Software-Engineering 2

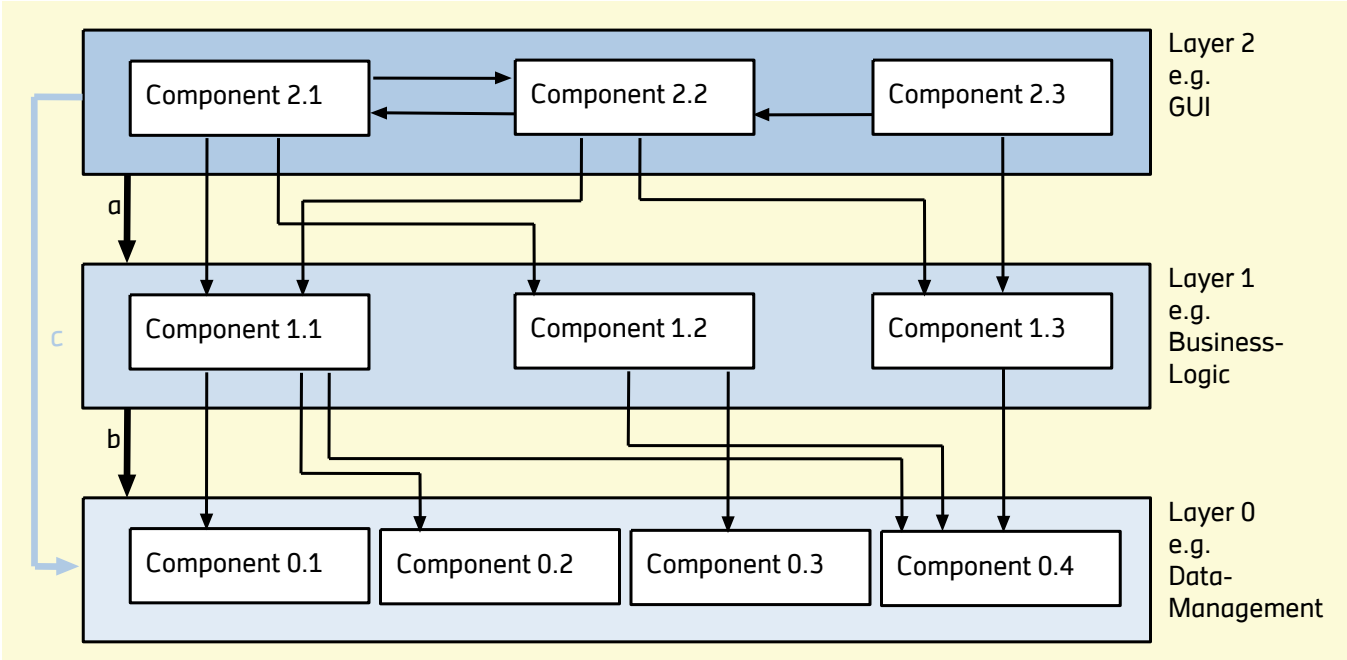
Prof. Dr.-Ing. Gerhard Wanner
Email: wanner@hft-stuttgart.de

QUALITY ASSURANCE FOR SW-ARCHITECTURES

Overview

- Principle of architecture management
  - 3-Layer-Architecture
  - Building a healthy software-basis
  - Consistency-analysis
  - Architectures and coupling
- The two dimensions of a software-architecture
- Tools
  - Criteria for architecture management tools
  - Tool list.

Layered architecture



Layered architecture with dependencies

Layered architecture (Cont.)

- Layered Architecture
  - Components within one layer can access each other in any way
    - But often additional rules, e.g. unidirectional usage → OSGi
  - Beneath the layers the access-rules are more strictly
    - Layers are ordered according to their abstraction-level
  - The Layers can be arranged in the following ways
    - Layers with strict order
      - Layers with a higher abstraction-level can access all layers with a lower abstraction-level, but not vice-versa
    - Layers with linear order
      - More restrictive
      - One layer is only allowed to access a component of the next deeper layer
      - Examples for layered model with linear order
        - TCP/IP-4 layered architecture
        - ISO/OSI-7 layered architecture
  - Layered Architectures are useful, if…
    - Services of a layer are on the same abstraction-level
    - Layers are ordered according to their abstraction-level. This means that a layer only needs services of lower layers.

Dealing with Software Projects (1/2)

- 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
  1. Process models (RUP, V-Model XT, …), agile development (Scrum, Kanban, XP, …)
  2. Requirements management as a discipline,
     IREB (International Requirements Engineering Board),
     Model-driven requirement management (SysML, …)

Dealing with Software Projects (2/2)

- 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.

Building a healthy software basis

- If we want to achieve this goal we need to keep in mind the internal structure of the software (its architecture) and some other basic 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 does not prevent a software system to *perform poorly* – the software itself does not care about documented features ("The software is the design")
  - 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".

How to assure Architecture Quality? (1/2)

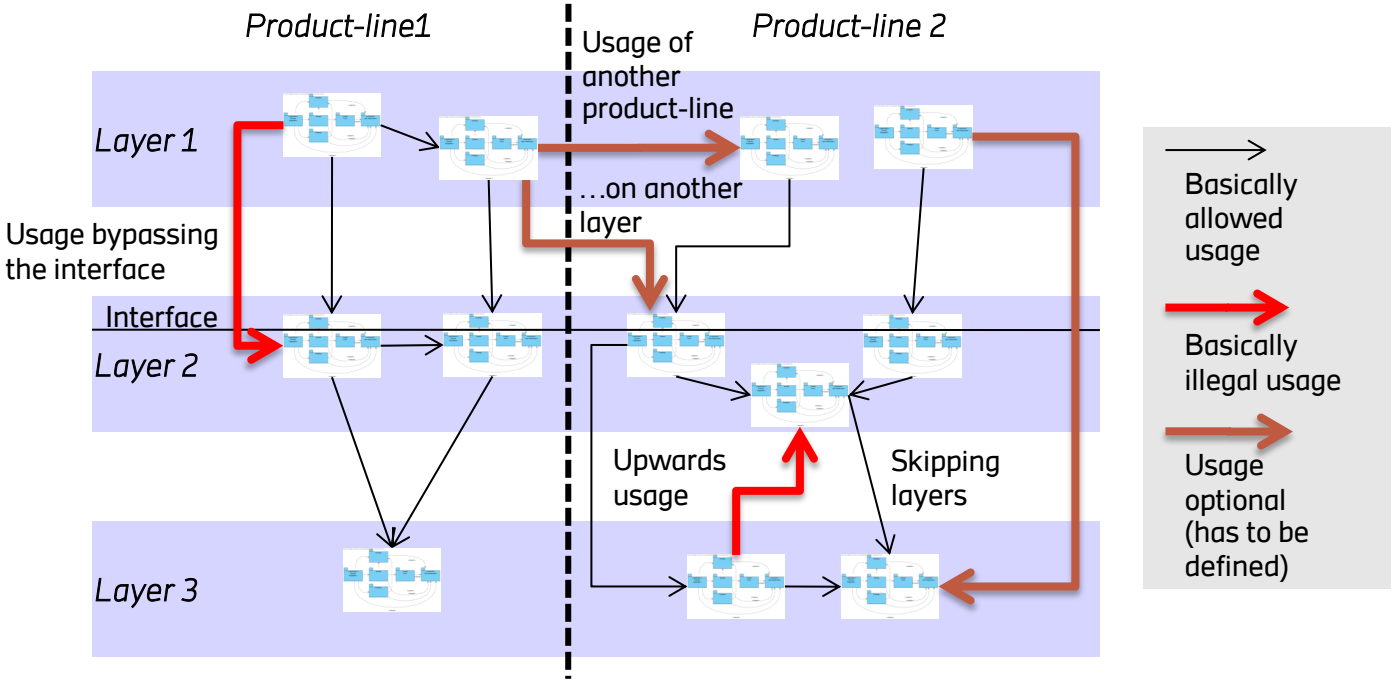
- So, there are two main goals to achieve
  - How to avoid that the *real structure* of the software decouples itself from its *planned architecture*
  - How to assure a *certain level of technical quality* by enhancing understandability and testability, while controlling complexity
- This can be done by
  - *Managing* the overall dependency structure *actively*
  - *Controlling* complexity *actively*
  - *Integrating* both activities directly into the *development cycle*
- Therefore, it's necessary to close 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.

