
Controlling Architecture with Structure101

White Paper

July 2007

www.headwaysoftware.com

Contents

Executive Summary	3
Architecture diagrams.....	4
Layering and composition	4
Layering Overrides	4
Combining diagrams	5
Mapping to physical code	5
Creating architecture diagrams	6
Structure101 client.....	7
Structure101 IDE plug-in.....	7
Structure101 Web Application	8
Summary	9
Further Information	9

Executive Summary

Version 2 of structure101 focused on understanding how your code *is* structured today, and where and why it is tangled or overly-complex.

Version 3 lets you additionally define how the code *should* be structured and communicate this to the team so they can actually make it happen.

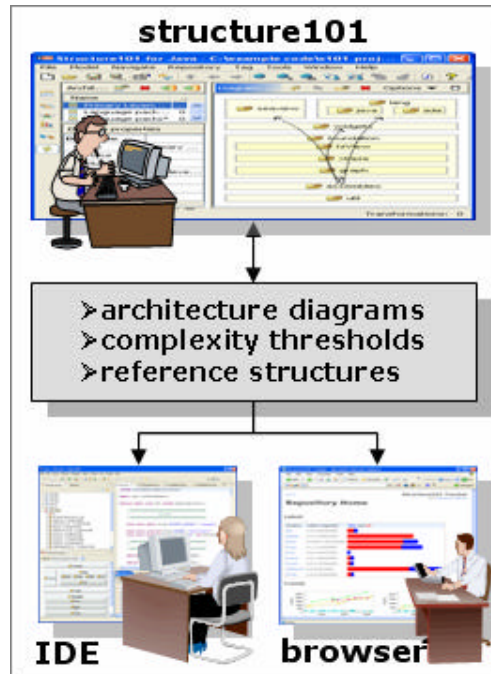
The structure101 client lets you create *architecture diagrams* from scratch, and/or automatically create them from the code. You can edit these interactively, map architectural elements to the physical code and discover where and why the code deviates.

Structure101 *IDE plug-in* makes architecture diagrams visible to each developer and warns them if they make code changes that are inconsistent with the defined architecture.

The structure101 *web application* makes the architecture diagrams visible in a web browser, and provides RSS alerts when new violations make it into the project mainline.

With the right information available when it is needed, each member of the team contributes to bringing the architecture back under control with little additional effort.

As the code-base structure and architecture improves, the development velocity increases, and more features make it into each iteration.



Architecture diagrams

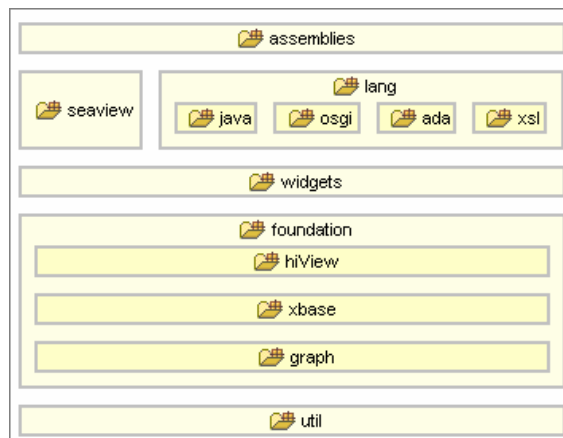
Often the physical structure of a code-base does not correspond to the way in which architects and developers think about the architecture. For example, the architecturally significant parts of the structure may be distributed throughout the physical structure. This can make it hard for architects to communicate the intended architectural constraints to developers in a way that encourages them to conform.

Layering and composition

Structure101 uses a concise visual notation for representing architectural layering and composition.

Here is an example of one of the architecture diagrams that we use for the structure101 code-base.

The principle is simple; components ("cells") should only depend on components at lower levels, not in the same or higher levels.



