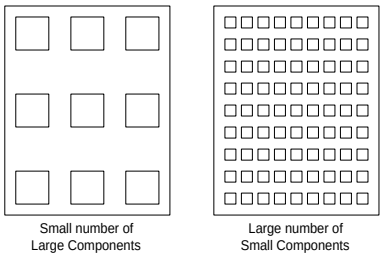
Aggregation Levels in Composable Architectures

-



Gerrit Muller
Buskerud University College
Frogs vei 41 P.O. Box 235, NO-3603 Kongsberg Norway
gaudisite@gmail.com

Abstract

The creation of a Product Family is an alternation of decomposition and synthesis steps. Composable architectures emphasize the composition of products, decreasing the importance of a priori decomposition. The products and intermediate compositions can be viewed as recursive aggregation levels. Careful trade-offs are required between the size of an aggregation level and the way it will be deployed. Flexibility and (configuration) manageability amongst others are balanced.
This article shows multiple viewpoints with respect to aggregation levels, the concerns per viewpoint and the relevant entities per viewpoint. For every viewpoint heuristics are given for the level of granularity.
This article is to be used in the "Family Engineering Handbook", a collective effort of Philips Research employees to consolidate family engineering based experiences.

version: 2.4 status: draft October 20, 2017

1 Problem description

This article is focusing on ”composable architectures”. Composable architectures are designed for a single application domain, enabling the composition of products of which the definition is still evolving or hidden in the future.

A crucial design question is: *What is the desired granularity of the design, what are useful abstractions?* The granularity of the design is directly related to the question: *What are the appropriate aggregation levels for composition and integration?*

Most Product Creation Processes are based on a single dominating decomposition and integration model. This oversimplification causes many problems for development.

This article decribes an approach based on multiple viewpoints, matching the wide variety of concerns involved. Per viewpoint heuristics are given.

Application of a multiview approach requires customization of viewpoints and concerns. In general this means identification of the most relevant, important of critical issues, which are used to select a small manageable amount of viewpoints as main focus.

2 Views on Aggregation

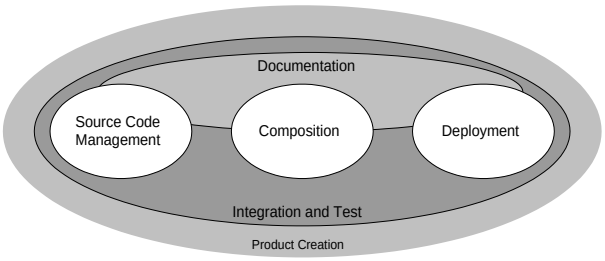


Figure 1: Venn diagram showing the overlap between Viewpoints on Aggregation Levels

Figure 1 shows a Venn diagram with 5 viewpoints with respect to aggregation levels, in the overall context of Product Creation. For every viewpoint the dominating concerns are mentioned in table 1 and the related aggregation levels or entities in table 2.

All entities in Documentation, Repository, Composition and Deployment are relevant for the Integration and Test viewpoint.

Viewpoint	Concerns
Documentation	Requirements, Specification, Design, Transfer, Test, Support
Source Code Management	Storage, Management, Generation
Composition	System, Subsystem, Function, Application
Deployment	Releasing, Distribution, Protection, Update, Installation, Configuration
Integration and Test	Confidence, Problem Tracking

Table 1: *Concerns per viewpoint*

Viewpoint	Entities
Documentation	Product Family, Product/System, Function/Feature, Subsystem, Component, Building Block, Module
Source Code Management	Package, File
Composition	Product, Executable, Dynamic Library, Component
Deployment	Distribution Medium (”CD”), Unit of Licensing (”SW key”), Package, Patch, Configuration data
Integration and Test	Test Configurations, Intermediate Integration results

Table 2: *Aggregation Levels or Entities per viewpoint*

3 Documentation

Many types of documentation are required when building Product Families by means of Composable Architectures. The granularity issues with respect to documentation are described in [3].