How to assure Architecture Quality? (2/2)

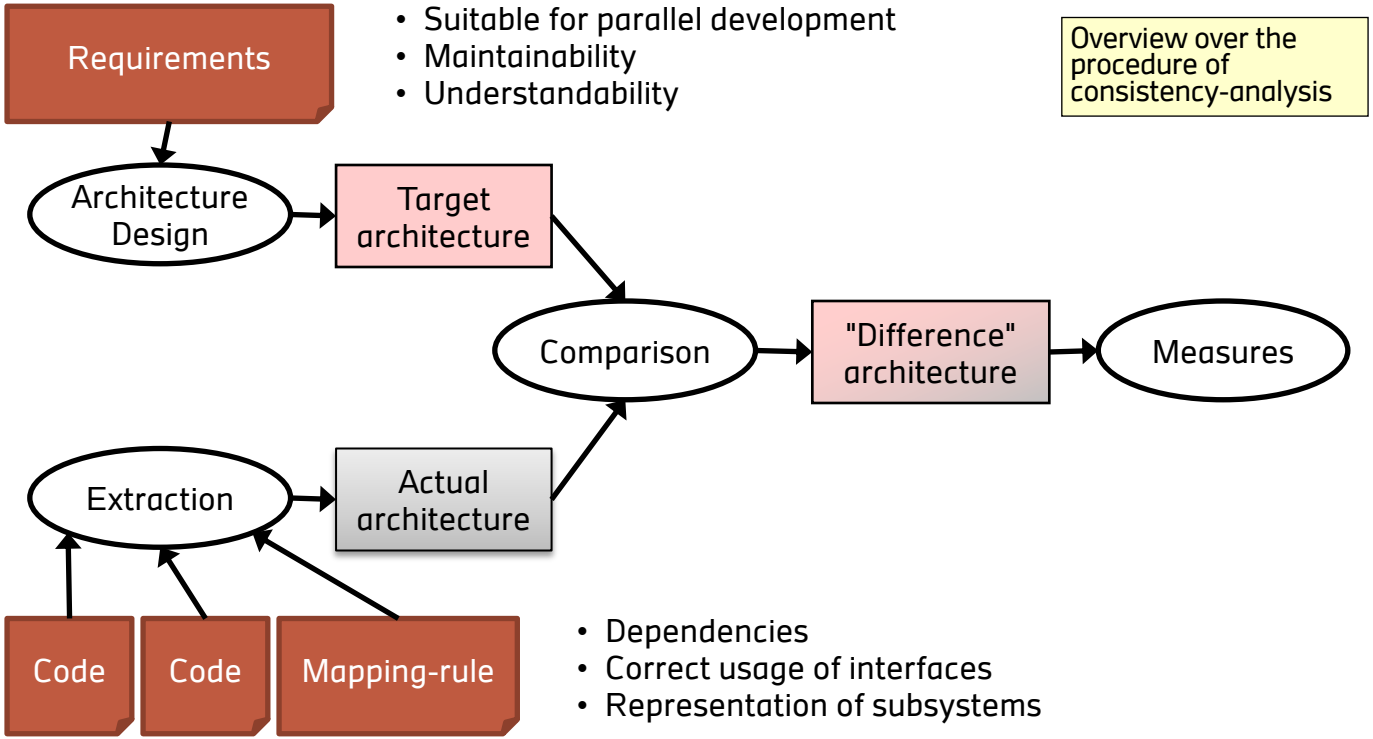
- Questions
  - Is the specified architecture adhered?
  - Does the architecture decay?
- How is it accomplished?
  - All checks are accomplished statically, without executing the system
- When is it accomplished?
  - During the system-development
    - Is the architecture adhered?
    - Identification of necessary changes of the architecture
  - During the reverse-engineering
    - Recovery of design-documents
    - Documentation of an abstract view of the system
  - To understand a program
    - Check of hypotheses of the architecture.

Consistency-Analysis – Layers and product-lines

- Allowed, optional and illegal dependencies



Consistency-Analysis – Overview



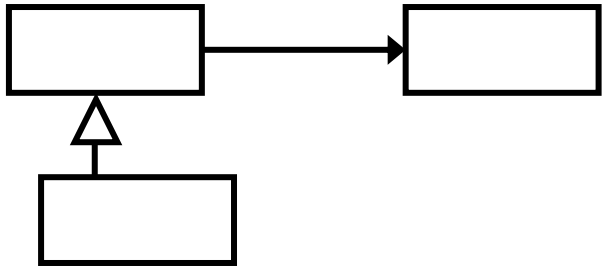
- Suitable for parallel development
- Maintainability
- Understandability

Overview over the procedure of consistency-analysis

- Dependencies
- Correct usage of interfaces
- Representation of subsystems

Consistency-Analysis – Extract the facts

- Actual architecture has to be "lifted" to the same level
- Starting point: basic artifacts and their relationships



Basic artifacts and their relationships

- Artifacts
  - Attributes, Variables
  - Methods, Functions
  - Classes
  - Files
  - Packages/Folders
  - Subsystems
- Relationships
  - Method-, Function-calls
  - Attribute-access
  - Inheritance
  - Usage of types (parameters, local variables)
  - Includes-Relationship.

