internal model by Scripts written in Groovy. *Sonargraph* scripts can introduce and calculate new metrics, add issues to the model, and build a list or tree of model elements or dependencies.

B.5.1. Create a New Script

You can create a new script either by menu "File" → "New" → "Other" → "Script" , or by selecting an existing script directory and selecting "Create Script..." from the context menu. The "New Script Wizard" will start, and at least a name for the new script must be given. After pressing "OK" the new script shows up in a Script view, where you can edit, compile and execute it.

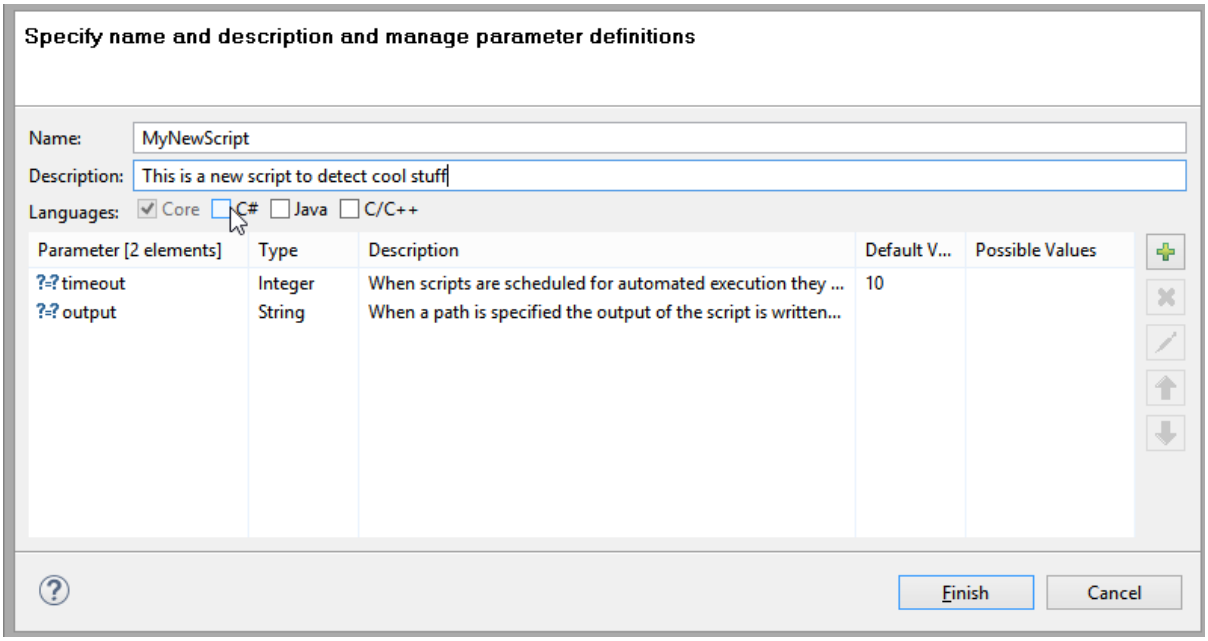


Figure B.20. Create a New Script

More details are provided in: Section 16.3, “Creating a new Groovy Script”

B.5.2. Execute Existing Script

To execute an existing script, go to "Files" tab and open directory "Scripts". Double click on a script and a Script view will open. The Script view consists of three parts: On the top the source of the script, in the middle the "Compile"/"Run" and "Update automated Script" buttons, and on the bottom five tabs for the result of the script. Press "Execute" and the script will run. Every tab in the bottom of the Script view that contains some data will show an exclamation mark in its title.

