

# STAN

## Structure Analysis for Java



White Paper

Summer 2008

**Abstract:** *This paper gives a brief introduction to structure analysis using STAN, a static code analysis tool bringing together Java development and quality assurance in a natural way.*

*STAN encourages developers and project managers in visualizing their design, understanding code, measuring quality and reporting design flaws.*

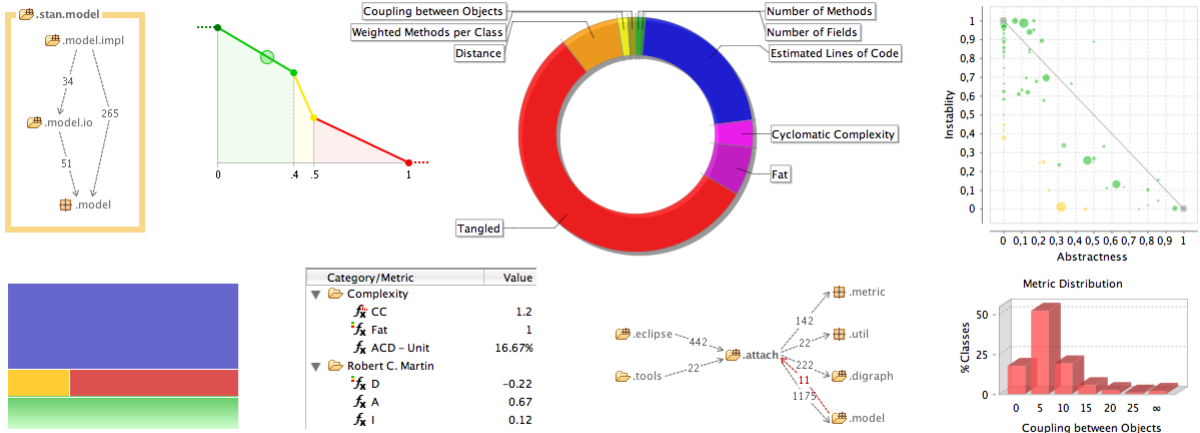
*STAN supports a set of carefully selected metrics, suitable to cover the most important aspects of structural quality. Special focus has been set on visual dependency analysis, a key to structure analysis.*

*STAN seamlessly integrates into the development process. That way, developers take care about design quality right from the start. Project managers use STAN as a monitoring and reporting tool.*



# Table of Contents

- Introduction.....3
  - Software Structure.....3
  - Rotten versus Good Design.....3
  - Fighting Complexity.....4
  - What is Structure Analysis?.....4
- Feature Survey.....5
  - Code Analysis.....5
  - Dependencies.....5
    - Composition.....7
    - Couplings.....8
    - Other Views.....8
  - Metrics.....9
    - Ratings.....9
    - Violations.....10
    - Queries.....11
  - Reports.....11
  - Eclipse Integration.....12
- Advanced Topics.....13
  - Acyclic Dependencies Principle.....13
    - Cycles.....13
    - Tangles.....13
  - Stable Abstractions Principle.....14
    - Main Sequence.....15
    - Distance.....15
- Conclusion.....16



## Introduction

As a matter of fact, many projects fail due to lack of **software quality**. Therefore, having an eye on code quality is not just an option, but may well prove mission critical. Furthermore, it's not appropriate to push quality control to the end of the software engineering process. Since early detection of quality issues makes them easy to resolve, monitoring quality should be placed as close as possible to the developer.

Over the past few years we observe that more and more companies recognize the importance of software quality. As part of this trend, testing has been widely accepted as an integral part of development. However, testing alone is only one step to go on our way to high quality software products. Equally important, we need to continuously validate our software against well established design principles, helping us to improve maintainability, flexibility as well as – testability.

When looking for solutions to ease development, control design and improve quality, it is worth considering the integration of **structure analysis** into the software engineering process. This is where STAN, a new structure analysis tool for Java, comes into play. STAN puts quality assurance to the hand of the developer, thereby achieving these goals with minimal effort.

## Software Structure

**Artifacts** are the things that make up a code base. For example, in Java, methods and fields are the building blocks for classes. Classes are organized into packages and packages are bundled into libraries. Finally, a set of libraries makes up an application. Members and classes lie on the **code layer**, whereas packages and libraries lie on the **design layer**.

**Software Structure** is understood as the way

1. how artifacts build into higher level artifacts
2. how artifacts depend on each other.

During development, the structure constantly changes. E. g., a new class is placed into a particular package or a new method adds dependencies to other classes and packages.

Structure is not just something hidden in the background. Structure reflects our design. Structure *is* our design!

## Rotten versus Good Design

As long as a project is small, developers have a vital image of their design in mind. They know every corner of their code and everything seems to be under control. As the project size evolves, however, things change: suddenly the software is hard to test, extend and maintain. It tends to be monolithic and somehow everything seems to depend on everything else.

Robert C. Martin describes this as *“The software starts to rot like a piece of bad meat”*.

Moreover, he identifies the following odors (among others):

- *Rigidity* – The system is hard to change because every change forces many other changes.
- *Fragility* – Changes cause the system to break in conceptually unrelated places.
- *Immobility* – It's hard to disentangle the system into reusable components.
- *Viscosity* – Doing things right is harder than doing things wrong.
- *Opacity* – It is hard to read and understand. It does not express its intent well.

To the contrary, a good design turns out to be flexible, solid, mobile, fluid and transparent.

## Fighting Complexity

The structure of large code bases tends to become very complex. Over-complex systems are hard to understand and maintain and thus do often break. Keeping complexity on a manageable level is a challenge.

We certainly must fail if we let our software structure evolve arbitrarily. It is therefore essential to keep an eye on it. Fortunately, there are common design principles which – if we follow them – can help us to succeed. We need to continuously validate our structure against these principles. And we need to discover and fix design violations early, before our software starts to rot!