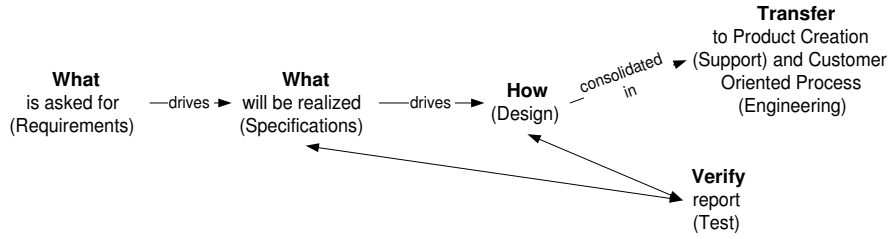


Figure 2: Visualization of documentation concerns

The aggregation levels for documentation are shown in table 2. Figure 2 visualizes the documentation concerns. For every level relevant documents should be produced, with respect to the *what* (requirements, specifications), *how* (design), *transfer* (to Customer Oriented Process), verification (test) and *how-to* (support to use reusable

assets in creation of products). In *what* and *how* documents a selected amount of *why* need to be present.

The documentation structure will evolve in time. This evolution requires explicit refactoring steps in the product family lifecycle. The *why* and to a lesser extent the *what* will be factored out, because this information is more stable and therefore re-useable than the *how*. Part of the information will move ”upward” in the aggregation level stack: generic patterns become clear, which are consolidated as abstractions on an higher aggregation level.

4 Source Code Management viewpoint

The elementary description of the system is in the source code. This source code is stored in a structured way in a repository, see figure 3. There is no hard requirement that the source code structure maps one-to-one on semantic entities in the composition world. However a one-to-one mapping helps in maintaining overview and understanding.

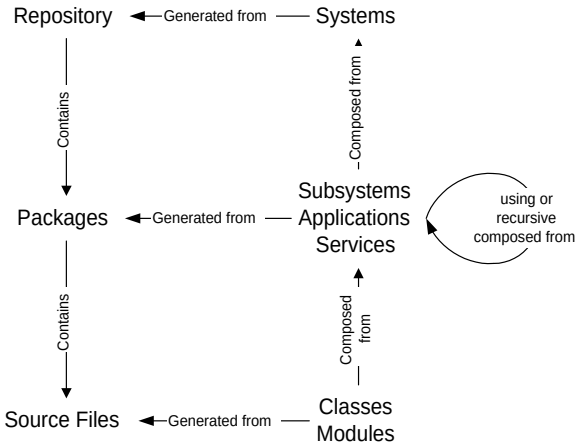


Figure 3: The source code is stored in files in a repository. The unit of structuring is called a package. These source code aggregation levels get a more semantic meaning when being used.

The main concerns in this view have to do with source code management:

- storage and accessibility of all source code
- version management; complete traceability of all versions and changes
- ownership for performance, quality and maintenance

The most widely used unit for management and storage of source data is **file**.

Source code in this context means all *original* formal descriptions, such as C, C++, include, text, data, make et cetera files. *Original* means that generated C-code does not belong to the source code, the data used for generating this code does belong to the source code.

The provide and require interface descriptions belong to the source code according to this definition, as do IDL interface definitions. For example see the KOALA component model as described in [6]. Generic subsystem configuration data defining the composition also belong to the source code.

Most source code need to be transformed in computer oriented intermediate formats before it can be used run time. The build step (compilation, building et cetera) required for this transformation may influence the repository structure. A well defined compile time dependency structure is desirable to enable a predictable composition step.

Table 3 shows the typical sizes, anno 2000, of source code repositories. The size is expressed in *lines of code* (loc). Historical data, see cost models in [2] and [1] shows a remarkable constant relationship between lines of code and the required manpower to create and maintain the software. The observed productivity in the Medical Imaging case study was ca. 10 kloc per manyear. Taking this number for a zero-order approximation the size of entities can be transformed in effort.