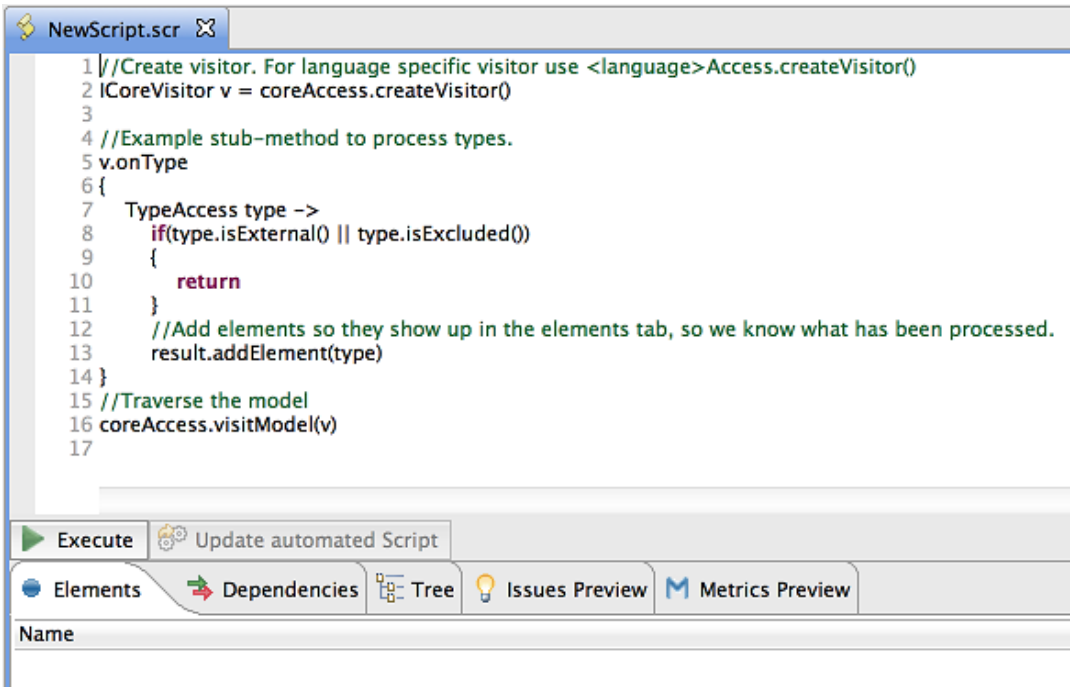


Figure B.21. Execute a Script

More details about how find specific elements or dependencies, create metrics and issues is provided in: Section 16.5, “Producing Results with Groovy Scripts”

B.6. Share Results

Once a *system* has been analyzed it is important to share the findings with others. This section explains the different types of export offered by *Sonargraph* .

B.6.1. Work with Snapshots

Sonargraph offers the capability to create snapshots to preserve the state of a *system* at any given time. It is available via "File" → "Save Snapshot..." and creates an archive file containing all the generated information in compact binary format together with or without the source files. This snapshot can be archived or passed on to co-workers for further evaluation. No data contained in a snapshot can be modified.

NOTE: A snapshot can also be created with *Sonargraph-Build*.

Snapshots can be opened via "File" → "Open From Snapshot..." .

An opened *system* can also be 'attached/detached' to/from an existing snapshot. This can be helpful in case you have a *system* which takes a long time to parse. Create a snapshot (ideally with sources) in your automated build with *Sonargraph-Build*, attach to it from *Sonargraph* using the corresponding system and continue to work on architecture aspects, scripts and others.

NOTE: When attached to a snapshot workspace modifying commands are disabled (e.g. create module, ...).

B.6.2. Define Quality Standards using Quality Models

A Quality Model allows defining a standard configuration that needs to be applied to several *systems*. It contains analyzer configuration (e.g. for metric thresholds) and scripts. Menu entries "File" → "Export Quality Model..." and "File" → "Import Quality Model..." can be used to export and import a Quality Model. Additionally, a Quality Model can be specified during creation of a system. See Section 6.4, “Quality Model” .

B.6.3. Export to Excel

Many *Sonargraph* views offer the export of the displayed data to *Microsoft Excel* . In case you are working on a non-Windows platform, the exported files can also be opened using *Open Office Calc* . For example the Metrics view lets you export all Metrics data (system level and all element levels) into a single Excel file.