## What is Structure Analysis?

Structure analysis is more than just measuring metrics and listing threshold violations. Structure analysis is about breaking down the system's complexity and letting the user inspect its artifacts at any level, from different perspectives.

An important aspect is **visualization**. A picture is worth a thousand words! A good structure analysis tool presents and visualizes the structural design in a way that is easy to understand for humans. If you need to learn the tool's language, there's something wrong.

When it comes to **dependencies**, a good structure analysis tool needs advanced graph layout capabilities to create neatly arranged dependency graphs. The user should be able to navigate through these graphs, to drill down, and so on.

A good structure analysis tool should support “modern” **metrics**. For each artifact, metric violations should be listed and ranked, allowing the user to distinguish important issues from negligible ones.

Not forget to mention, a good structure analysis tool should be **easy to handle** and useful from the beginning. It will be a natural part of daily development, because it's fun!

So far we focused on structure analysis as a development task. From time to time, it is also desirable to measure the overall quality of our structural design. Project managers need **reporting** functionality at the touch of a button.

Reports should contain meaningful information about the structural quality of a code base. Of course, metric violations should be listed. However, reports may also include visualizations of certain design aspects or selected design violations. Reports also provide an easy way to trace structural quality over time.

## Feature Survey

STAN provides you with structure analysis for Java. STAN analyzes byte code rather than source code. You don't need the sources and you can directly analyze any compiled Java code, whether it's yours or not.

STAN is available in two **variants**:

1. as a standalone application for Windows and Mac OS,
2. as an extension to the Eclipse Integrated Development Environment (IDE)

The standalone application is targeted to architects and project managers who are typically not using the IDE.

The Eclipse extension integrates STAN seamlessly into the IDE and allows the developer to quickly explore the structure for any bunch of code she desires.

## Code Analysis

While analyzing Java byte code, STAN collects all the information needed to build a detailed model of the application's structure. The code base is determined by choosing Java archive (JAR) files and class folders. **Filter Patterns** may be provided to include or exclude specific parts, e. g. to ignore test classes.

The **Level of Detail** specifies if our model shall include the member layer with all the classes' fields and methods or if it shall be limited to the class layer and above.

We have already mentioned Java's package concept as the basic way for grouping classes into higher level units. However, the package structure also builds a tree. For example, the packages `com.stan4j.db` and `com.stan4j.ui` are sub-packages of package `com.stan4j`. To take this into account, STAN allows you to toggle between the **Flat Packages** and **Package Tree** modes.

As another point, you might want to look at your application as one big code base or you might want to pay respect to the library layer. By switching to **Show Libraries** mode, you can inspect each library on its own as well as how the libraries are related to each other.

## Dependencies

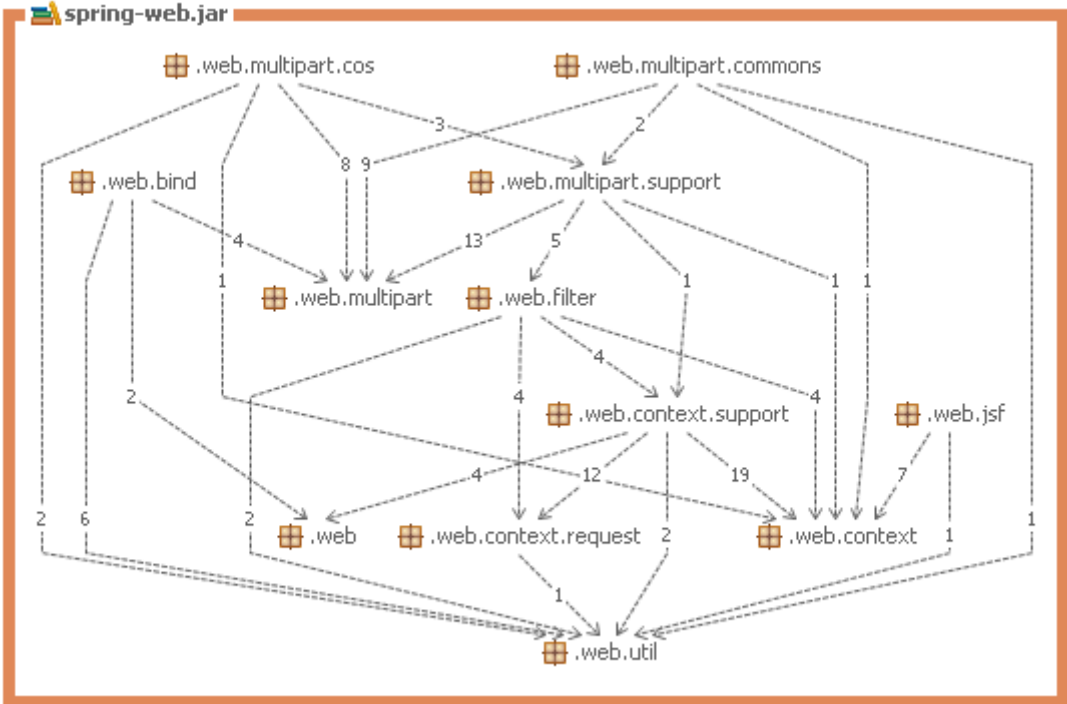
When dealing with software structure, dependency analysis becomes a central topic. For non-trivial applications, it's almost impossible to understand or even control how classes, packages and package trees interact with each other by just examining source code.

However, understanding how things depend on each other is crucial for making the right design decisions during development. We cannot expect to have a good design if we don't know it!

What we really want is to look at our artifacts, see what they need and what needs them. Given this, we can make better decisions and will end up with a better design. Sometimes we discover the need for refactoring to improve our design. Sometimes we just want to discuss some design issues. Instead of stumbling through the code, isn't it much better to look at a dependency graph?

