How to Organize your Code

Dietmar Menges

ObjektForum, Andrena Objects, 26.06.2017

© 2005-2017, hello2morrow

Abstract

In vielen nicht-trivialen Softwareprojekten steigen die Wartungskosten (Änderungen, Erweiterungen, Fehlerkorrekturen) über deren Lebenszeitraum signifikant. Dies kann unter Umständen sogar zur frühzeitigen Einstellung solcher Softwareprojekte führen. Was sind die Gründe dafür? Wie kann man durch Code Organisation dem entgegen wirken?

© 2005-2017, hello2morrow

Über mich …

- Seit 1995 aktiv als Softwareentwickler
- Gearbeitet für:
  - 1996 DerData
  - 1997 OOTec
  - 2000 Valtech
  - 2005 hello2morrow (Mitgründer)
- In mehr als 35 Softwareprojekten aktiv mitentwickelt für Firmen wie:
  - BMW
  - Siemens
  - Krauss-Maffei Wegmann
  - LV1871
  - HypoVereinsbank
  - Tecoplan
  - Continental
  - Bayer

© 2005-2017, hello2morrow

Mein Weg zur statischen Codeanalyse …

- OOTec GmbH
  - 1997 Large-Scale C++ Software Design [John Lakos], Aktiver Einsatz des Component Dependency Analyzers von John Lakos
- Valtech GmbH
  - 2002 Dependometer (Java, Open Source)
  - 2003 Java Messplatz für BMW
- hello2morrow
  - 2005 SonarJ (Java)
  - 2009 Sonargraph – bis Version 7.x (Java)
  - 2012 Sonargraph – Next Generation, ab Version 8.x (Java, C#, C/C++)

© 2005-2017, hello2morrow

Inhalt

- Bedeutung wartbarer Software
- Zusammenhang zwischen Code Organisation und Wartbarkeit
- Logische Architektur als Hilfsmittel
- Positive Effekte
- Code Organisation mit Sonargraph

© 2005-2017, hello2morrow

Softwareprojekte erfordern permanente Wartung

- Fehler müssen korrigiert werden
- Neue Funktionalität wird implementiert
- Bestehende Funktionalität wird angepasst/erweitert
- Neue Technologien werden hinzugenommen oder ersetzen bereits Verwendete