Entity	Typical size loc	packages
repository	1M-10M	10-100
package	10k–100k	
file	100-1k	

Table 3: *Typical Sizes of SW for Aggregation Levels*

This simple table illustrates a number of very essential design criteria, in relation to granularity of management.

Rules of thumb for typical file sizes are:

- Files should be larger than 100 loc;
  The overhead per file and the ”value” per file must be balanced.
- Files should be less than 1000 loc;
  Large files reduce the overview within the module. Larger files are an indication for a lack of modularity.

The number of packages in the repository is mostly restricted by usage and testing configuration management concerns. A fine granularity with respect to packages (subsystems, applications or services in the composition view) enables a fine grained and powerful composition. Coarse granularity of packages means

that more code is a priori bundled, constraining the freedom of the composer The downside of fine granularity is a combinatorial explosion of the amount of configurations.

From more pure source code point of view the considerations for package size are:

- at least 10 files per package;
  Packaging files or modules generates some overhead in usage and management. The value of this packaging must be substantial to offset this additional overhead.
- at most 100 kloc per package to maintain overview;
  For unambiguous package-ownership and sufficient overview.

5 Composition viewpoint

Composition involves glueing together and configuring available components. The result of the composition process are ”executable” entities such as components and plug-ins and more conventional executables and dynamic link libraries.

The granularity of these entities determines at the one hand the deployment flexibility at the other hand it determines the amount of testing and configuration management work.

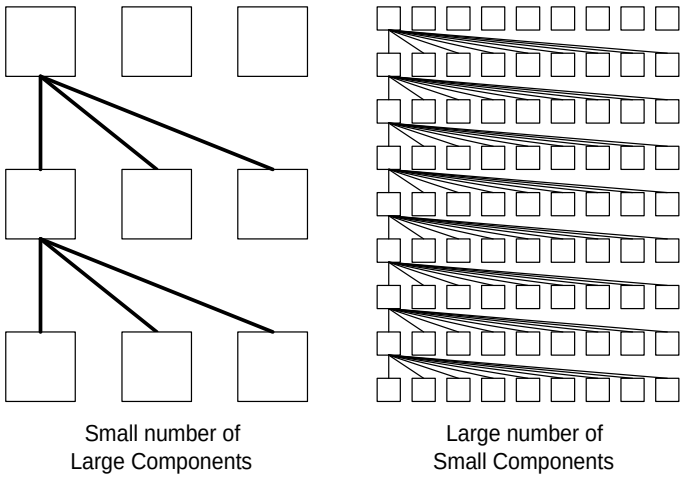


Figure 4: Coarse versus Fine grained with respect to the number of connections and relations; 9 large Components with 18 Connections, 81 small Components with 648 Connections

The number of relations between components is roughly in the order of $n^{1.5}$. Table 4 shows the number of components and the number of connections between

them. The number of desired architects is derived from the number of connections by means of zeroth order model. The "capacity" of an architect, the number of relations kept consistent and balanced by one architect, is used to determine the required number of architects:

$NumberOfArchitects = NumberOfConnections/Capacity$

Capacity of architects c		10	20	40
Number of components n	Number of relations $r = n\sqrt{n}$	Number of Architects $a = r/c$		
2	3	0	0	0
4	8	1	0	0
10	32	3	2	1
20	89	9	4	2
40	253	25	13	6
100	1000	100	50	25
300	5196	520	260	130
1000	31623	3162	1581	791

Table 4: *The relation between the number of components and the required number of architects, zero order model*

A somewhat more realistic model takes into account that large components will have more complex connections with other components than small components. Table 5 shows the same model with an additional *weight* factor to model the complexity of the connection. The weight curve applied is rather arbitrary, it reflects the experience of the author.

5.1 Optimal granularity for composition

The simple models in tables 5 and 4 make it immediately clear that a large quantity of components is undesirable. Assuming a total crews of circa 100 developers (which corresponds with today's multi-million lines of code repositories) it is reasonable to have 10 architects. The optimal number of components is than in the order 20 to 40.