Unfortunately, it's impossible to look at everything at once. To get useful graphs, we have to carefully select the perspectives and scopes. Otherwise we'll end up with very big and clumsy graphs. For example, we may want to look at an artifact to see how its contained artifacts interact or how the artifact itself interacts with the rest of the application.

Another issue is graph layout. Without sophisticated layout capabilities, even rather small graphs cannot be put into a picture we want to look at. At the same time, graph layout is a complex task and much effort has been put into STAN's graph layout engine to obtain good and fast results.



STAN shows dependency graphs for all levels of abstraction. Generally, nodes denote artifacts and edges denote dependencies. An edge's weight reflects the dependency's strength, which is the number of underlying code dependencies.



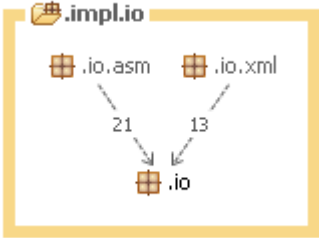
Selecting an edge will show you those dependencies.

Source	-->	Target
.MetricRators	contains	.MetricRator
.MetricRators.1	implements	.MetricRator
.MetricRators.1.category(Double)	returns	.MetricRator.Category
.MetricRators.NULL_RATOR	is of type	.MetricRator
.MetricRators.addRator(MetricRator)	calls	.MetricRator.getDomain()
.MetricRators.addRator(MetricRator)	has param	.MetricRator
.MetricRators.getRator(MetricDomain)	returns	.MetricRator
.MetricRators.map	references	.MetricRator

Graphs may be zoomed, narrowed and their orientation may be flipped to get optimal insight into the dependency relations of the artifact under consideration.

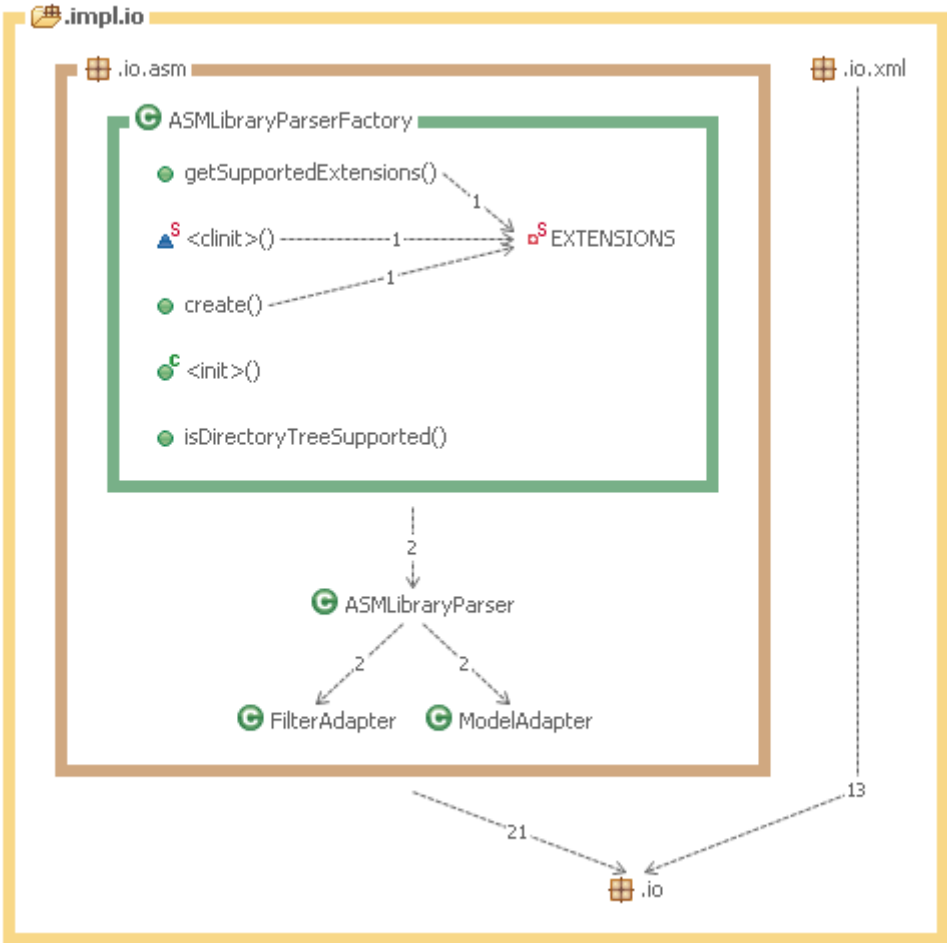
### Composition

The **Composition View** allows to *look into* the selected artifact, to see all its contained children and the dependencies between them. You may investigate dependencies between members of a class, classes of a package, packages, children of a package tree and between libraries.



STAN's layout engine guarantees, that – as far as possible – edges point into the same direction, either top to bottom or left to right, depending on the chosen layout orientation.

Artifacts can be expanded to arbitrary depth to dive deeper into the structure.



Additionally you may navigate into any of the displayed artifacts as well as up and down the artifact hierarchy.

