Structure101 Studio

For Structure101 and Restructure101 users

Structure101 Studio brings our 2 prior products – Structure101 and Restructure101 – together under one hood. This means more than just the convenience of having all that functionality in one place; it support really cool new workflow and use cases.

The power comes from the interaction between the Levelized Structure Map (LSM) (from Restructure101) and the Architecture Diagrams (from Structure101).

Using 2 products - Structure101 with Restructure101

The prior 2 products focus on 2 separate problems. First knocking your codebase into shape with the LSM in Restructure101, then keeping it in shape with Structure101 XS detection and Architecture diagrams. The interaction between the 2 products is via changes to the code-base.

Restructuring (in Restructure101) means simulating restructuring actions in the LSM until you have a model you are happy with, exporting the action list to IDE, and doing the refactoring work. The end result is that your code-base has the same structure as the one you modelled. Doing the work means implementing the new structure in some physical way, using packages, Maven projects, namespaces, assemblies, or whatever.

Once your code-base's physical structure reflects the modular hierarchy you want (perhaps because you used Restructure101 to get it to that point), you use Structure101 to keep it thus by defining Architecture Diagrams that at the same time communicate the intended structure to your team, and define constraints on dependencies and visibility so that violations are flagged at compile and build time.

(As well as letting you define what is subjectively your preferred architecture, Structure101 lets you control objective structural over-complexity measures, and gives you all kinds of useful graphical views into your code-base for general comprehension and analysis).

The implication of the code-base itself being the interface between the 2 products is that it is really necessary to physically change the code-base to reflect the Restucture101 model, in order to continue the modularization process into definition, communication, and enforcement with Structure101. The end result is very sweet, but the upfront commitment is big – you are potentially talking about touching nearly every file in your code-base in order to realize the full benefit of either product.

Using 1 product - Structure101 Studio

We have a great many customers that are very happy working in this way with the 2 products. But by joining up the LSM model with the Architecture diagrams model, it is now possible to construct a model

of your architecture which maps to your current, chaotically-organized codebase, but does not require the code-base to be physically aligned in order to gain most of the benefits you'd get if it was. In addition, any manipulations to your structural model in the LSM are immediately reflected in your Architecture diagrams, so you can now use the power of the LSM to define refactoring actions that repair violations to your defined architecture.

Another way to think of it is that we are treating the package structure as what it really is – a naming structure – which may or may not reflect an "architecture" (more often than not it doesn't). The goal is to organize your classes into a separate hierarchy of logical containers (call them whatever you want, modules, subsystems, buckets, …), define rules for how these containers can use each other, and spot where they don't. With Structure101 Studio you can get to this stage in a matter of hours, without changing a line of code. Next you can simulate and export a list of refactoring actions that will result in the code in the classes (the "implementation" level) conforming to your hierarchical model (the "logical" levels). This is a much smaller list (and is arguably the more important list to tackle first) than the Restructure101 list that included all the repackaging commands. These actions can be fed into your refactoring list as part of ongoing development.

At this point, you have defined and started convergence on a defined architecture with relatively low risk and upfront commitment. In a relatively short period you can have a defined architecture to which your implementation code conforms, and against which ongoing changes are verified. A big step forward. If your original package structure happens to be close to an architecture, you can use it as a start point for your model. If not, you can build your model from the bottom up, using class-to-class dependencies to identify cohesive clusters (embryonic modules). Structure101 Studio will even create a first cut at this automatically for you ("auto-levelize" command, context menu). Once you are happy with your model, and wish to align your package structure with your model, Structure101 Studio will give you the mapping of classes to containers which you can implement as rename class actions (in fact they are simple enough to script). But this step is no longer a prerequisite to a code-base that conforms to a defined architecture.

Understanding mappings and actions in Structure101 Studio

The main functionality happens on the ***Model*** and ***Rules*** tabs. You can create multiple Models and multiple Architecture Diagrams. A Model manifests as a Levelized Structure Map (LSM) (as in Restructure101). The Rules are expressed as Architecture Diagrams (as in Structure101). There is now some advanced capability that uses the interaction between the 2. It is still perfectly possible to work individually with models/LSM and/or rules/diagrams, but you do need to understand what is going on to use it right, or not use it unintentionally!

Models

In effect a Model uses an LSM to map all the classes in the project to one container in a containment hierarchy (which may or may not be derived from a physical structure such as packages or namespaces - to emphasise this "optional" association with any physical structure, you can even choose to "tools/show all containers as modules"). You can do all kinds of manipulations to create and edit this

mapping. For example you can initialize the structure using “auto-levelize”, move classes between containers, rename containers, etc.