© 2005-2017, hello2morrow

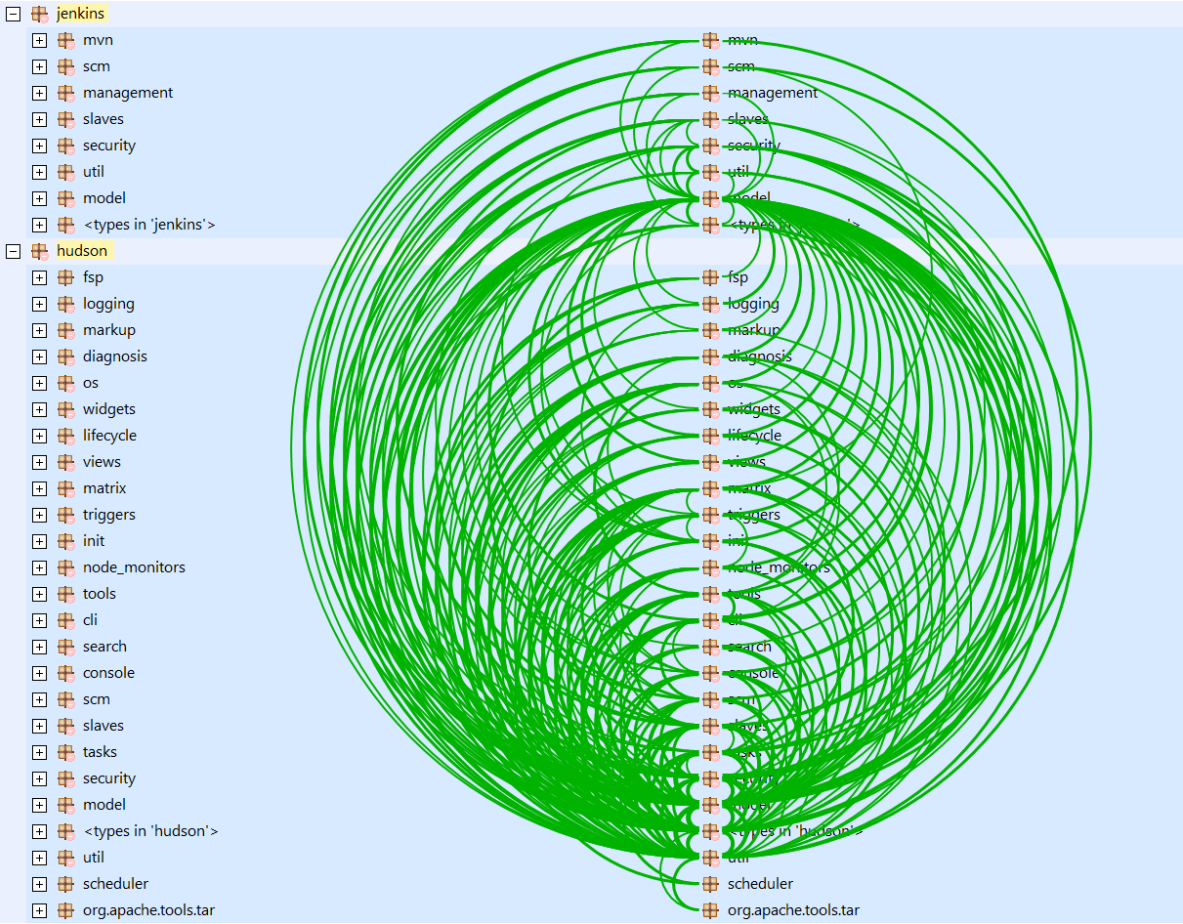
Über die Zeit stellen sich negative Symptome ein

- Die Fehlersuche/Korrektur gestaltet sich aufwändiger
- Es wird immer mehr Zeit benötigt neue Funktionalität zu implementieren
- Die Erweiterung der bestehenden Funktionalität ist zäh und führt zu vielen Regressionsfehlern
- Unzufriedene Teammitglieder springen ab
- Neue Teammitglieder haben Schwierigkeiten sich einzuarbeiten

$\Rightarrow$ Solche Symtome führen zu Software, die irgendwann nicht mehr wartbar ist!

© 2005-2017, hello2morrow

Software Erosion



- Immobility
- Opacity
- Fragility
- Rigidity
- Viscosity

(Robert C. Martin)

© 2005-2017, hello2morrow

Symptoms (Robert C. Martin)

- Rigidity – The system is hard to change because every change forces many other changes.
- Fragility – Changes cause the system to break in conceptually unrelated places.
- Immobility – It's hard to disentangle the system into reusable components.
- Viscosity – Doing things right is harder than doing things wrong.
- Opacity – It is hard to read and understand. It does not express its intent well.

Overall: "*The software starts to rot like a bad piece of meat*"

© 2005-2017, hello2morrow

Wenig Fortschritt und schlechte Stimmung im Team können viele Gründe haben

- Schlechte oder keine Führung
- Schlechte oder keine Prozesse
- Schlechtes oder kein Anforderungsmanagement
- Mangelnde Abstimmung/Kommunikation im Team
- …

⇒ Wenn jedoch die Wartbarkeit der Software gegen “0” geht, hilft auch keine der begleitenden Aktivitäten mehr!

© 2005-2017, hello2morrow

Code Organisation ist Abhängigkeitsmanagement

Um die Wartbarkeit einer Software zu ermöglichen und zu erhalten, ist die sorgfältige und aktive Verwaltung der Abhängigkeitsstruktur entscheidend.

Die Zuordnung der Programmierartifakte zu den Elementen einer logischen Architektur (als übergeordnete Abstraktionsebene) trägt entscheidend zur Erhaltung der Wartbarkeit bei.