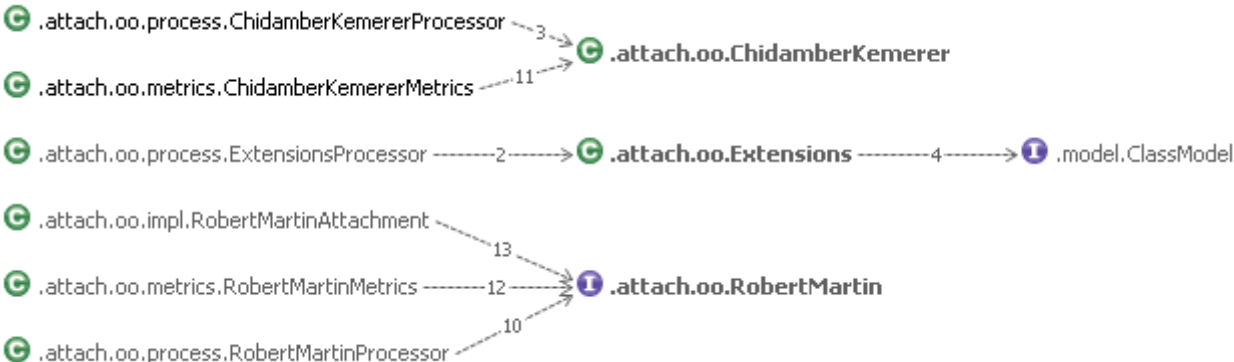
### Couplings

The **Couplings View** allows to *look around* the selected artifact, to see all its incoming and outgoing dependencies. You may investigate dependencies to and from classes, packages, package trees or libraries.

In addition to the direct dependencies one can optionally make visible intermediate dependencies between the displayed artifacts.



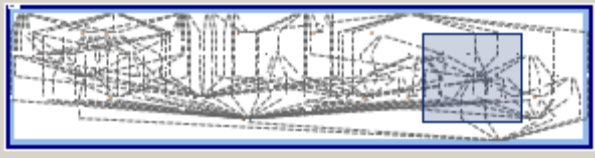
Here, *artifacts can be split* to explore how its children contribute to the shown dependencies.



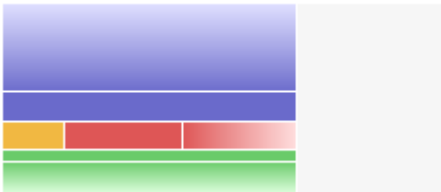
### Other Views

If the graph is too large to fit in your view, the dependency graph overview simplifies navigation.

- The **Graph Thumbnail** lets you choose the shown region visually.
- The **Graph Contents** allows you to reveal a particular node by selecting it from a list.



The **Dependency Landscape** serves as a quick indicator for an artifact's dependency relations. The shown area represents the entire code base. Generally, the code represented by each of the colored blocks in the diagram depends on the code of the blocks located below it and has bi-directional dependencies to blocks on the same level.



By selecting an arbitrary artifact *A*, the code base is partitioned into five categories: *A* itself (amber), artifacts tangled with *A* (red), artifacts required by *A* (green), artifacts depending on *A* (blue), and artifacts unrelated to *A* (gray). For the red, green and blue areas the diagram makes a distinction between direct dependencies (solid filled areas next to *A*) and indirect dependencies (fading areas).

## Metrics

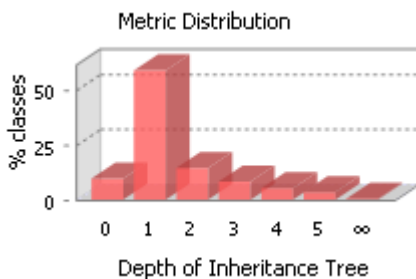
To be able to value certain aspects of structural quality, STAN also computes several metrics. A metric is simply a function mapping artifacts of some kind to numbers. Metric computation can be as simple as counting the number of classes in a package or as complex as determining the average component dependency in the package dependency graph.

STAN's aim is not to be a mass metrics tool. There are hundreds of metrics that could be computed quite easily, but who really wants to see them all?

STAN currently supports

- Several counting metrics
- Estimated Lines of Code
- McCabe's Cyclomatic Complexity
- Average Component Dependency, Fat and Tangled
- Metrics by Robert C. Martin
- Metrics by Chidamber & Kemerer

Metrics are collected into categories. A whole category at once as well as individual metrics can be enabled or disabled.



Where it makes sense, STAN promotes metric **averages** and **distributions** from lower level to higher level artifacts.

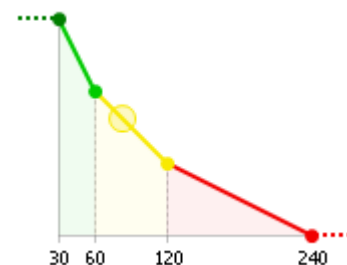
Selecting such a metric on a higher level artifact will show the distribution as a bar chart.

Category/Metric	Value
Count	
Units	10
Classes	12
Classes / Class	0.08
Methods / Class	3.58
Fields / Class	1.08
ELOC	238
ELOC / Unit	23.8
Complexity	
CC	1.28
Fat	14
ACD - Unit	33.33%
Robert C. Martin	
D	-0.18
A	0.6
I	0.22
Ca	25
Ce	7
Chidamber & Kemerer	
WMC	4.58
DIT	0.83
NOC	0.25
CBO	3
RFC	4.92
LCOM	0.58

## Ratings

When it comes to assessment, metric thresholds are needed to let the user define the boundaries between acceptable and unacceptable metric values. However, a single threshold may not provide us with the granularity desired for certain metrics.

For example, we might want to say, a value for a method's lines of code up to 30 is perfect, up to 60 is still fine, up to 120 is critical, above 120 is really bad and 240 is the worst we can imagine.



STAN provides what we call **Traffic Light Ratings**, partitioning the value range into green, amber and red subranges. Back to our example, 70 lines of code is better than 110. So, even if both values fall

into the amber range, they will be rated differently.

STAN comes with a reasonable set of default ratings. However, ratings can be added, removed and adjusted easily. Furthermore, the current settings can be exported and imported. This allows to **share preferences** with others or to switch between **multiple profiles**.

### Violations

An artifact is said to violate a metric if it is rated amber or red for that metric. For each artifact, STAN shows you all metrics that are violated by the artifact itself or by contained artifacts.

