HELLO2MORROW

# SonarJ White Paper

Sonar stands for Software and Architecture. It is meant to support software developers and architects in their daily work.

Software over its lifecycle needs to be changed, adapted, enhanced or extended. There are different main challenges dealing with software projects:

1. Managing the overall software developing process
2. Getting down to the real requirements
3. Building a healthy software basis

Point 1 and 2 are explored in broad depth. Point 3 is often underestimated in its importance, but finally the source code (or what is deployed) does the job and needs to be enhanced and changed many times over its lifecycle by different developers. To amortize the money invested into the development of a software system its lifetime should have an adequate length - the longer it runs the better. For software to be run over a long period of time it must be possible to test it effectively and make changes to it without enormous extra costs and risks. Moreover new developers should be able to understand the system within a reasonable time span.

If we want to achieve this goal we need to keep in mind the internal structure of the software (its architecture) and some other basis characteristics related to technical quality. A healthy software basis is needed to manage a development project successfully over a longer period in time. Documentation is required, but it dos not prevent a software system to perform poorly – the software itself does not care about documented features ("The software is the design" [ASD]). Getting the real requirements worked out, which in itself is already difficult and really important, won't help if technical quality is not. „Quality is terrible as a control variable. You can make very short-term gains (days or weeks) by deliberately sacrificing quality, but the cost – human, business, and technical – is enormous"[EXP].

The software is designed upfront with some basic ideas (a so called 'architecture vision') emerged from a team of architects. Everyone in the team more or less commits to this vision. Furthermore nobody in the development team would say "let us discard effective testability", "lets go and make the code overwhelmingly complex", "lets go and use 10 different xml parsers and logging frameworks" - no! Everyone would say "let's go and make our job easier by assuring a certain level of technical quality to cope with the real existing complexity – the implementation of what the end user needs". So everyone would naturally commit to the following statement: „*For software to be tested effectively, it must be designed from the start* with that goal in mind … Testability, like quality itself, cannot be an afterthought: it must be *considered from the start – before the first line of code is ever written* " [LSD].

So there are two main goals to achieve:

- How to avoid that the real structure of the software decouples itself from its planned architecture.
- How to assure a certain level of technical quality by enhancing understandability and testability, while controlling complexity.

This can be done by:

- Managing the overall dependency structure actively.
- Controlling complexity actively.

  ⇨ Integrating both activities directly into the development cycle

Sonar basically closes the gap between the specification of a logical architecture description in terms of layers, subsystems and vertical slices and the most important artifact: the source code. It helps to spot potential problems of technical quality related aspects. Sonar does this without dividing the team in the ones who check the achievement of the two main goals and the ones developing the software. Sonar is a lightweight Java-based tool that controls certain project guidelines during the developing phase. The analysis runs automatically every time the developer saves source code changes.

## *A word on actively managing dependencies*

The software should adhere to the allowed dependencies coming from a logical architecture description for a good reason and the software should avoid "unhealthy" dependency constructs to a certain level (at least to package level).

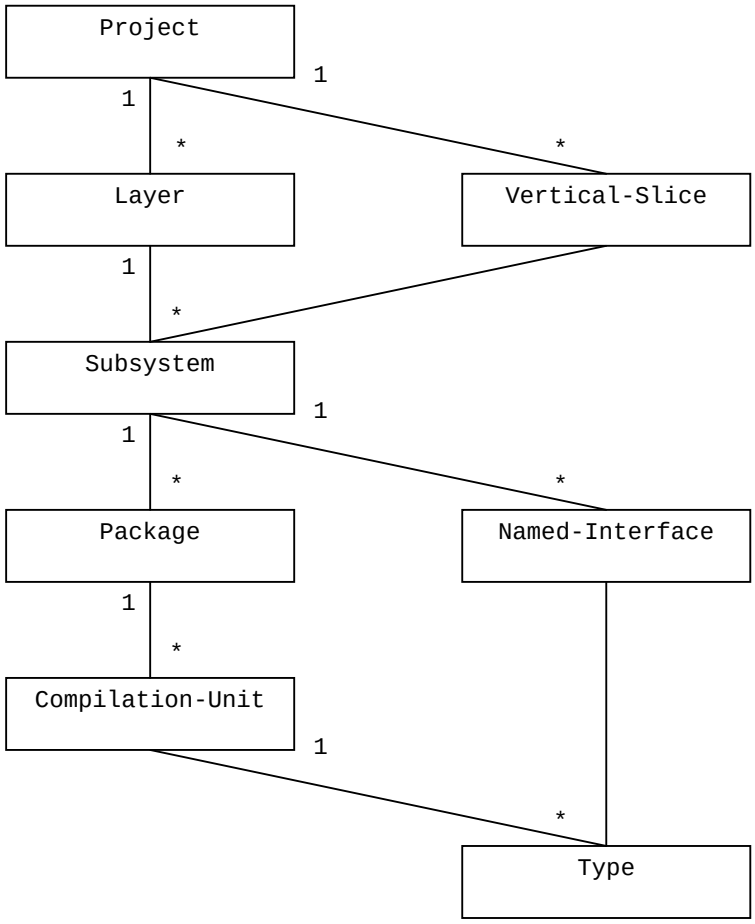
„Guideline: No Cycles between Packages. If a group of packages have cyclic dependency *then they may need to be treated as one larger package in terms of a release unit. This is undesirable because releasing larger packages (or package aggregates) increases the* likelihood of affecting something." [AUP]