Layering Overrides

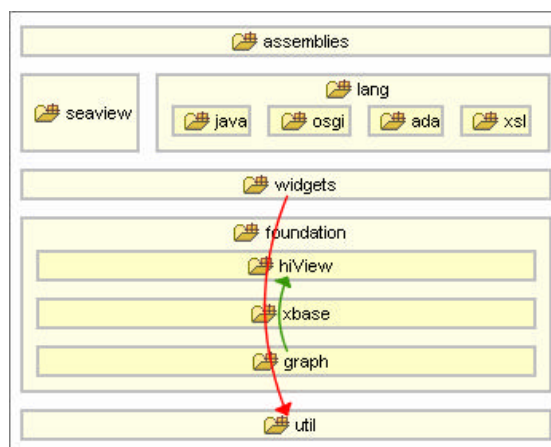
Sometimes a top-down dependency structure is too simple to capture the intent of an architecture.

"Overrides" allow you to override the default layering of a diagram.

For example we may decide to allow a specific dependency from a cell to a higher-level cell. The override is shown as a green ("allowed") arrow on the architecture diagram.

(Note that enabling this "upward" dependencies practically merges the "hiView", "xbase" and "graph" components from the perspective of testing, reuse, development, etc.)

A more common example is where we wish to enforce a more strict layering. For example we may want one layer to only use the next layer down, but not layers below that.

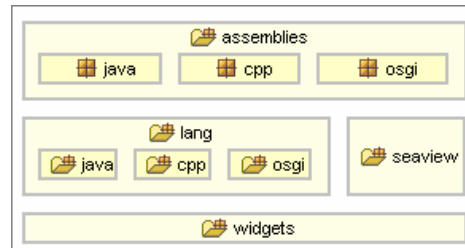


Such an override is shown as red (“disallowed”) on the architecture diagram.

Combining diagrams

It is not necessary to include all aspects of an architecture on a single structure101 architecture diagram.

A common scenario is where a number of “add-ins” are distributed across several packages. For example, this diagram shows part of the structure101 architecture.



It is correct, but incomplete. Classes in assemblies.X should never depend on classes in lang.Y. We could express this by adding several overrides, but it is much cleaner to use a separate diagram for this aspect of the architecture.

The next diagram defines a number of “language packs” that do not have a direct equivalent in the physical structure (they are “pure” architecture components), but express the architectural constraint that was missing above.



components), but express the architectural constraint that was missing above.

The combination of the 2 diagrams defines the intended architecture.

Mapping to physical code

In order to understand how a physical code-base conforms to an intended architecture, it is necessary to map the architectural components to physical code.

Simple patterns are used to establish this mapping.

This has a number of benefits:

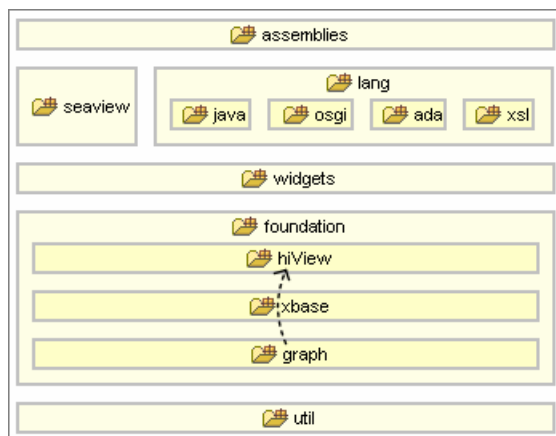
- If a diagram contains a component mapped to `com.headway.lang.*` and the team creates a new package `com.headway.lang.cobol`, then the diagram is not rendered obsolete – all the classes in the new package map to the intended component.
- You can create components with more complex mappings with expressions such as `com.headway.*.test.?`
- I can create and show a component for which no code has yet been created, either by specifying no pattern or specifying the paths where I expect the new code to be implemented.

- I can effectively “hide” physical entities from a diagram. For example any code in `com.headway.lang.cobol` will simply map to a component with the expression `com.headway.lang.*` - I do not need to show package `cobol` on the diagram if I don’t want to.

Another flexibility is that a physical entity maps to the component with the **most specific pattern**. For example if I include 2 components, one with `com.headway.lang.*` and the other with the expression `com.headway.lang.java.*`, then the class `com.headway.lang.java.myClass` will map to the latter. The effects of this can be at the same time subtle and powerful. For example I could move the component that maps to `com.headway.lang.java.*` into another “parent” altogether.

Finally, each diagram has a (possibly empty) expression that maps to “excluded” items. This is useful if some physical entities would otherwise undesirably map to a component in the diagram.