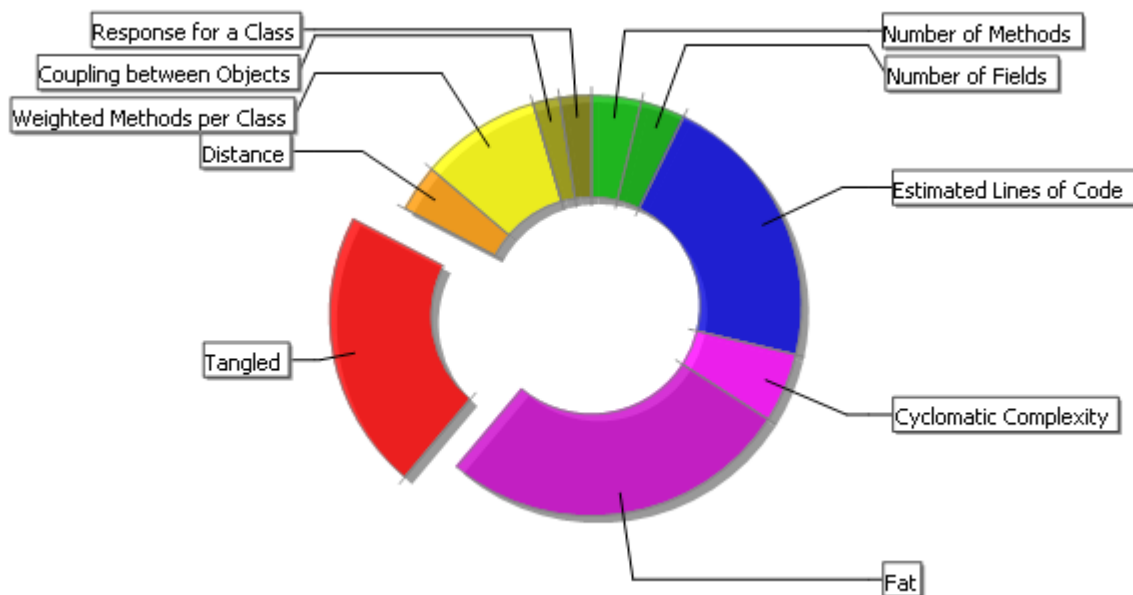
To take further profit of our ratings, we can use them to **rank** metric violations. However, simply sorting by rating isn't good enough. What we need is a measure for **relevance**.

STAN assumes that violations at “big” artifacts are worse than violations at “small” ones: it should be more relevant if a package received a bad rating for some metric *A* than if one of its 42 classes received a similar rating for some metric *B*. Moreover, even if a package is rated amber for *A*, this might be more relevant than if one of its classes is rated red for *B*.

To take this into account, STAN prioritizes a metric violation by weighting its rating with the amount of the artifact's underlying code. The result is shown in the **Violations View**.

Artifact	Metric	Value	Rating	Actions
.conf.impl.RootConfigImplParser	ELOC	334		
.conf.impl.RootConfigImplParser.addBaseRuleInstances(Digester)	CC	27		
.conf.impl.ConfigImpl	Fat	63		

Finally, for a given artifact, we might want to get a feeling for how metrics contribute to its violations and how badly the artifact is polluted by violations. STAN's **Pollution Chart** shows us exactly this.

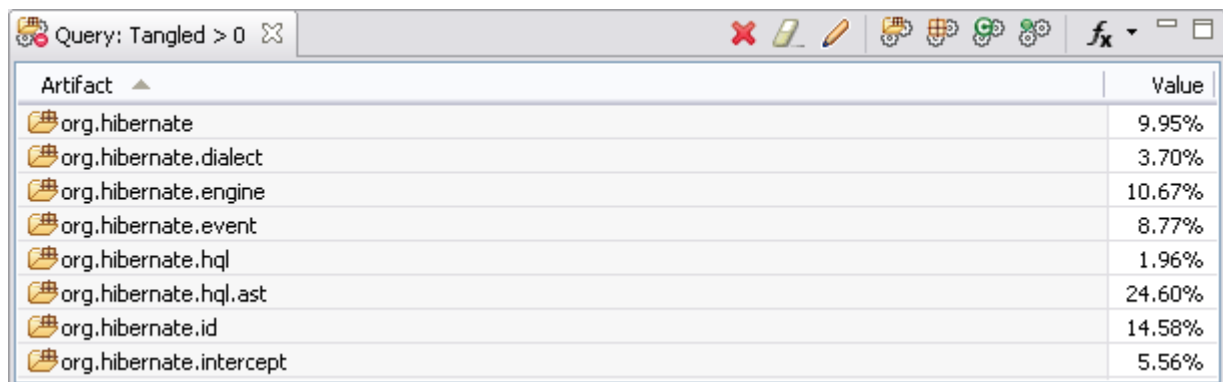


Selecting a slice as shown above filters the Violations View to the corresponding metric.

The wider the ring, the higher the degree of pollution of the underlying code. On the application level, the ring's thickness can serve as a quick indicator for the overall structural quality of the code base.

## Queries

Metric queries allow to track down artifacts that exceed some metric **threshold** or rating category. You'll find it convenient to have these artifacts at one place, to browse and investigate them, without losing scope. For example, the query *"Tangled > 0"* will give all the package trees with cyclic dependencies between their children:



The screenshot shows a window titled 'Query: Tangled > 0'. It contains a table with two columns: 'Artifact' and 'Value'. The table lists several packages from the org.hibernate namespace and their corresponding percentage values.

Artifact	Value
org.hibernate	9.95%
org.hibernate.dialect	3.70%
org.hibernate.engine	10.67%
org.hibernate.event	8.77%
org.hibernate.hql	1.96%
org.hibernate.hql.ast	24.60%
org.hibernate.id	14.58%
org.hibernate.intercept	5.56%

Double clicking a row will make the corresponding package tree the current artifact, thus showing the tangled graph in the Composition View.