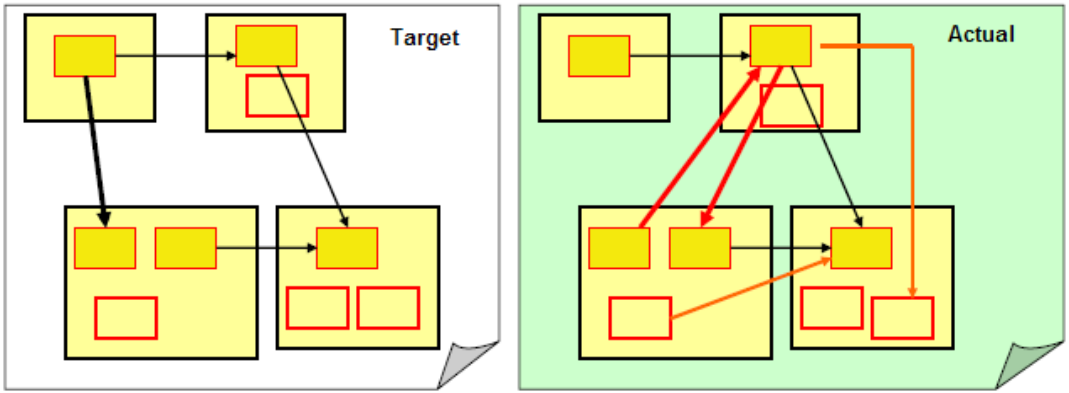
Consistency-Analysis – Mapping rules

- Subsystems are not covered in common programming languages
- Additional mapping rules are necessary, that declares membership of the artifacts
- Approaches
  - Based on directory-structures / package-names
  - Based on name-schemata or other attributes (author, comment, change-history)

- Subsystem A
  - All files in directory X
- Subsystem B
  - All artifacts with the prefix "Util"
  - All artifacts without the "internal"
- Subsystem C
  - All files that are edited by author "Bill"

Examples

Consistency-Analysis - Subsystems



Comparison of the target and the actual architecture or a system

- Consistent dependencies
- *Planned relationships that are not there (redundance?)*
- *Relationships that are there but not planned (violation?)*
- Interface-violation.

Architectures and coupling

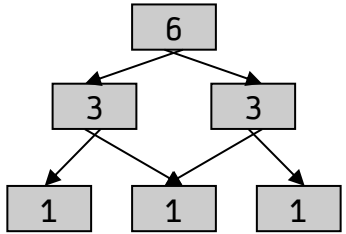
- Only flexible architectures are good architectures
- You are always shooting on moving targets
- To gain flexibility and potential re-usability you have to minimize coupling

Always avoid cyclic dependencies

Your systems flexibility is inverse proportional to its coupling.

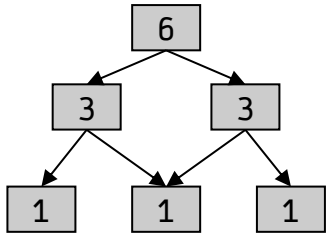
Architectures and coupling – How to measure coupling? (1/2)

- CCD: Cumulated Component Dependency
- ACD: Average Component Dependency
  - Average number of direct and indirect dependencies
  - ACD = CCD / number of elements → lower is better
- Relative ACD
  - RACD = ACD / number of elements * 100 → lower is better

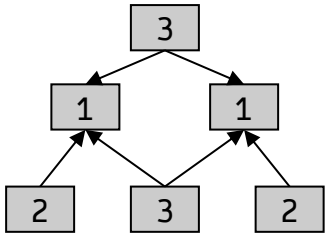


CCD = 15
ACD = 15/6 = 2,5
*RACD = 2,5/6*100 = 41,7%*

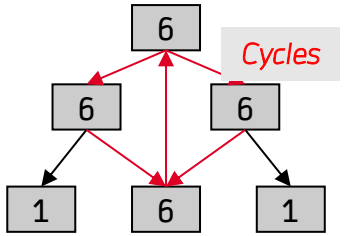
Architectures and coupling – How to measure coupling? (2/2)



CCD = 15
ACD = 15/6 = 2,5
*RACD = 2,5/6*100 = 41,7%*



Dependency Inversion
ACD = 12/6 = 2
*RACD = 2/6*100 = 33,3%*



Cycles
ACD = 26/6 = 4,33
*RACD = 2,5/6*100 = 72,2%*

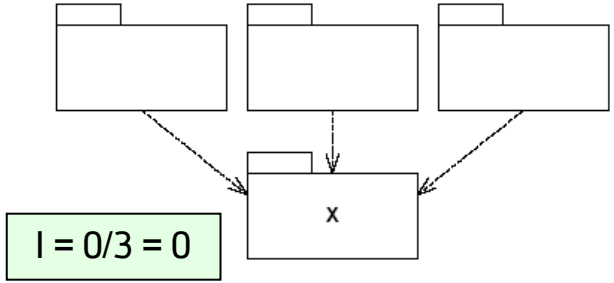
	Components	ACD	RACD
Spring 2.5.1	1.595	40	2,5%
Groovy 1.5.4	833	229	27,6%
Hibernate 3.2.6	1.085	727	67,0%

It's easier to achieve lower RACD values for bigger systems.

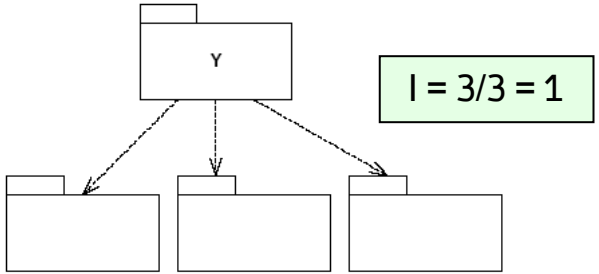
Soucre: JavaMagazin 11.2017

Architectures and coupling - Architecture metrics (1/3)

- Instability (Robert C. Martin)



X is „stable“



Y is „instable“

- D_i = Number of incoming dependencies
- D_o = Number of outgoing dependencies
- Instability $I = D_o / (D_i+D_o)$

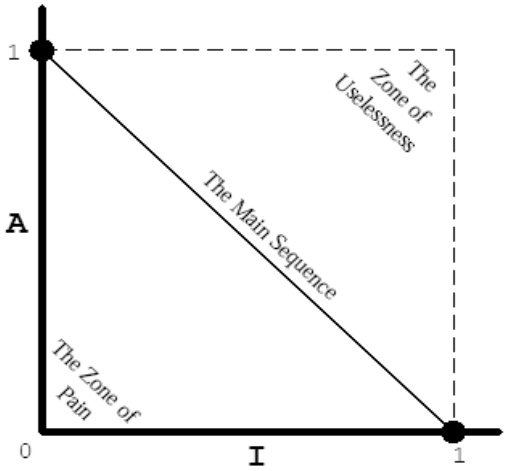
- *Build on abstractions, not on implementations!*

Architectures and coupling - Architecture metrics (2/3)

- Abstractness (Robert C. Martin)
  - N_c = Total number of classes in a package
  - N_a = Number of abstract classes and interfaces in a package
  - Abstractness $A = N_a/N_c$
  - The range for this metric is 0 to 1, with A=0 indicating a completely concrete package and A=1 indicating a completely abstract package.