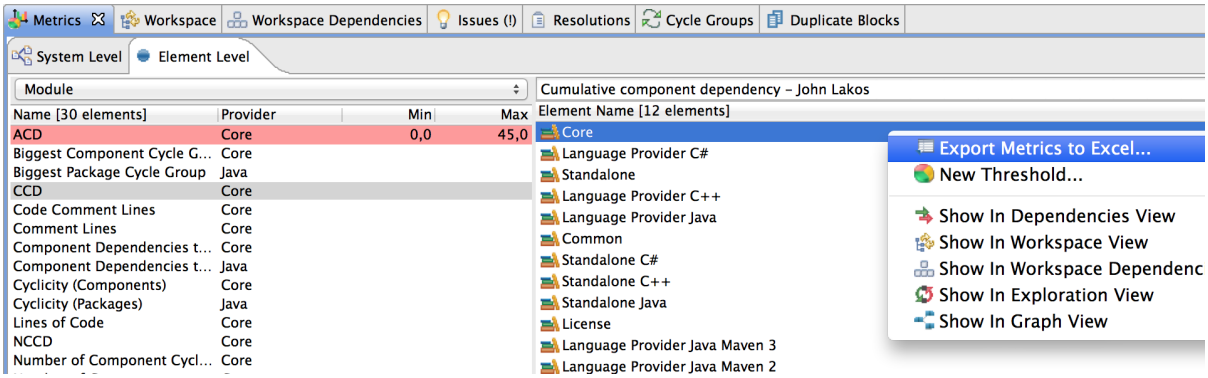


Figure B.22. Export Metrics to Excel Context Menu

Appendix C. Tutorial - C#

This is a step-by-step tutorial illustrating the analysis of the Open Source project *NHibernate*. It will be demonstrated how to setup the workspace and quickly get an overview of the state of software quality. Some issues are reported by *Sonargraph* right away without further configuration. *Sonargraph* also allows to easily analyze the dependency structure in more detail. As this functionality is mostly language-independent, we refer you to the appropriate sections within the Java Tutorial.

This tutorial is intentionally kept as short as possible. For more detailed information about certain functionality, links are provided that will steer you to the corresponding chapters of the user manual.

C.1. Setup the Software System

This section describes how the *Sonargrah* system is setup for the NHibernate project. As a precondition, ensure that NHibernate builds successfully in Visual Studio (or whatever you use for C# development). The tutorial is based on NHibernate master branch, checked out on 2019-02-11.

Sonargraph offers to import Microsoft Visual Studio C# Solution files. Select menu "File" → "New" and select the wizard "System based on C# Visual Studio Solution file". Specify the name of the directory of the *Sonargraph* system and where its files will be stored. It is a best practice to store the system close to the actual source code and place it under version control. Using a Quality Model is explained in Section B.6, “Share Results” ; you can leave this option unchecked for now.

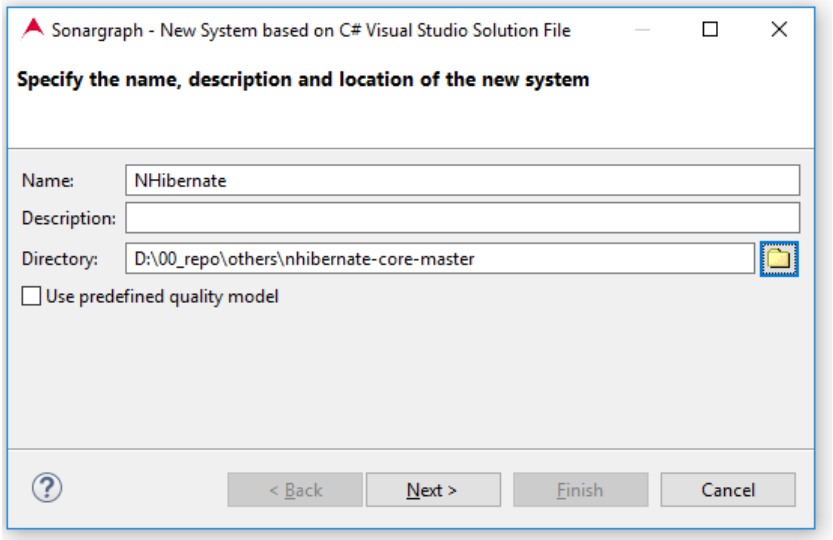


Figure C.1. System based on C# Visual Studio Solution File

The next wizard page allows to specify the Microsoft Visual Studio Solution file and then lets you pick the modules you want to analyze. If there are different 'flavors' (like 'net6.0) for a module they will all be displayed. If you select am module with a flavor the other choices must be of the same flavor or without a flavor. Before importing the system please make sure that you are able to build the solution on the computer where Sonargraph is running. By the way, you can change the selection of analyzed modules at any time by selecting "File" → "New" → "Module" → "Update C# module selection".