© 2005-2017, hello2morrow

“Component” – die kleinste physikalische Einheit

- C++: Implementierungsdatei (.cpp, …) + Header (.h, …)
- C#: Source Datei (.cs)
- Java: Source Datei (.java)

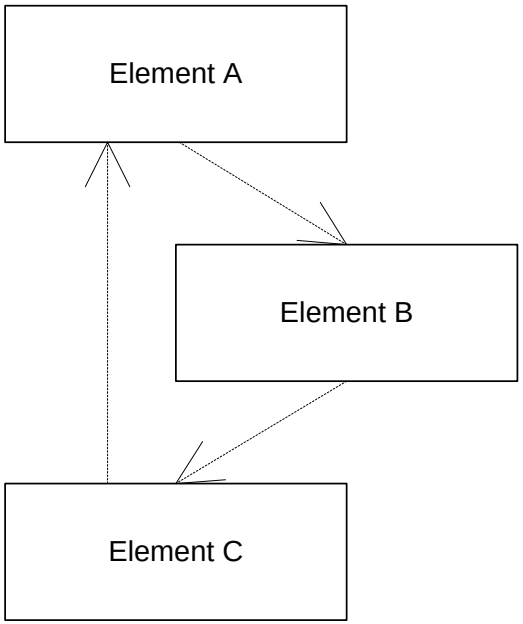
- Die “Components” werden über Verzeichnisse organisiert unterhalb ihrer Wurzelverzeichnisse innerhalb eines Moduls

- C# und C++: Verzeichnisstruktur muss nicht der Namespacestruktur entsprechen
- Java: Verzeichnisstruktur == Packagestruktur

⇒ Module/RootDirectory/Directory(*)/Component

© 2005-2017, hello2morrow

Abhängigkeiten und “Levels”