The above reasoning is entirely macroscopic, calibrated with some typical Philips products. In specific cases plenty of reasons can exist which enable a higher number of components. For instance:

- presence of a stable reference model
- variation of components hidden behind an effective abstraction
- tangible and therefore understandable, predictable domain

	Capacity of architects c		10	20	40
Number of components n	Number of relations $r = n\sqrt{n}$	weight w	Number of Architects $a = (r * w)/c$		
2	3	12	3	2	1
4	8	9	7	4	2
10	32	4	14	7	3
20	89	2	22	11	5
40	253	2	39	19	10
100	1000	1	114	57	28
300	5196	1	534	267	133
1000	31623	1	3176	1588	794

Table 5: *The relation between the number of components and the required number of architects, first order model*

6 Field Deployment viewpoint

The granularity in the field deployment is determined by pragmatics of the Customer Oriented Process [5]. These pragmatics can be further decomposed, see table 6.

- granularity of sellable features and services
- lifecycle support
- internal logistics and production process

Table 6: *Decomposition of Field Deployment granularity drivers*

Conventional embedded products do not have any field deployment activity, these systems run out of the box. The increasing availability of network connectivity enables field updates, with all related configuration management consequences.

At this moment no heuristics are available for the granularity with respect to the drivers in table 6.

7 Integration and Test viewpoint

The real challenge in composable architectures is the integration and testing. Building small building blocks is the easy part, getting them to work correctly together with many other building blocks is more difficult.

A bottom up test philosophy, where every building block is verified in isolation helps, because it reduces the number of difficult to trace errors during integration. Bottom up testing needs to be complemented by an integration philosophy.

The time needed for verification of a building block depends exponentially or worse on its size. The combinatorial explosion of possible (and useful) states limits the optimal size of elementary building blocks. The typical size for verifiable modules is between 100 loc and 10 kloc. In section 5 the optimal number of components is derived to be between 20 and 40. For multi-million loc products a typical component will exceed the size of being bottom up verifiable.

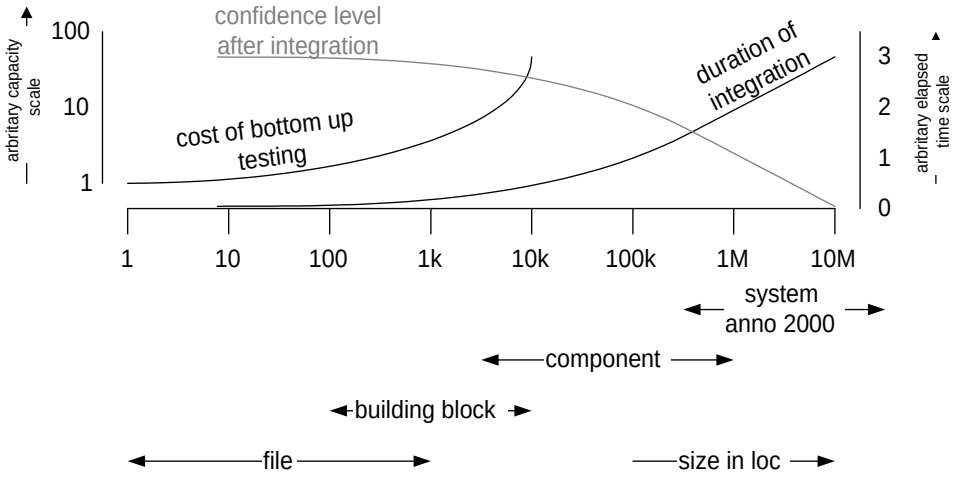


Figure 5: Integration and testing as function of size

Figure 5 shows the cost of bottom up testing as function of the size and the duration of the complementary integration also as function of the size. Note that the integration duration more or less increases linear, while the size increases exponential. The simple explication for this is that every integration step halves the number of modules to be integrated, the schedule looks like an horizontal binary tree. In other words the duration is logarithmic with the total size.