Architectures and coupling - Architecture metrics (3/3)

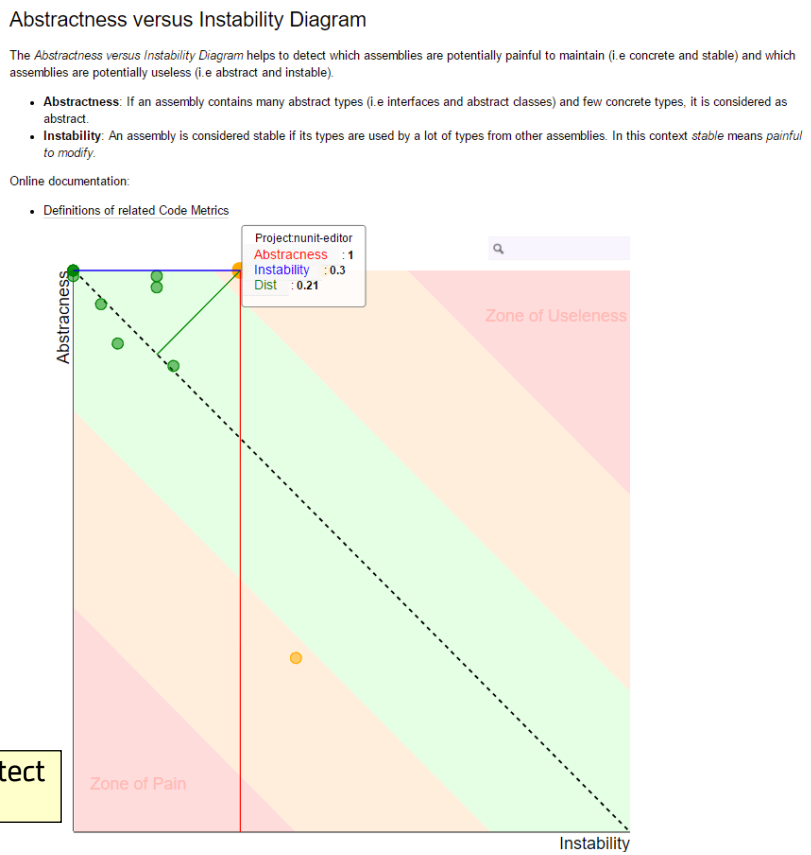
- Distance (Robert C. Martin)
  - *D = A + I - 1*
    - Distance = Abstractness + Instability -1
  - Value range *[-1 .. +1]*



  - Negative values are in the „Zone of pain"
  - Positive values belong to the „Zone of uselessness"
  - The more abstract a package is, the more stable it should be since it should have many clients that depend on its abstractions
    - Packages that are maximally stable (I = 0) and at the same time maximally abstract (A = 1) are considered good. Similarly, packages that are unstable (I = 1) and at the same time concrete (A = 0) are also "good packages."
  - Good values are close to zero (e.g. -0,25 to +0,25).

Architectures and coupling – Architecture metrics (tool example)

- Jarchitect of Coder Gears
  - Interactive Abstractness vs. Instability Diagram.



Source: Coder Gears – JArchitect https://www.jarchitect.com

Architectures and coupling – Problem of cyclic dependencies (1/3)

- Cyclic dependencies are evil

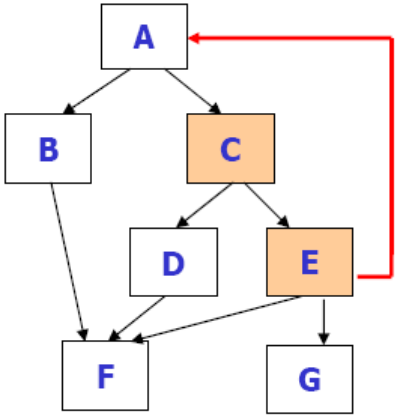
> Guideline: *No Cycles between Packages*. If a group of packages have cyclic dependency then they may need to be treated as one larger package in terms of a release unit. This is undesirable because releasing larger packages (or package aggregates) increases the likelihood of affecting something
>
>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

> *The dependencies between packages must not form cycles.*
>
> Rober C. Martin

> *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.
>
> John Lakos

Architectures and coupling – Problem of cyclic dependencies (2/3)

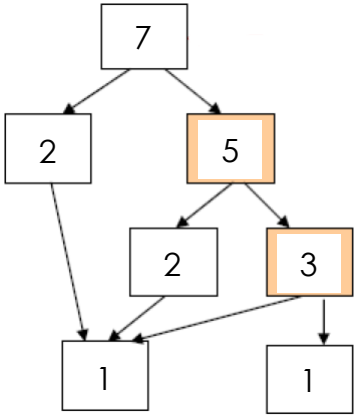
- Without cycle
  - E depends on F, G
  - C depends on D, E, F, G
  - C has to be consistent with D, E, F, G
  - Changes to C don't affect other parts
  - To compile/test E only F, G is necessary
  - Ordered integration of the complete system possible (contrariwise to the dependencies)



- With cycle
  - E depends on F, G, A, B, C, D
  - C depends on D, E, F, G and A, B
  - C has to be consistent with D, E, F, G and A, B
  - A, C, E must be consistently changed at the same time
  - To compile/test E, also other parts are necessary!
  - Consistent integration needs possibly two cycles.

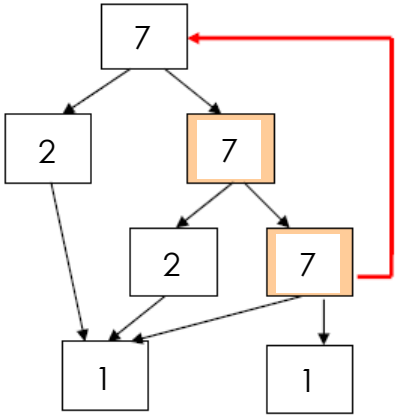
Architectures and coupling – Problem of cyclic dependencies (3/3)