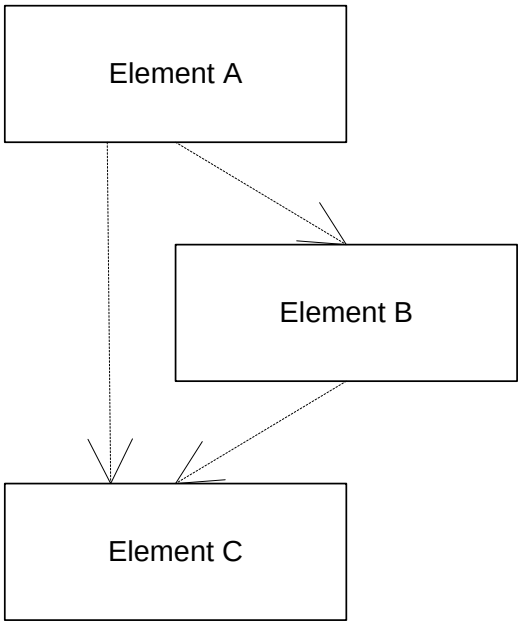


LEVEL

3

2

1



cyclic

acyclic

© 2005-2017, hello2morrow

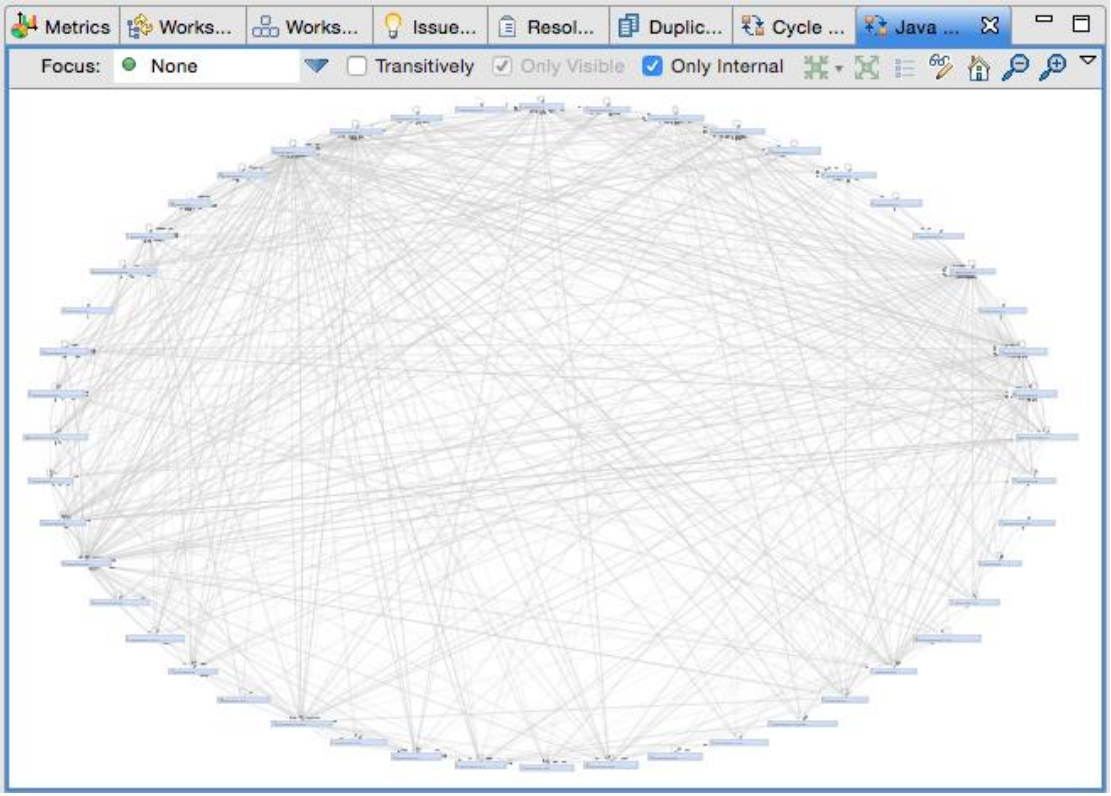
Negative Auswirkungen zyklischer Abhängigkeiten

- Testbarkeit
- Verständlichkeit
- Wiederverwendung
- Build/Versions Management
- Teambildung

“Cyclic physical dependencies in large, low-level subsystems have the greatest capacity to increase the overall cost of maintaining a system“ [John Lakos]

© 2005-2017, hello2morrow

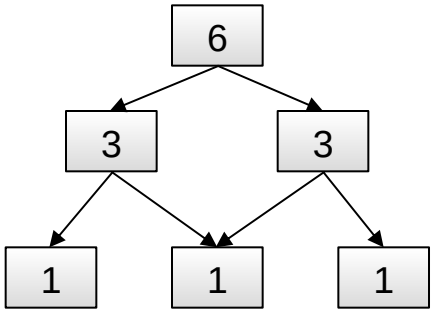
Eine kleine “entgleiste” Abhängigkeitsstruktur …



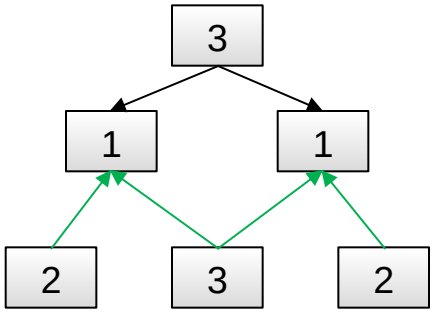
© 2005-2017, hello2morrow

Measure Coupling (John Lakos)

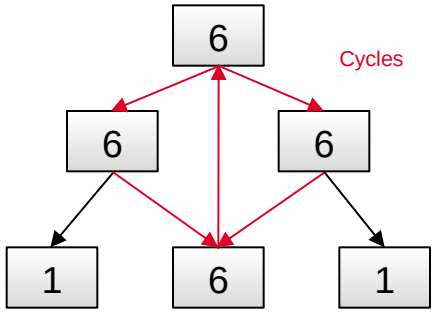
- Depends upon = The number a component directly and indirectly depends upon (+1 for itself)
- ACD (Average Component Dependency) = The sum of all depends upon values divided by the number of components



ACD = 15/6 = 2,5



Dependency Inversion
ACD = 12/6 = 2



ACD = 26/6 = 4,33

© 2005-2017, hello2morrow

Meine Empfehlungen zur Code Organisation

- Mindestens Zyklenfrei auf Ebene der Namespace/Packagestruktur
- Für C# und C++ zyklenfrei auch auf Ebene der Verzeichnisstruktur
- Ideal: Zyklenfrei auf Ebene der Components

© 2005-2017, hello2morrow

Was ist denn “logische” Architektur?