When a dependency is introduced that violates the architecture diagram, it is shown on the diagram as a curved dotted line as shown here between component “graph” and the higher-level package “hiView”.



It is easy to discover the code-level cause of a violation by selecting it on the diagram within a structure101 client or IDE plug-in.

Creating architecture diagrams

You can create a diagram from scratch or initialize one from the physical code.

An interactive editor lets you edit diagrams to adjust the layering, add new cells, hide cells, add overrides, etc.

When and how you create architectural diagrams will vary. For example if you are starting a new project, you can create a diagram (or set of diagrams) from scratch and map the cells to the expected implementation path, and then observe the code-base as the defined architecture is “filled-in”.


If you already have a substantial code-base, you can create a set of diagrams automatically from the code-base, to the appropriate level

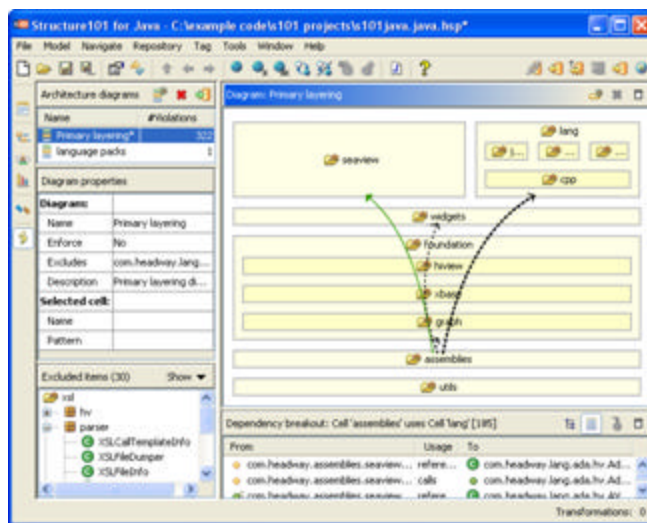
of detail. Using the editor, you can adjust the layering where the architecture should differ from the current as-built structure.


On an iterative project, you can edit the current architecture diagrams so that they represent the architecture at the end of the next iteration so as to guide the team during the iteration, and to give you a benchmark against which to monitor progress.

Structure101 client

An architecture perspective allows the creation and editing of the architecture diagrams associated with a project.

As well as starting with a blank diagram, you can initialize an architecture diagram from structural diagrams in the composition and slice perspectives (look for the  button). For example you could create a diagram from a design tangle in the slice perspective, and then edit it to define how the packages **should** be layered.



The architecture perspective is selected by selecting the  button on the vertical perspective tool bar on the top-left of the window.

Tagging is available as in other perspectives, and persists across perspectives. This is useful for analyzing the physical to logical mapping (tag in one perspective and then switch to another).

Once the diagrams are published to a project in a repository, any developers that have installed the IDE plug-in and linked it to that project will see them and start receiving warnings based on them.

Structure101 IDE plug-in

The structure101 IDE plug-in makes the project architecture diagrams visible to each developer and warns them if they make inconsistent code changes.

Once developers link the plug-in to the structure101 repository that contains the project architecture diagrams, they will always see the most recent updates within the IDE. They can immediately see any existing violations on the diagrams, and can navigate to the

Summary

Structure101 allows you to define concise, expressive architecture diagrams that map to the physical code.

You can define these up-front on a green-field project ; by auto-generating from an existing code-base and then editing; and/or edit the current architecture for the next major iteration.

Existing architecture violations are overlaid on the diagrams within the structure101 client, IDE plug-in and web application, and you can easily discover their origin.

Keeping the architectural model in a central location means that the team always works off the current version.

Warning developers immediately when they create or modify code in violation of the architecture prevents the structure degrading any further, and encourages gradual improvement.

The ability to differentiate recent violations keeps architectural compliance close to development activities and therefore manageable.

Having all the diagrams and derived information accessible by browser means that the whole team has instant access.

Being web-enabled, structure101 works equally well for localized or distributed development projects.

Using structure101 to communicate a target architecture to the whole team makes architectural control simple, practical and even enjoyable for the first time.

Further Information

Email: hwinfo@headwaysoftware.com

Web: www.headwaysoftware.com

Blog: <http://chris.headwaysoftware.com>