Queries may be defined for any supported metric by specifying a threshold constant (as in the example above). Additionally, queries for rated metrics may be defined to match certain rating categories, i.e. "amber or red" or "red", thus collecting all artifacts violating the given metric.

## Reports

All gone after closing STAN? No! STAN generates **customizable** reports, ranging from giving a brief overview to detailed lists of metric violations. STAN reports currently may contain some or all of the following sections:

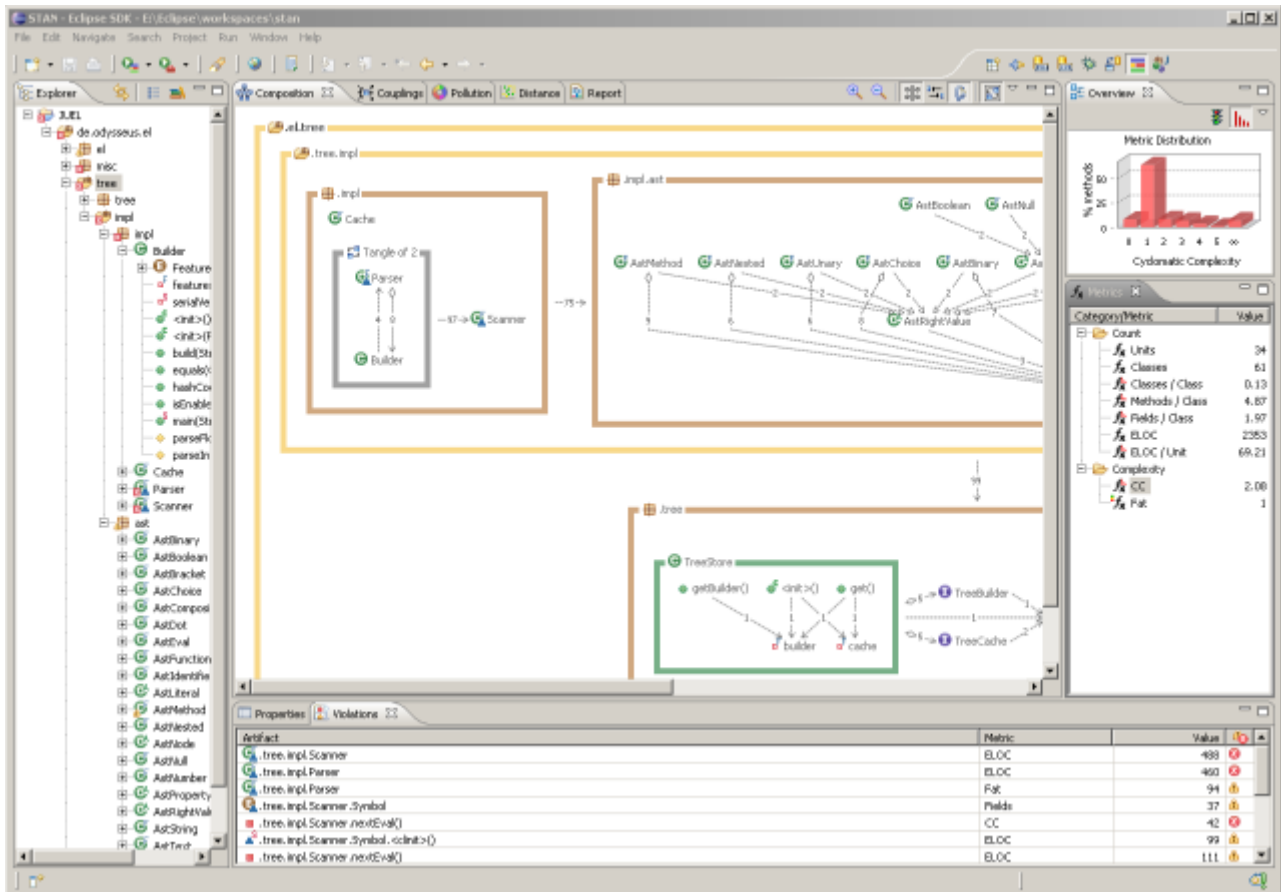
- A library dependency graph image
- A metrics summary for the application
- Top 10 ranked metric violations
- All metric violations grouped by metric
- Dependency graph images of package tree as well as flat package tangles
- Chart for Robert C. Martin's Distance (D) metric

Reports capture the structural quality of a project at a certain point of time. Periodically generating reports can help to **trace quality** and discover bad trends early.

STAN provides an **Ant task** which can be used to generate reports from an Ant script, without running the UI. This enables for generation of quality reports as part of the build process.

## Eclipse Integration

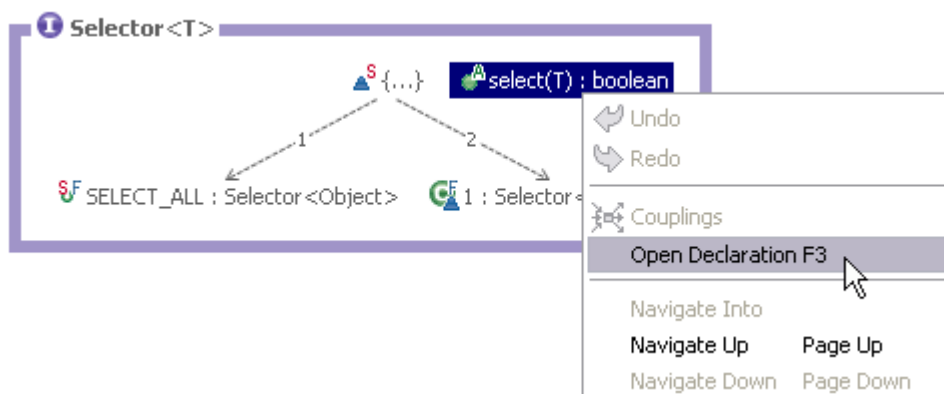
As stated earlier, STAN seamlessly integrates into Eclipse, the leading platform for Java development. Here's a screen shot of the **Structure Analysis Perspective**:



Running an analysis is as simple as selecting Java elements in Eclipse' Package Explorer, opening the context menu and choosing **“Run as... Structure Analysis”**.