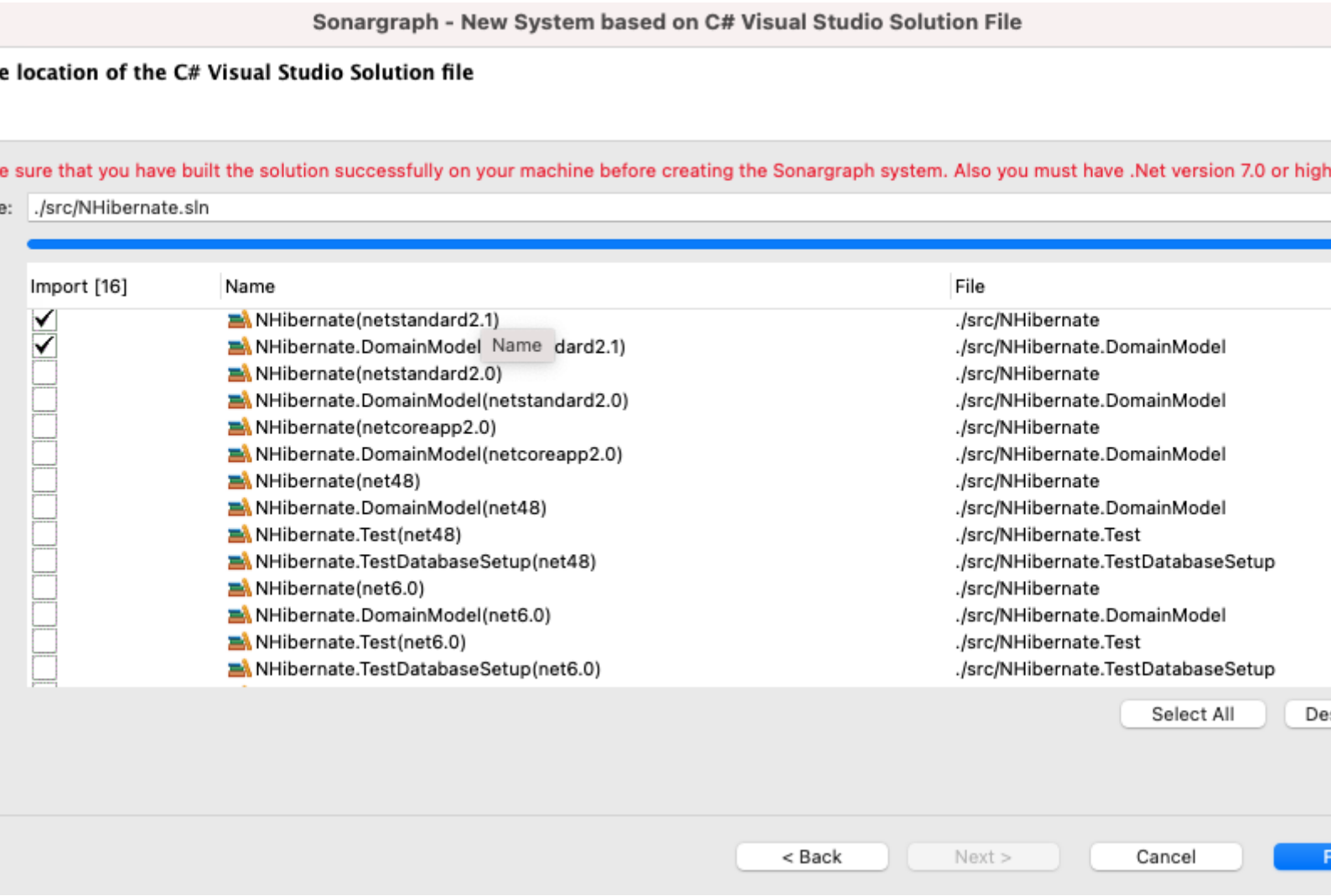


Figure C.2. Select C# Solution File, Configuration and Platform

After the system creation was successful, it's now time to execute "refresh" via the second icon in the toolbar or via **F5** to start the parsing. After the parser finished it is a good idea to check the "C# Parser Log" via "Windows / Show View / C# Parser Log". It contains the error messages of the Roslyn parser that is used for the analysis. If you see a lot of errors there it probably means that not all references could be resolved successfully. Even if there are many errors you can still work with Sonargraph. It just means that a couple dependencies were not resolved properly.

The next step is to open the Issues view and examine the detected cycle groups and duplicate code block issues.