Within a model you can also perform lower, implementation-level actions that do not reassign classes to new parents. For example you might move methods between classes, delete dependencies, etc.

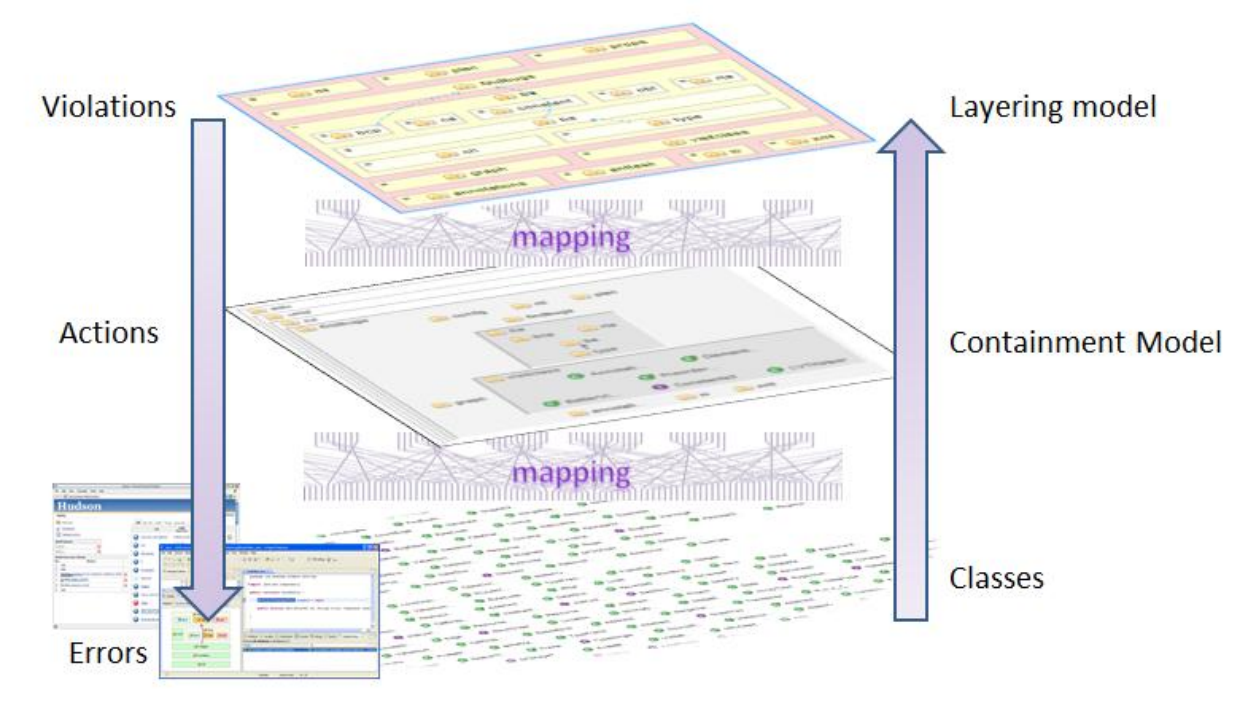
As in Restructure101, every action you make to the LSM will be added to the sequential Action list. The Action list is now complimented by 2 other lists – the Class map, and the Refactorings list. If you wish to align the physical code structure with the model, then you will export and apply the Class map. If you wish to enact code-level changes that improve the structure or repair rules violations, then you will separately export the Refactorings list. This separation allows you to maintain a containment model (and associated rules) without necessarily aligning the physical codebase structure with it. Of course you can also export the complete sequential action list which ignores this distinction.

Rules

The cells in Architecture Diagrams map to classes in the currently selected Model. When you create a diagram from a model, the patterns associated with each cell in the new diagram will assume the structure in the model at that time. If you subsequently switch to a different model, some of the patterns in the diagram may no longer match to classes. This can be extremely powerful, but you need to remember what is going on!

IDE plugin

In order for the 2-level mapping to work in the IDE, the “shared” model is applied within the IDE, before the patterns in any “enforced” diagrams are resolved against the classes in the local copy of the code. This means that you do need to “share” the right model. You will be prompted for the desired combination of Actions/refactorings/mappings and shared model at project publish time.


Violations
Layering model
mapping
Actions
Containment Model
mapping
Classes
Errors