- Without cycle



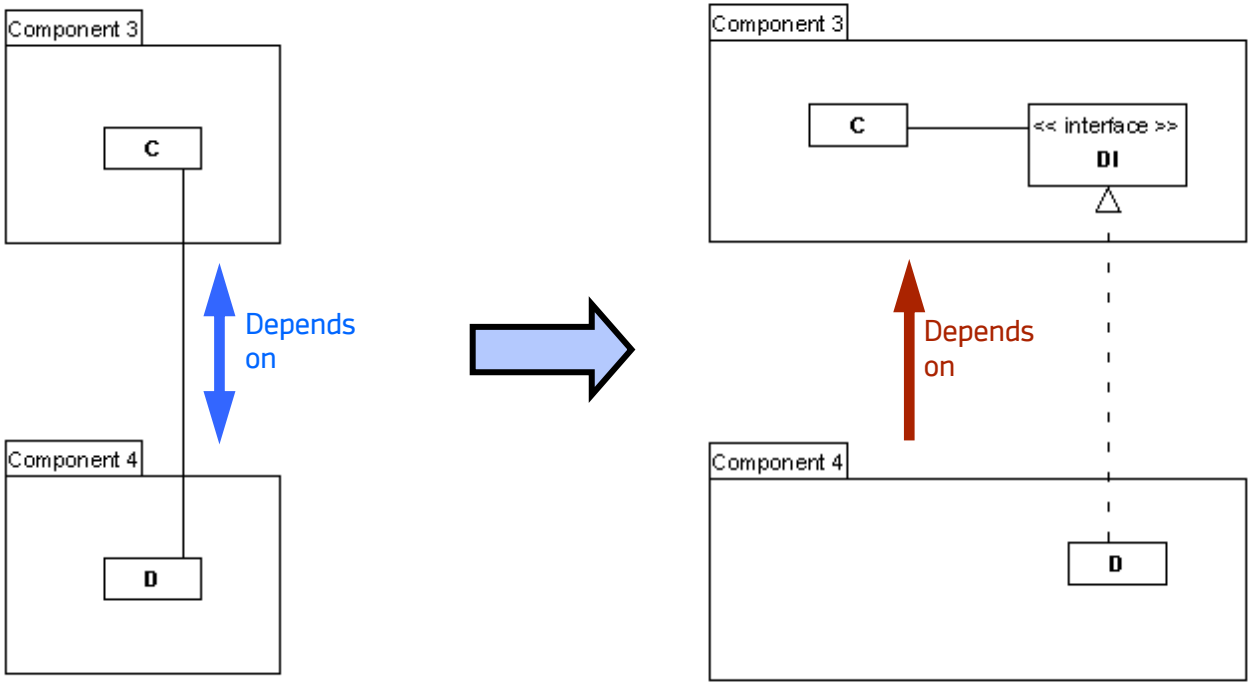
- CCD = 21
  ACD = 20/7 = 3

- With cycle

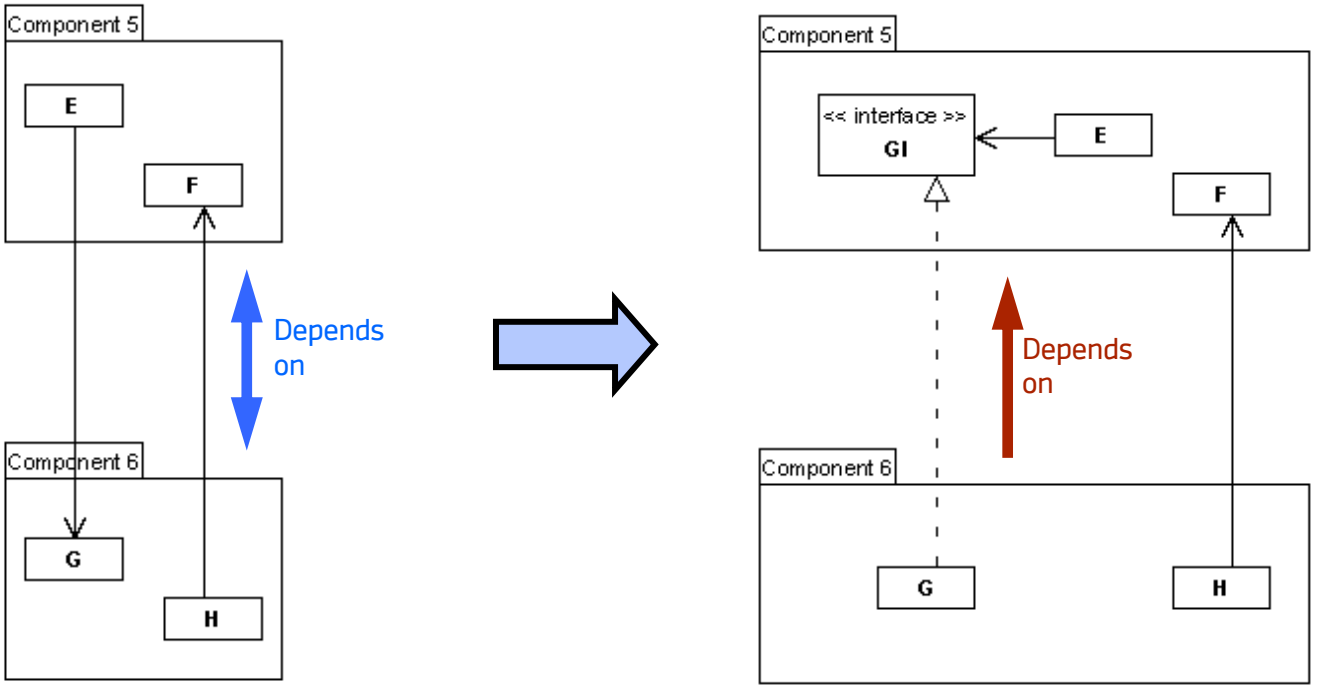


- CCD = 27
  ACD = 27/7 = 3,9

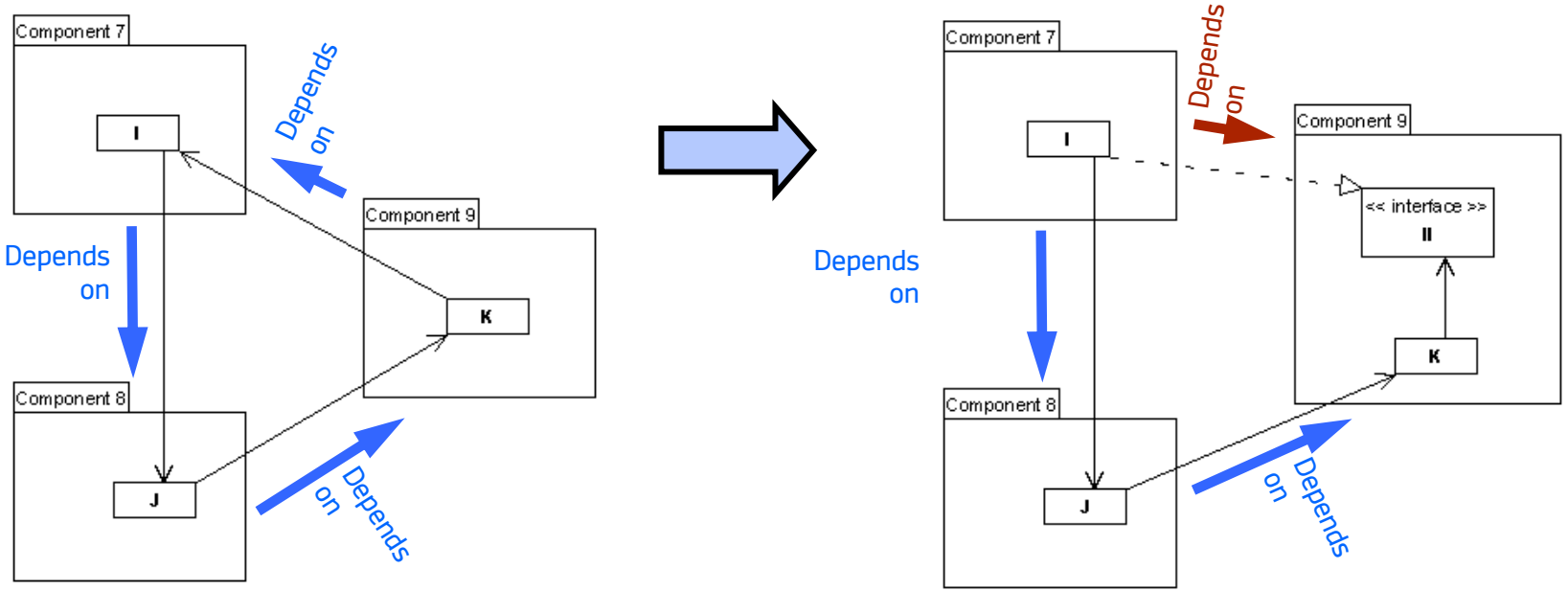
Architectures and coupling – Eliminating cycles (1/3)