"*The dependencies between packages must not form cycles.*" [ASD]

„Cyclic physical dependencies among components inhibit understanding, testing and reuse … *Every directed a-cyclic graph can be assigned unique level numbers; a graph with cycles* cannot … A physical dependency graph that can be assigned unique level numbers is said to be levelizable … In most real-*world situations, large designs must be levelizable if they are to* be tested effectively … Independent testing reduces part of the risk associated with software *integration* " [LSD]

## *Division of a project: The Software Architecture Model*

Sonar internally uses a model for the division of a project as mainly described in [AUP] (and many other sources). It is a widely adopted method of describing the overall software architecture (static view).



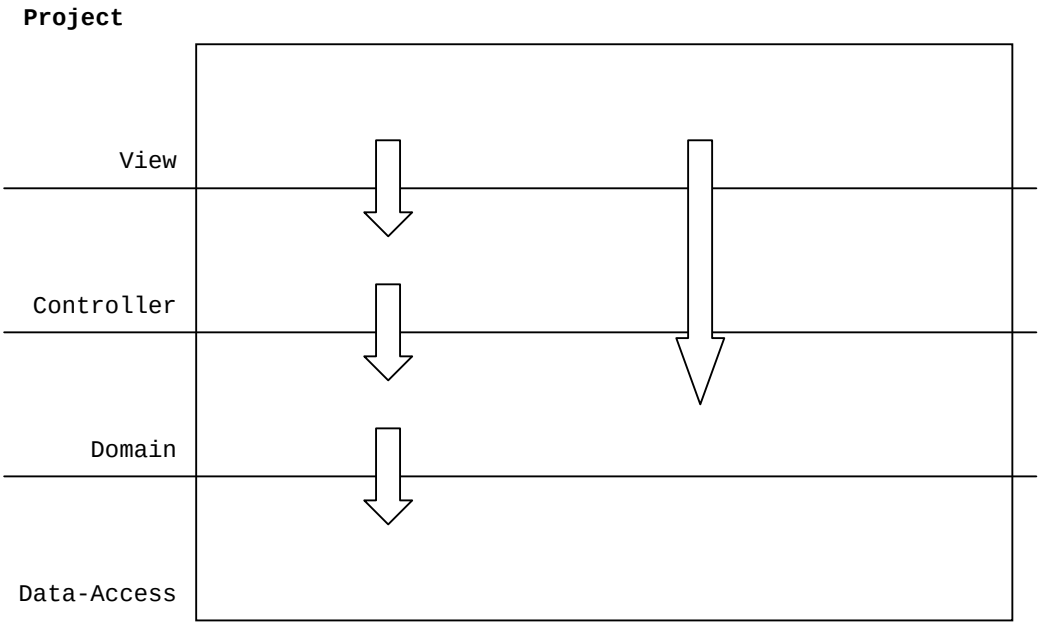
## *How to find the overall static structure for a project?*

### Step 1 : Divide horizontally

First of all you should develop an idea of the technical layers dividing your project horizontally and their interdependencies. Typical layers are:

- View
- Controller
- Domain
- Data-Access
- …

This could lead to the following overall picture:

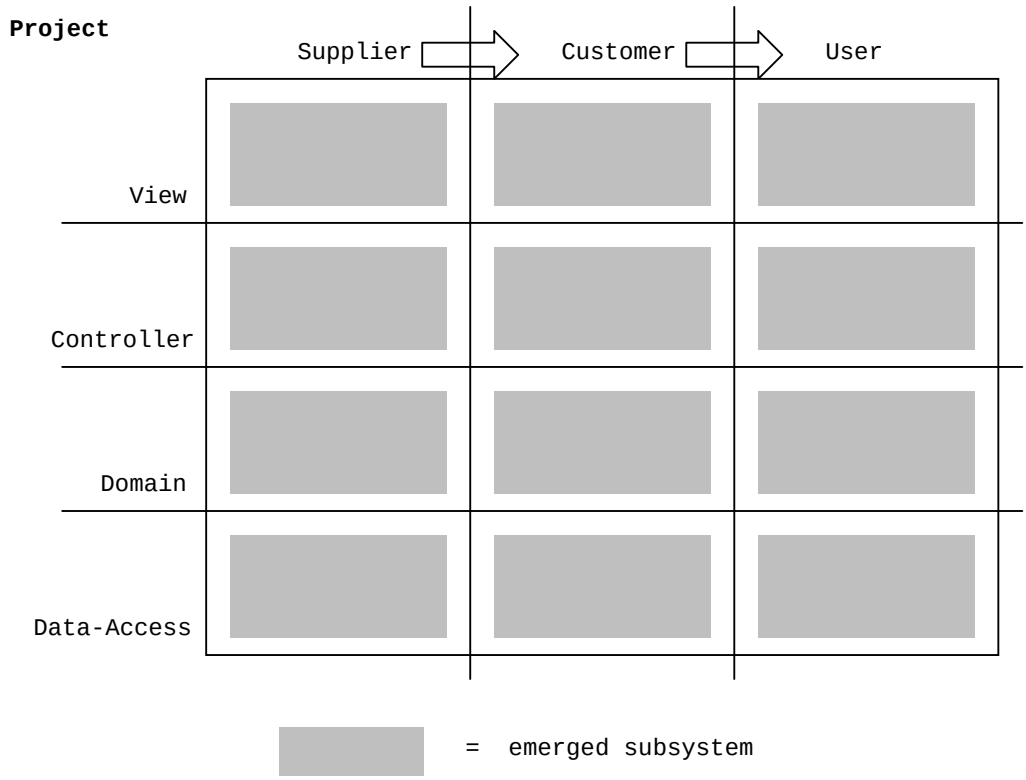



Note that this is a very simplified example that may not be appropriate for a real life project – may be it would be a good idea of inverting the uses dependency of the domain layer to the data-access layers (see discussion in [ASD]).

## Step 2 : Divide vertically

The vertical division is normally driven by business or application logic related aspects finding divisions like supplier, customer, user, … with suggested uses dependencies.

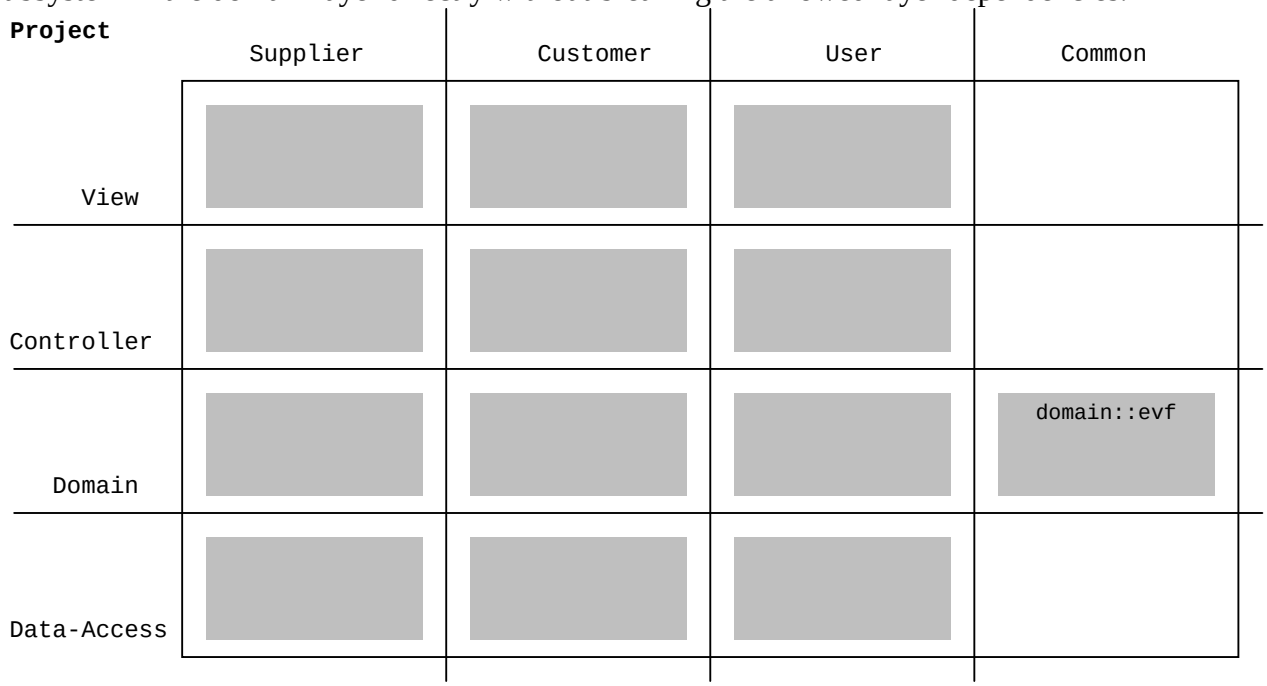
This could lead to the following picture:




Dividing the project vertically has led to a lot of business-logic-oriented subsystems.