- Die identifizierbaren gröberen Blöcke (Cluster) und deren Abhängigkeiten
  - User Interface Code soll den Code der Domain Objekte benutzen und nicht umgekehrt
- Die Schlüsselmechanismen
  - Wie werden Änderungen an Domain Objekten zum User Interface notifiziert?
  - Wie wird der Zustand der Domain Objekte persistiert?

© 2005-2017, hello2morrow

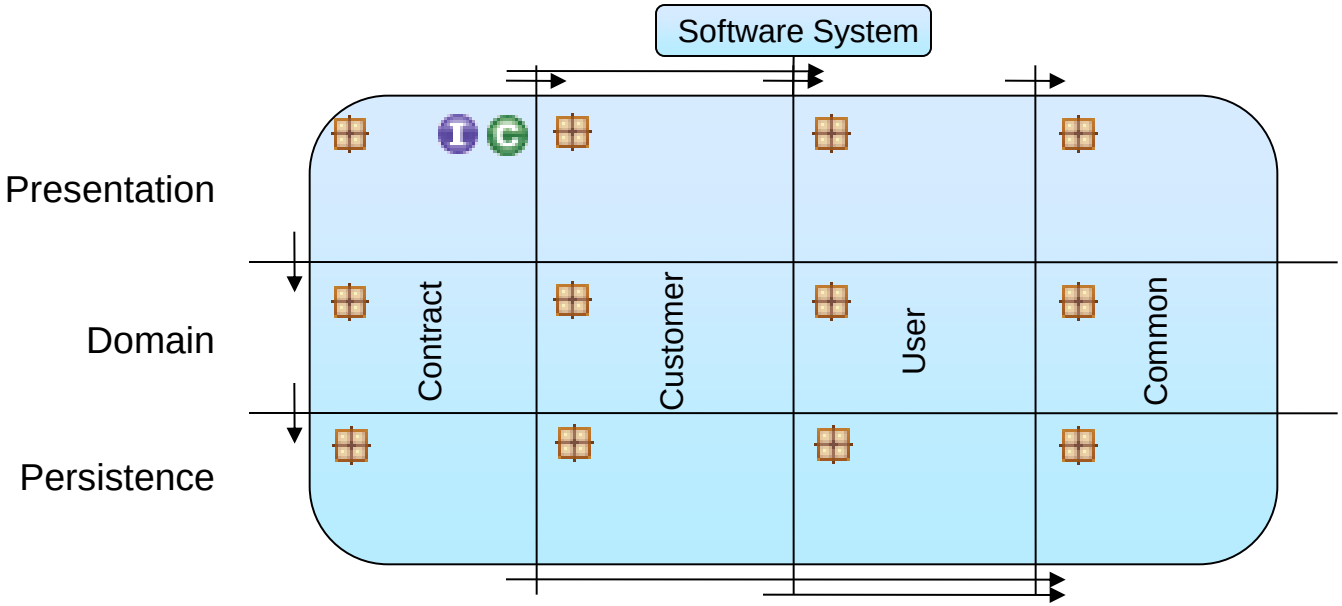
Wie komme ich zu einer logischen Architektur?

Um eine logische Architektur zu entwickeln, bietet es sich an folgende orthogonale Aspekte heranzuziehen:

- Fachlich: Was sind die fachlichen Blöcke, die man gegeneinander abgrenzen will?
- Technisch: Welche Blöcke haben abgrenzbare technische Verantwortung und benutzen (u.U.) unterscheidbare Technologien?