Architectures and coupling – Eliminating cycles (2/3)



Architectures and coupling – Eliminating cycles (3/3)

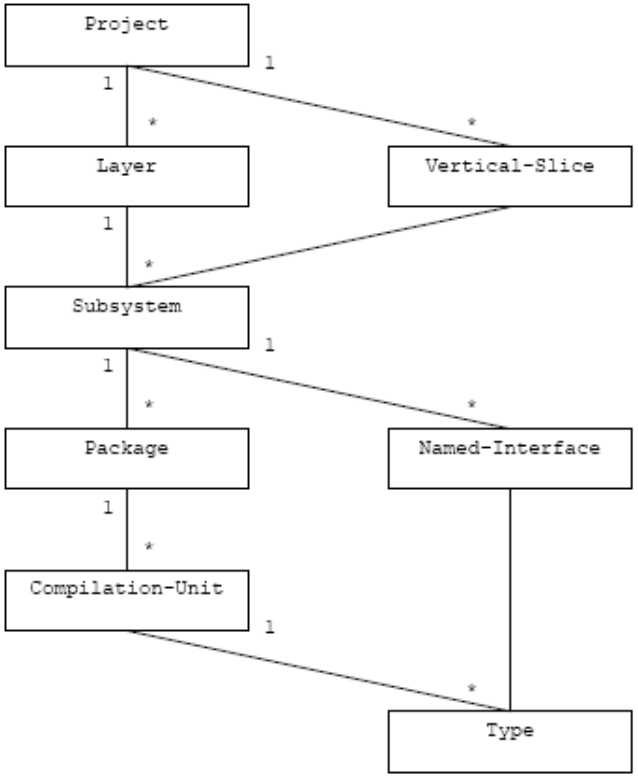


Overview

- Principle of architecture management
  - 3-Layer-Architecture
  - Building a healthy software-basis
  - Consistency-analysis
  - Architectures and coupling
- The two dimensions of a software-architecture
- Tools
  - Criteria for architecture management tools
  - Tool list.

Division of a project: The Software Architecture Model

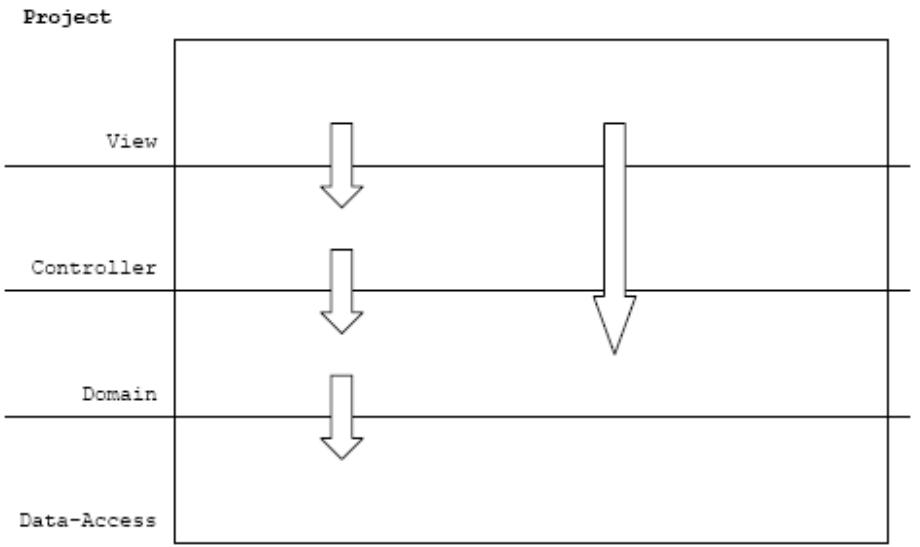
- "Craig Larman" (and many other sources) describes a model for partitioning a project
  - It is a widely adopted method of describing the overall software architecture (static view).



Example for partitioning a project

How to find the overall static structure for a project? (1/4)

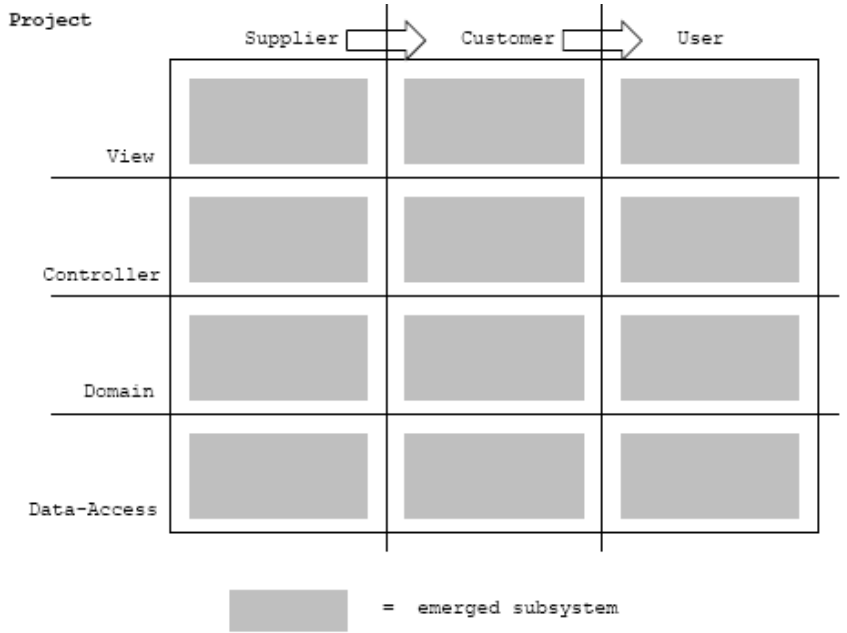
- Step 1: Divide horizontally
  - First of all, you should develop an idea of the technical layers dividing your project horizontally and their interdependencies
- Typical layers are
  - View
  - Controller
  - Domain
  - Data-Access
  - …



Example for an overall picture of a layered architecture

How to find the overall static structure for a project? (2/4)