## Step 3 : Add pure technical subsystems

Normally it is necessary to create subsystems dealing with pure technical aspects. In this example an event notification subsystem will be introduced to connect the view with the domain layer avoiding a direct cyclic coupling. It would be possible to introduce another layer for our first technical subsystem (maybe called foundation) but it is also possible to place the subsystem in the domain layer directly without breaking the allowed layer dependencies.




The subsystem is called `evf` (Event Framework) and it was added to a new vertical-slice common, indicating that it is a technical oriented vertical-slice. Its fully qualified name would be domain::evf, like all others would have such a fully qualified name (layer-name::vertical-slice-name).

## Step 4 : Define the mapping from physical packages to logical subsystems

It is a good idea to have a fixed mapping and naming scheme for packages assigned to subsystems (which packages implement a subsystem).

For example:
`[company-prefix].[vertical-slice-name].[layer-name]`
Every package starting with an identical prefix belongs to the same subsystem.

For example the following packages implement `domain::evf`

- `com.hello2morrow.application.common.domain.event`
- `com.hello2morrow.application.common.domain.event.generation`
- `com.hello2morrow.application.common.domain.event.dispatcher`

## The basic configuration idea

Sonar's basic configuration idea is to separate global project settings from individual views of developer teams.

The project description contains:

- Logical architecture description
- Mapping of packages to subsystems
- Assertion support settings
- Thresholds for complexity checks

The workspace description contains the relevant part a team or a single developer works on.

## Summary

Controlling and managing allowed dependencies between logical components of a software system is not just an option for medium and large projects – it is a precondition for success. This assumption is heavily underpinned by relevant literature (see references), research and practical experience.

Therefore you must have a strategy for dependency management, if you want to make a medium or large project successful without risking significant schedule or budget overruns.

To implement such a strategy, you need at least one person, which takes responsibility for the overall structure of the system. In many cases this responsibility will be taken by the leading software architect. But how can this person enforce a given architectural software structure?

One way would be to use some open source tools (e.g. JDepend) to analyze dependencies and compute some overall metrics. This approach will work to a certain degree, but requires a lot of manual effort. There are two problems with that approach:

- None of the open source tools known by us supports the definition of an overall logical architecture, i.e. the definition of legal and illegal dependencies between logical components.
- Therefore it requires a significant manual effort to find architecture violations. Moreover each violation must be communicated to the team member that created it.

If you check the market for commercial tools you will find solutions that allow the definition of a logical architecture to a certain degree. You will find support for subsystems and layers (nested subsystems inside of a "super subsystem"). Nevertheless some issues remain:

- Since vertical slices are not supported, the effort to create a logical architecture can be significantly higher.
- Still a detected architecture violation must be communicated to the person that created it. Therefore the software architect can easily find himself becoming a communication bottle neck.

- Currently we do not know any solution, that supports a light weight integration into the developer's IDE.

SonarJ offers an answer to all of these issues. It supports the incremental definition of a logical architecture. The architecture is stored in an XML file that can be easily distributed to the whole development team. Versioning and change management is supported by using standard solutions like CVS or Subversion. Developers stay within their well known development environment and are notified immediately if a change causes an architecture violation. In most cases no extra communication is needed to fix the problem.

SonarJ allows you to enforce a defined logical architecture even for very big Java projects. This major benefit comes without imposing any major additional effort to the architect or to the development team. Efficiency will be increased while communication overhead is minimized.

## *References*

[TOS] Testing Object-Oriented Systems, Beizer, Addison-Wesley 2000
[ASD] Agile Software Development, Robert C. Martin, Prentice Hall 2003
[LSD] Large-Scale C++ Software Design, John Lakos, Addison-Wesley 1996
[AUP] Applying UML And Patterns, Craig Larman, Prentice Hall 2002
[EXP] Extreme Programming, Kent Beck, Addison-Wesley 1999

## *More Infos:*

**US Office:**

hello2morrow Inc.
1050 Winter Street
Suite 1000
Waltham, MA 02451
+1 (781) 839-7310

**European Office:**

hello2morrow GmbH
Kastanienallee 2
D-82049 Pullach
+49-89-55 26 34 82

Email: info@hello2morrow.com
www.hello2morrow.com