Launch configurations are stored for later use and may be adjusted to your needs in the standard Eclipse Run Dialog.

Even if STAN parses Java byte code, navigation from code layer artifacts to source code is supported: where appropriate, STAN provides **“Open Declaration”** items in context menus.



## Advanced Topics

Now that we covered the very basics of structure analysis with STAN, let's dive a little deeper and learn about some common design principles, their associated metrics and visualizations.

### Acyclic Dependencies Principle

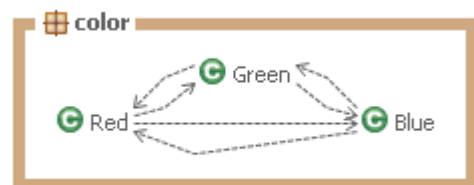
Cycled components can only be used together. They can only be tested, reused, deployed and understood together. The bad thing with cycles is that every node on a cycle depends on any other. Having lots of cycles lets explode the number of indirect dependencies within the system. Without early intervention, the system starts to rot.

We should therefore avoid dependency cycles within the design layer. That is, no library, package or package tree dependency cycles! This is known as the **Acyclic Dependencies Principle**.

### Cycles

Let's assume we have cyclic dependencies and want to remove them. Therefore, we want to do some refactoring that modifies our dependency graph to be acyclic. A naive approach would be to list all cycles, break them, one by one, until no cycles are left. Breaking a cycle could be done by removing or reversing a dependency. Let's see how that works.

Consider a graph with all pairs of nodes connected by edges into both directions. If we have two nodes, there's exactly one cycle. Adding a third node, we get five cycles: three cycles between pairs of nodes and two cycles containing all three nodes, clockwise and counterclockwise. Now let's increase the number of nodes to – say – ten. Guess how many cycles we get? More than a million! Seems like we better abandon the idea of fiddling around with individual cycles...



### Tangles

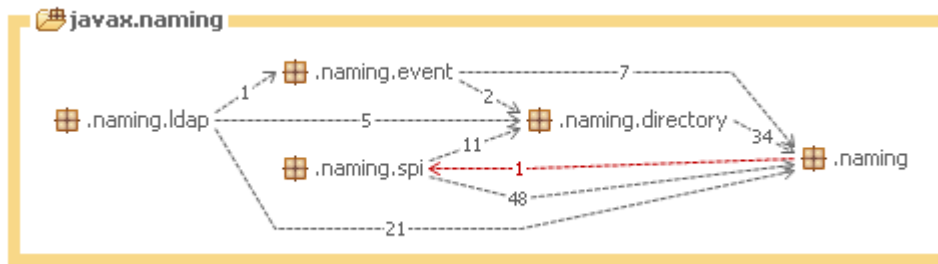
A **Tangle** is a subgraph with at least two nodes, where each node is reachable from each other. It is a tangle, where our cycles live. Every cycle lies in a tangle and every tangle consists of just cycles. In other words: A graph is acyclic if and only if it has no tangles.

Instead of breaking individual cycles, we could try to break tangles! Breaking a tangle means to transform it into an acyclic graph.

However, the edges in STAN's dependency graphs are weighted with the number of underlying code dependencies. Obviously, it is easier to remove or reverse a light edge than a heavy one. Therefore, we should select a minimum weight set of edges to break our tangle.

In graph theory, this is known as a **Minimum Feedback (Arc) Set**. The minimum feedback set is the “predetermined breaking point” of a tangle. Or, from another point of view, the minimum feedback set contains the edges, that point into the “wrong” direction. Feedback edges are the primary key to the elimination of cyclic dependencies.

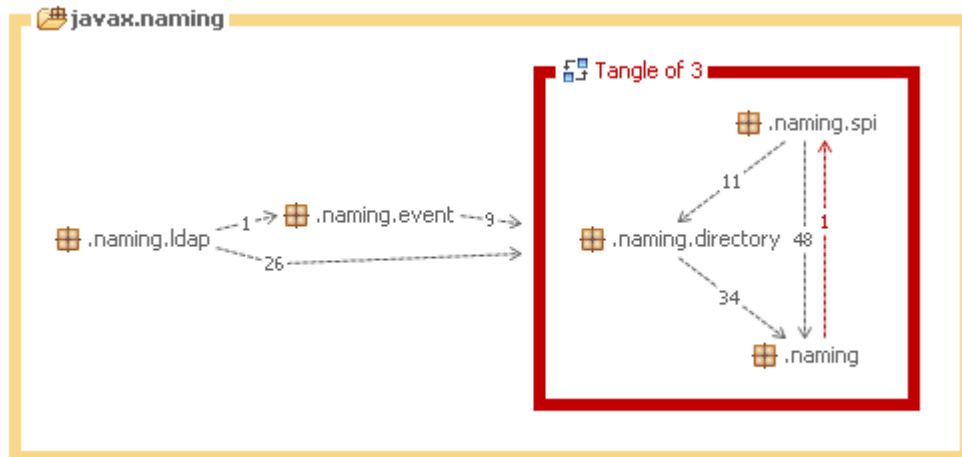
STAN's graph layout algorithm takes this into account: in a dependency graph, edges from the



minimum feedback set point into the opposite direction than other edges. Additionally, for design tangles, these edges are colored red.

When looking at a tangled graph, it's often hard to identify the boundaries of a tangle. Its nodes may be spread over the graph, so it's sometimes difficult to see which nodes and edges make up the tangle.