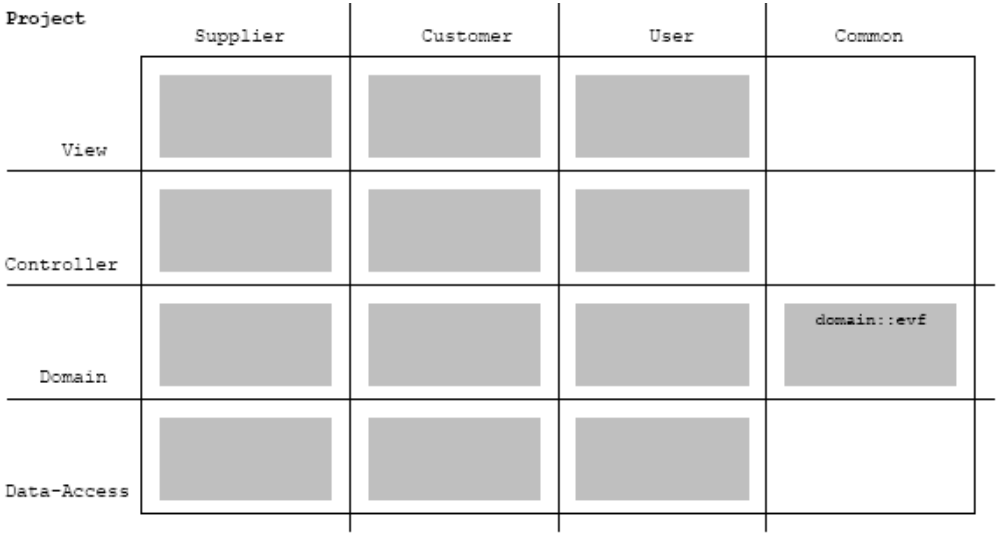
- Step 2: Divide vertically
  - The vertical division is normally driven by business or application logic related aspects finding divisions like supplier, customer, user, … with suggested uses dependencies.



Example for a vertically divided project

How to find the overall static structure for a project? (3/4)

- Step 3: Add pure technical subsystems
  - Normally it is necessary to create subsystems dealing with pure technical aspects
  - Also possible: Additional layer (e.g. foundation).



Example for a vertically divided project with an additional technical subsystem

How to find the overall static structure for a project? (4/4)

- 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). Example:

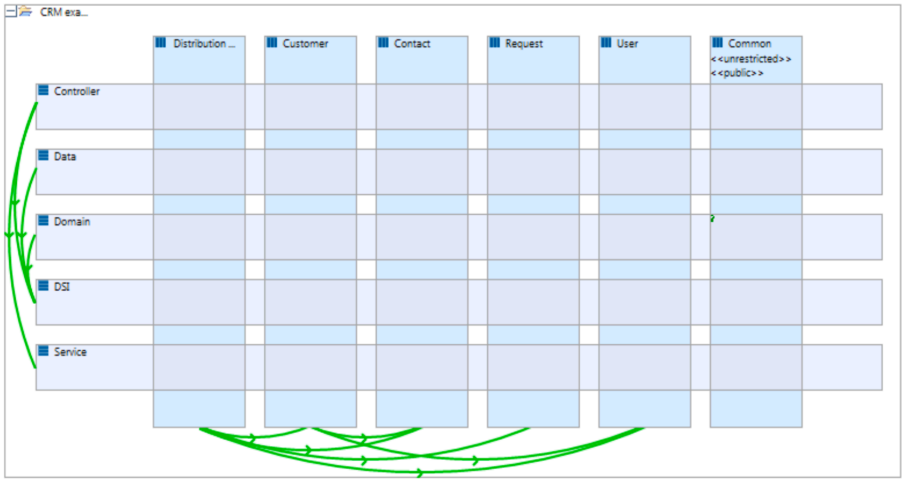
    ```
    [company-prefix].[vertical-slice-name].[layer-name]
    ```

  - Every package starting with an identical prefix belongs to the same subsystem. For example, the following packages implement domain::evf

    ```
    de.hftstuttgart.application.common.domain.event
    de.hftstuttgart.application.common.domain.event.generation
    de.hftstuttgart.application.common.domain.event.dispatcher
    ```

Sonargraph – DSL (1/4)

- Sonargraph 8 and higher allows the definition of an architecture via a DSL that is expressive and readable enough so that every developer is able to understand it
  - The graphical representation in Sonargraph 7 allowed the creation of your architectural blueprint in one single diagram
  - This leads to potentially very big and complex diagrams that are difficult to understand.



Source: Sonargraph Tutorial v.7.1 Architecture Modeling

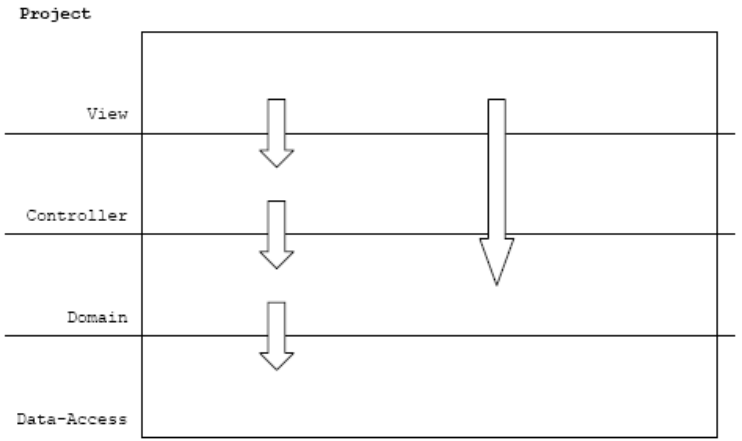
Sonargraph – DSL (2/4)

- The requirements for the new DSL approach in Sonargraph 8 were the following:
  - It should be possible to describe an architecture in a set of files
    - Some of them should be generic enough so that they could be reused by many projects, e.g. a generic template describing the layering of a system
  - It should be possible to describe an architecture in form of several completely independent aspects
    - E.g. one aspect describes layering, another aspect describes components, and a third aspect looks at separation of client and server logic
  - On the other hand, the language should also be powerful to describe the complete architecture in a single aspect
  - The DSL must be easy to read and easy to learn
  - The restrictions for dependencies should allow also the specification of dependency types (e.g. "new", "inheritance", etc.).