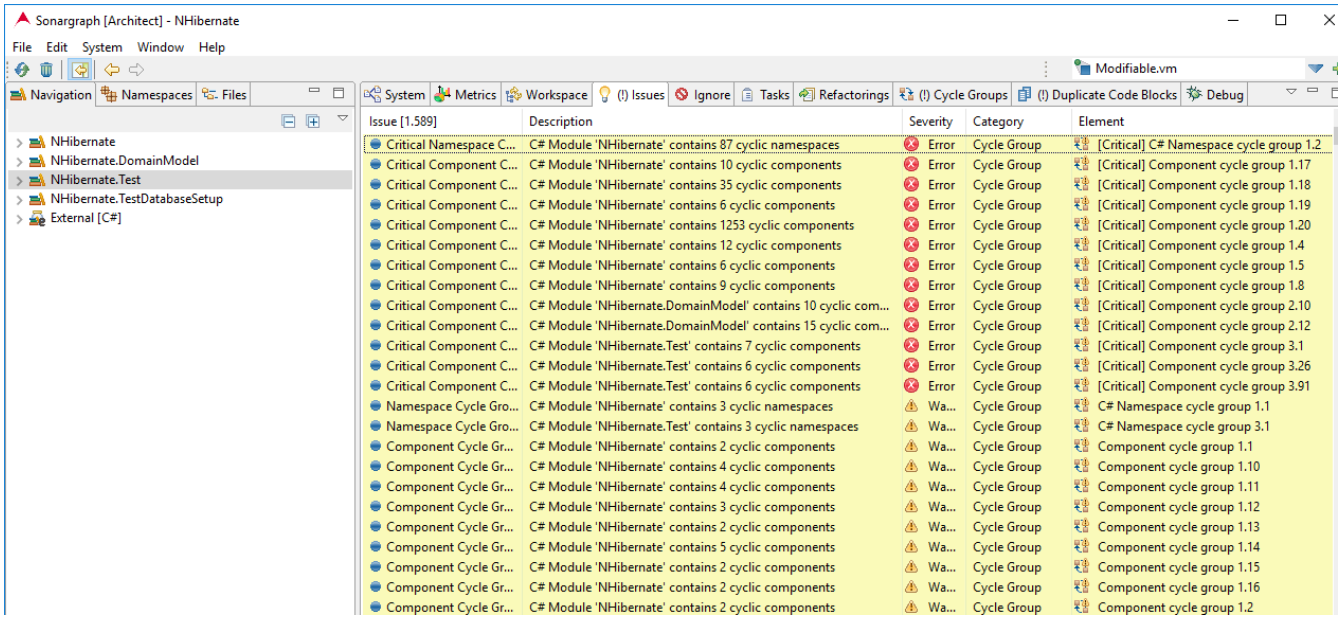


Figure C.3. Select C# Solution File, Configuration and Platform

Related topics:

- Chapter 3, *Licensing*
- Chapter 6, *Creating a System*
- Section 7.3, “Creating or Importing a C# Module”
- Chapter 9, *Handling Detected Issues*

C.2. Further Steps

After the workspace for a C# system has been defined, the further steps to analyze are the same as for a Java system. Please check the following sections of the Java tutorial:

- Section B.2, “Initial Analysis”
- Section B.3, “Problem Analysis”
- Section B.4, “Detailed Dependency Analysis”
- Section B.5, “Advanced Analysis With Scripts”
- Section B.6, “Share Results”

Appendix D. Tutorial - C++

This is a step-by-step tutorial illustrating the analysis of the Open Source project *POCO*. It will be demonstrated how to setup the workspace using different importers and quickly get an overview of the state of software quality. Some issues are reported by *Sonargraph* right away without further configuration. *Sonargraph* also allows to easily analyze the dependency structure in more detail. As this functionality is mostly language-independent, we refer you to the appropriate sections within the Java Tutorial.

This tutorial is intentionally kept as short as possible. For more detailed information about certain functionality, links are provided that will steer you to the corresponding chapters of the user manual.

D.1. Setup the Software System - Compiler Definitions

Sonargraph internally uses the C/C++ parser of EDG (Edison Design Group, www.edg.com). To successfully parse your code the parser must be able to emulate your real compiler. To do that we use the concept of compiler definitions. Such a definition contains information like where to find the implicit system include directories and a list of predefined macros. Sonargraph comes with a couple of ready to use compiler definitions for the GNU compiler family, CLang and a few others. For Visual C++ you have to tell Sonargraph where Visual Studio is installed on your computer. You can add Visual Studio installations via the C++ preference pages under the Windows/Preferences menu.