With STAN, you can partition dependency graphs into tangles. This isolates tangles by making them compound nodes. Note that the partition graph itself is acyclic: all cycles have been moved into the tangle nodes. This presentation is optimal to focus on cyclic dependencies.



As a side effect, it also reduces the complexity of the graph, because edges between nodes inside and outside of a tangle have been cumulated, now connecting the whole tangle with the outside world.

The Acyclic Dependencies Principle is reflected by the **Tangled** metric, which is calculated as the ratio between the weight of the minimum feedback edges and the total weight of all edges in the graph. Thus, values greater zero indicate cyclic dependencies.

### Stable Abstractions Principle

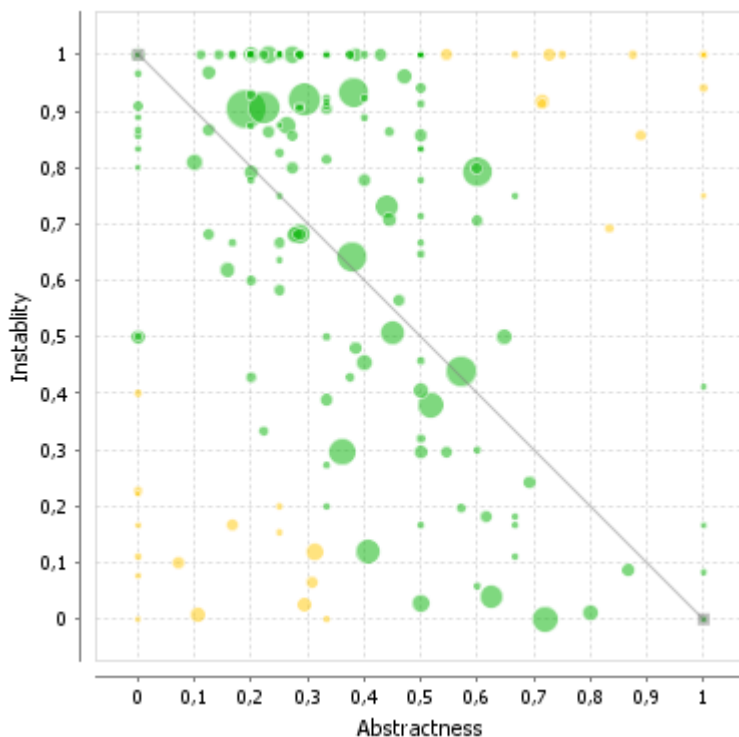
Robert C. Martin proposed the idea that for well designed software there should be a specific relationship between two package measures: the abstractness of a package, which shall express the portion of contained abstract types, and its stability, which indicates whether the package is mainly used by other artifacts (stable) or if it mainly depends on other artifacts (instable). The desired relationship is captured in the **Stable Abstractions Principle**: *A package should be as abstract as it is stable.* By sticking to this principle we avoid getting packages which are used heavily by the rest of the application and which, at the same time, have a low degree of abstraction. Such packages are a constant source of trouble, since they are hard to change or extend.

## Main Sequence

Let's get a little bit more detailed now.

- The **Abstractness**  $A$  for package  $P$  is calculated as the ratio of the number of abstract types contained in  $P$  to the total number of types in  $P$ . Thus, the resulting values range from zero (only concrete classes) to one (only interfaces and abstract classes).
- The **Instability**  $I$  for package  $P$  is calculated as the ratio between the number of classes outside  $P$  required by  $P$  and the total number of classes outside  $P$  related to  $P$ . As above, the resulting values range from zero (only incoming dependencies) to one (only outgoing dependencies).

What was this good for? Well, given these two metrics, the Abstractness and the Instability, we can place every package of our application in a diagram which shows the unit square with the



Abstractness on the horizontal axis and the Instability on the vertical axis. Furthermore, we can restate the Stable Abstractions Principle in graphical terms: packages should not lie too far away from the falling diagonal of the diagram, which is called the **Main Sequence**. This means that packages with a low degree of Instability should have a high degree of Abstractness and vice versa.

The packages which reside in the corners apart from the Main Sequence show specific problems: the lower left corner is called the *Zone Of Pain*, since its inhabitants are stable and concrete, thereby contravening the Stable Abstractions Principle, whereas the upper right corner is called the *Zone Of*

*Uselessness* containing packages that are highly abstract and that nobody depends upon. These are the two areas of the diagram which should be avoided.

## Distance

So now we've got everything together to define the **Distance**  $D$ , which indicates how far a package is away from the Main Sequence:

$$D = A + I - 1$$

Calculating the Distance this way, we get values between -1 and 1. A zero value means the package lies exactly on the Main Sequence, the sign indicates if the package is located above or

below the Main Sequence. A derived metric, the Absolute Distance (IDI), omits the sign, thereby allowing to compute meaningful average values for higher level artifacts.

STAN's **Distance Chart** shows you where your packages live, whether they are located near to the Main Sequence, as desired, or if they tend to drift to the bad corners. Every package is displayed by a bubble, the size of which is determined by the number of classes in the package. The color of the bubble reflects the rating of the package's Distance value, which is, as always, adjustable to your requirements.

## Conclusion

Integrating STAN into the development process has several clear benefits:

- Structure analysis helps in understanding code as well as keeping code understandable.
- Dependency analysis provides sophisticated design diagrams.
- Design flaws will be detected early, where it's still easy to fix them.
- Expressive metrics are supported to ensure compliance with design principles.
- Highly customizable metric ratings allow the creation of individual quality profiles.
- Ranking metric violations helps to focus on most relevant issues.
- Reports provide all the essential information at a quick glance.
- Seamless Eclipse integration makes structure analysis available where needed mostly.

All in all, utilizing STAN can improve the overall quality of your software products, thereby lowering costs, speeding up development and satisfying your customers.