Sonargraph – DSL (3/4)

- Example DSL (Layering)

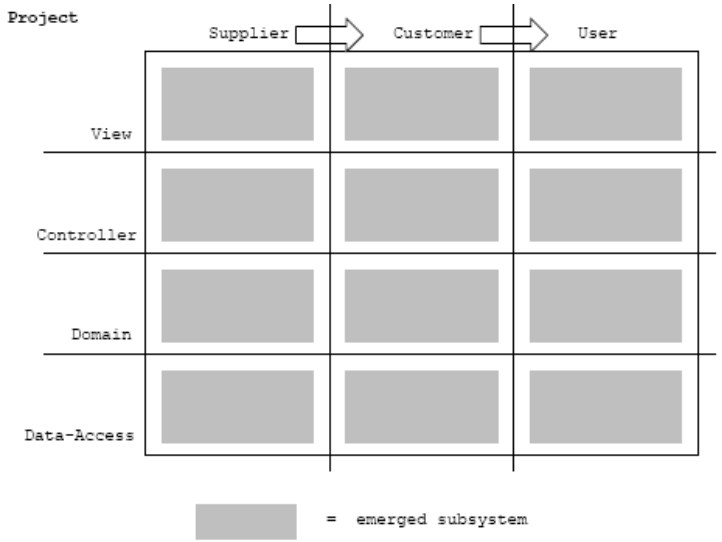
```
// Layering
artifact Data-Access
{
    include "**/data-access/**"
}

artifact Domain
{
    include "**/domain/**"
    connect to Data-Access
}

artifact Controller
{
    include "**/controller/**"
    connect to Domain
}

optional artifact View
{
    include "**/view/**"
    connect to Controller, Domain
}
```



Sonargraph – DSL (4/4)

- Example DSL (Components)

```
// Components
artifact User
{
    include "User/**"

    apply "Layering"
}

artifact Customer
{
    include "Customer/**"

    apply "Layering"
    connect to User
}

artifact Supplier
{
    include "Supplier/**"

    apply "Layering"
    connect to Customer
}
```



Measures to fix architecture problems

- Dependency violations
  - Missing abstraction-layers?
  - Are sub-functions in the wrong subsystem?
- Interface violations
  - Are the violations justifiable?
  - Is an additional interface necessary?
- Cycles
  - Eliminate cycles at central artifacts
    - There is a high chance to eliminate many cycles at once.

Overview

- Principle of architecture management
  - 3-Layer-Architecture
  - Building a healthy software-basis
  - Consistency-analysis
  - Architectures and coupling
- The two dimensions of a software-architecture
- Tools
  - Criteria for architecture management tools
  - Tool list.

Criteria for Architecture Management Tools (1/6)

- Questions the tool must answer
  - Which parts of an application are used from which other parts or uses other parts?
    - „Parts" are physical units of the programming language, e.g. jar-Files, packages or types
    - „used" includes all possibilities of referencing, e.g. inheritance, declaration or instantiation
  - Which parts of the application aren't used?
  - Which parts of the application are used heavily (changes in these parts have a big impact)?
  - Which 3rd party libraries are used? Where are the used? Is the usage intended?
  - Are there 3rd party libraries that contain similar functionality?
  - Which license models are carried into the applications through the usage of 3rd party libraries?

Criteria for Architecture Management Tools (2/6)

- Support of rules / logical model of the architecture
  - The tool must support the definition of rules to define which parts of an application are allowed to use which other parts of the application
    - These allowed and forbidden dependencies can be defined on the level of the physical components
  - More progressive is the definition of a logical model of the architecture by the software architect
    - These architecture rules are defined on a logical level and then mapped to the physical components by the defined model
  - Additional Questions
    - Is there a tight coupling inside a logical component?
    - Are the components itself coupled loosely?
    - Are there cyclic dependencies (one cyclic dependency between components makes them to one big single component)?

Criteria for Architecture Management Tools (3/6)

- Refactoring support
  - Which possibilities to break-up architectural defects are available?
    - The analysing architect wants to know where and how he can apply refactoring-measures
  - Is it possible to apply the refactoring-measures virtual to see the impact on the overall architecture?
    - Then it's possible for the architect to make a decision before the refactoring
  - Is it possible to put all refactoring-measures in a work list to assign them to different developers?

Criteria for Architecture Management Tools (4/6)

- Support for key ratios
  - Does the tool provide common statements about quality, especially about maintainability?
  - Does it provide key ratios to test coverage, documentation coverage, compliance to programming guidelines, metrics, code-duplication?
  - Is it possible to compare different snapshots of a project to see trends of key ratios?

  - But:

Source: dilbert.com

Criteria for Architecture Management Tools (5/6)

- Visual support
  - There are often hundreds of thousands physical parts an application consists of (especially if 3rd party libraries are used)
  - Does the tool produce reports that can be viewed using a browser?
  - Are the results visualized in an adequate way?
  - Is it possible for the observer to drill-down the results from a project-focus down to the method-level?
  - Are the different key ratios of a component pictured in an adequate manner?

Criteria for Architecture Management Tools (6/6)

- Integration support
  - Is it possible to integrate the tool into the work environment of the developers?
    - Optimal is an integration into the IDE to directly inform the developer if he/she violates the rules of the architecture before committing code into the repository
  - Is it possible to automate the execution of the tools?
    - This is necessary to integrate the tool into the build- or QA-process.

Tools (excerpt)

- Sonargraph Architect from Hello2Morrow
  - http://www.hello2morrow.com/
  - Commercial product
    - Free for open-source projects and academic usage
  - Integration into Eclipse with direct feedback to the developer
  - *From version 7 to version 8 changed from graphical definition of an architecture to a dsl-based definition*
- XRadar
  - http://xradar.sourceforge.net
  - Open source
  - Integration into build-process (Ant, Maven)
  - Possibility to compare different builds
- JDepend
  - http://www.clarkware.com/software/JDepend.html
  - Open source
  - Mainly to calculate metrics
  - JDepend Fixture for FitNesse supports dependency analysis.

Tools (excerpt)

- ArchUnit
  - https://www.archunit.org/
  - Open source
  - Enables the import of the classes of an application into a special Java code structure that allows testing the code and its structure with any Java unit testing framework
  - ArchUnit provides a fluent DSL which lets you attach conditions to a given situation
  - Included rules: Package dependency checks, Class dependency checks, Class and package containment checks, Inheritance checks, Annotation checks, Layer checks, Cycle checks
  - Nice introduction: https://blogs.oracle.com/javamagazine/unit-test-your-architecture-with-archunit

Tools (excerpt)

- ArchUnit (examples)

```
ArchRule rule = ArchRuleDefinition.classes()
    .that().resideInAPackage("..domain..")
    .should().onlyBeAccessed()
    .byAnyPackage("..domain..", "..application..");
```

- The ".." in the package notation refers to any number of packages
  - This syntax is inspired by AspectJ pointcuts
  - Thus, in this example, the ArchRule applies to any class inside the package com.company.app.domain.model

```
ArchRule rule = ArchRuleDefinition.methods()
    .that().arePublic()
    .and().areDeclaredInClassesThat()
        .resideInAPackage("..adapters.primary.web..")
    .and().areDeclaredInClassesThat()
        .haveSimpleNameEndingWith("Controller")
    .and().areDeclaredInClassesThat()
       .areAnnotatedWith(Controller.class)
    .should().beAnnotatedWith(RequestMapping.class);
```

Literature

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  - Design Patterns: Elements of Reusable Object Oriented Software
  - Addison-Wesley Longman, 1997, ISBN 0201633612
- Robert C. Martin
  - Agile Software Development
  - Prentice Hall 2003
- Craig Larman
  - Applying UML And Patterns
  - Prentice Hall 2002
- John Lakos
  - Large-Scale C++ Software Design
  - Addison-Wesley 1996
- hello2morrow GmbH
  - Sonargraph - Whitepaper.