There is always one compiler definition that is considered to be active. This is the one that is used for parsing your code. After a successful parser run Sonargraph will remember the compiler definition used and automatically activate it the next time you open the same project. When you parse a project for the first time we will use the compiler definition that is currently activated. To check your available compiler definition and to make sure the right one is activated you can go to the C/C++ preference pages. From there you can manage the available compiler definitions, modify existing ones or even create new ones.

If there is no compiler definition for your compiler we recommend to use our compiler definition wizard to create one. You start the wizard by selecting "New..." → "Configuration / New Compiler Definition". If you have used our other tool Sotograph before you can import Sotograph compiler definitions directly in the first step of the wizard. Otherwise just follow the instructions of the wizard.

If the C/C++ parser finds issues, they will be recorded in the C/C++ parser log window. You can open the parser log by selecting "Windows" → "Show View - C/C++ Parser Log". Errors recorded there are usually not a problem for the quality of the analysis. In the worst case a dependency might be missing if the parser cannot properly resolve a symbol. If there are many problems in this view this could indicate a problem with your compiler definition. An parser run will only fail if there are too more than 1000 errors in a compilation unit or if referenced include files cannot be found.

D.2. Setup the Software System - Capture Compile Commands with ccspy

This section describes how to create a new C++ system using ccspy. Select "New" → "System based on ccspy capturing directory" . Specify the name of the directory of the *Sonargraph software system* and where its files will be stored. It is a best practice to store the *software system* close to the actual source code and place it under version control. Using a Quality Model is explained in Section B.6, “Share Results” ; you can leave this option unchecked for now.

You will have to do a complete rebuild of your system where you replace your compiler with ccspy in your make or cmake configuration. ccspy will then record all compile commands in a designated directory and then call your real compiler. This means you can keep ccspy in your make or cmake configuration. It should only minimally increase the time needed for a build, but will ensure that Sonargraph will always know the latest compile commands.

ccspy is delivered with Sonargraph in the bin directory of the Sonargraph installation. The documentation for properly running and configuring ccspy can be found on *Github*.

Once your build is finished your ccspy directory should contain one text file for each compilation unit. The ccspy import wizard will guide you through the neccessary steps to complete the Sonargraph system setup and works basically in the same way as the cmake import wizard.

The next step to get started with your analysis is perform a refresh so the required information is picked up from the set-up modules.

D.3. Setup the Software System - Visual Studio Import

This section describes how to create a new C++ system by importing a Visual Studio Solution file. Select "New" → "System based on C/C++ Visual Studio 2010 Solution file" . Specify the name of the directory of the *Sonargraph* system and where its files will be stored. It is a best practice to store the system close to the actual source code and place it under version control. Using a Quality Model is explained in Section B.6, “Share Results” ; you can leave this option unchecked for now.

On the next wizard page, select the solution file to import and the configuration and platform combination.