Integration is a non-exhaustive activity. Best case the most relevant (from usage and test risks perspective) areas are touched. This means that the level of confidence obtained by integration decreases with increasing size.

An acceptable level of confidence is only reached by a combination of bottom up testing, integration testing and intermediate common sense verification steps in between.

8 Acknowledgements

This paper has been written as part of the ”composable project”. The project members are: Pierre America, Hans Jonkers, Jürgen Müller, Henk Obbink, Rob van Ommering, William van der Sterren, Jan Gerben Wijnstra and Gerrit Muller. It has been discussed within the team, and the team contributed significantly to the contents.

Jürgen Müller suggested several improvements with respect to flow, consistency and balance. Wim Vree indicated multiple improvements, amongst others ”local terminology and acronyms”, which have either to be avoided or explained.

References

[1] Chris M.S. Abts, Barry W. Boehm, and Elizabeth Bailey Clark. COCOTS: A COTS software integration lifecycle cost model- model overview and preliminary data collection findings. `http://sunset.usc.edu/publications/TECHRPTS/2000/usccse2000-501/usccse2000-501.pdf`, 2000.

[2] Barry W. Boehm et al. Cocomo ii homepage. `http://sunset.usc.edu/research/COCOMOII/index.html`, 2000.

[3] Gerrit Muller. Granularity of documentation. `http://www.gaudisite.nl/DocumentationGranularityPaper.pdf`, 1999.

[4] Gerrit Muller. The system architecture homepage. `http://www.gaudisite.nl/index.html`, 1999.

[5] Gerrit Muller. Process decomposition of a business. `http://www.gaudisite.nl/ProcessDecompositionOfBusinessPaper.pdf`, 2000.

[6] Henk Obbink, Jürgen Müller, Pierre America, and Rob van Ommering. COPA: A component-oriented platform architecting method for families of software-intensive electronic products. `http://www.hitech-projects.com/SAE/COPA/COPA_Tutorial.pdf`, 2000.

History

Version: 2.4, date: June 13, 2002 changed by: Gerrit Muller

- minor change

Version: 2.2, date: February 16, 2001 changed by: Gerrit Muller

- layout update

Version: 2.1, date: November 28, 2000 changed by: Gerrit Muller

- extended the abstract
- added a description of ”composable architectures” to the problem description
- Source Code management:
  - ”MGR” replaced by Koala-citation
  - added explanation of loc and relation to amount of work
  - more explanation added about package size considerations

Version: 2.0, date: october 13, 2000 changed by: Gerrit Muller

- added section "problem description"
- redesign of figure 1, "Venn diagram"
- extended table 1 with "integration and test"
- replaced ”engineering” by ”transfer”
- replaced ”repository” by ”source code management”

Version: 1.2, date: september 29, 2000 changed by: Gerrit Muller

- Added ”support” as documentation concern
- Adapted layout change of frontpage

Version: 1.1, date: september 28, 2000 changed by: Gerrit Muller

- removed error in table ComponentFirstOrderConnectionTable
- Figure Views on Aggregation Levels renewed
- removed the Integration and Test column from the concerns and entities tables

Version: 1.0, date: september 27, 2000 changed by: Gerrit Muller

- ALsourceCodeViewpoint: ”Component” removed
- AggregationLevelsViews split up in new figure plus 2 tables, one with concerns per viewpoint, the other with entities per viewpoint
- Added documentation viewpoint figure
- Added documentation concerns: engineering and test
- replaced ”object” by ”building block” and ”module”
- Composition viewpoint: tables reorganized, ”span-width” replaced by ”capacity”
- Slides: Titles reflect viewpoint

Version: 0.2, date: september 25, 2000 changed by: Gerrit Muller

- no changelog maintained yet

Version: 0.1, date: september 19, 2000 changed by: Gerrit Muller

- no changelog maintained yet

Version: 0, date: april 3, 2000 changed by: Gerrit Muller

- Created, no changelog yet