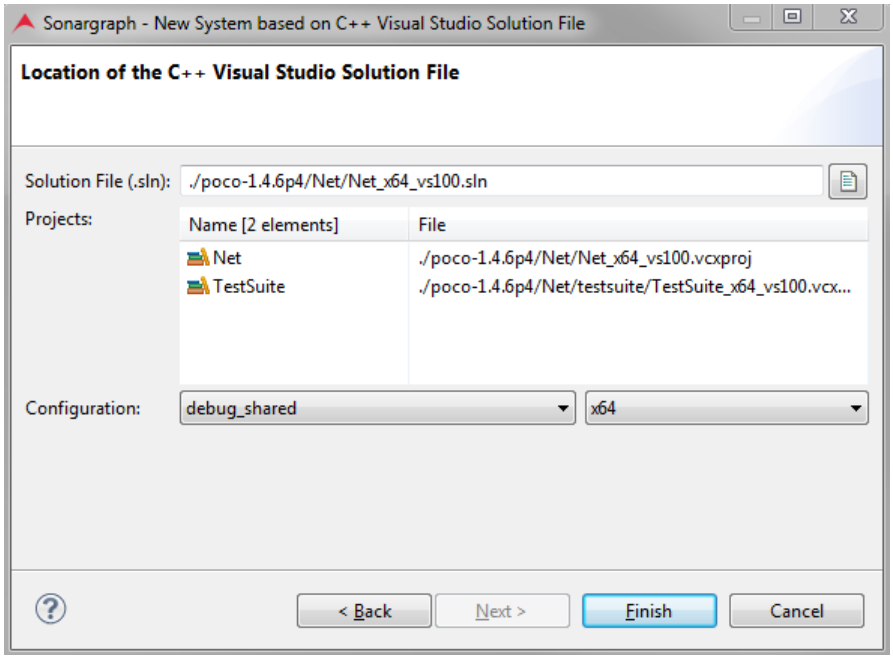


Figure D.1. Specify Visual Studio Solution File

If the system is refreshed and the active compiler definition does not match the imported solution, you might run into the following two problems which can both usually be fixed by selecting the correct compiler definition on the preference page as described in Section 4.7, “C/C++ Compiler Definitions” . In this case, the correct compiler definition is "VisualCpp_11.0_x86".

1. MSBuild Exception: Sonargraph uses internally MSBuild to determine the source files to compile and the compiler options to be used. Usually, if MSBuild fails some built-in variable is not resolved correctly.

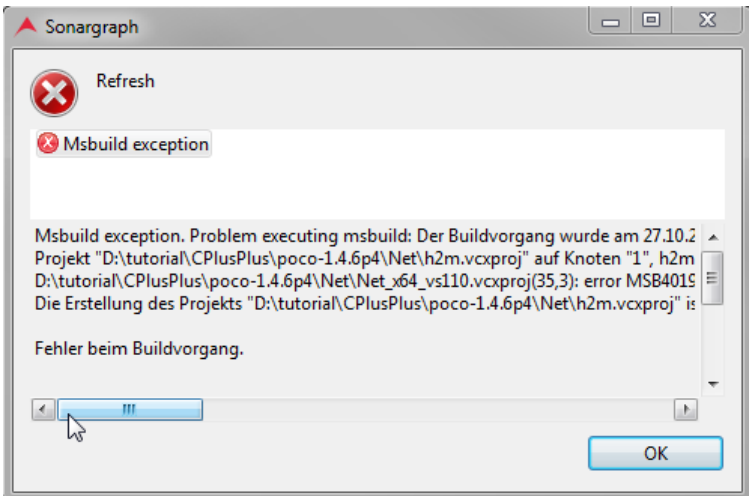


Figure D.2. MSBuild Exception

2. Parse Error: The parsing is aborted if a header file cannot be found. Check the folder where the header file can be found on disk and select a compiler definition that contains this folder as part of its --sys_include options.

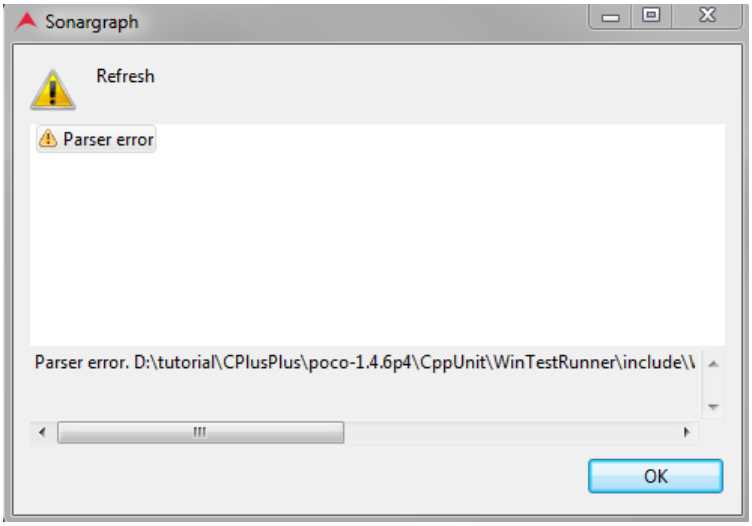


Figure D.3. Parse Error (Missing Header File)

D.4. Further Steps

After the workspace for a C/C++ system has been defined, the further steps to analyze it are the same as for a Java system. Please check the following sections of the Java tutorial:

- Section B.2, “Initial Analysis”
- Section B.3, “Problem Analysis”
- Section B.4, “Detailed Dependency Analysis”
- Section B.5, “Advanced Analysis With Scripts”
- Section B.6, “Share Results”

Appendix E. Sonargraph Script API Documentation

Script API is documented via JavaDoc that is available within the help system of the application and can be accessed using a standard browser. Different packages exist for the language-independent core functionality and language specific parts.

Link to JavaDoc of Sonargraph Script API.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W