© 2005-2017, hello2morrow

Modellieren im Grid



- Schritt 1: Schneide horizontal technisch
- Schritt 2: Schneide vertikal fachlich
- Schritt 3: Definiere Abhängigkeiten (natürlich zyklenfrei)
- Schritt 4: Zuordnung von Components

© 2005-2017, hello2morrow

Weitere Aspekte …

Jede weitere (sinnvolle) Einschränkung von Abhängigkeiten hilft einen präziseren Rahmen für die Entwicklung zu haben. Dies könnte z.B. sein:

- Beziehungen zwischen Modulen
- Welche Beziehungen gibt es zu externen Bibliotheken?
- Welche Blöcke dürfen welche Abstraktionen verwenden, die Schlüsselmechanismen implementieren (z.B. welche Programmteile dürfen Abhängigkeiten zu unserem Event Framework haben)?

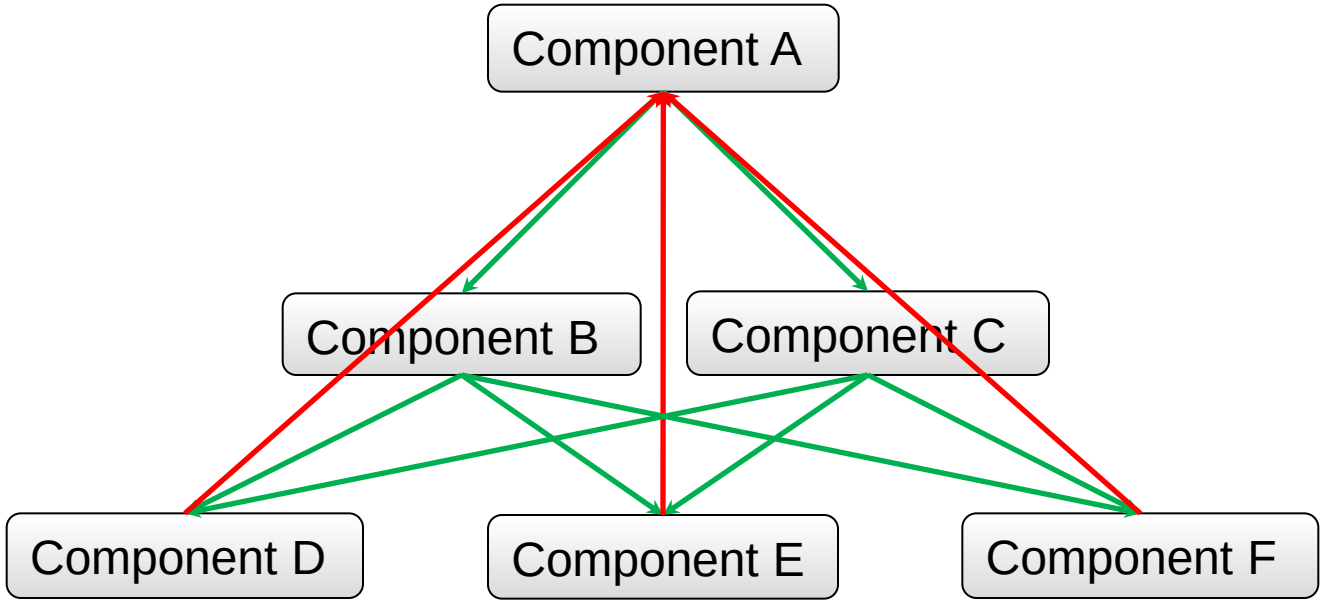
© 2005-2017, hello2morrow

Software Entropy (Lehman)

- Ein Computer Programm, das benutzt wird, wird modifiziert
- Wird ein Computer Programm modifiziert, wächst dessen Komplexität, wenn nicht aktiv etwas dagegen getan wird

© 2005-2017, hello2morrow

Komplexität nimmt zu wenn man nicht entgegensteuert



"It is the dependency architecture that is degrading, and with it the ability of the software to be maintained."
(Robert C. Martin)

© 2005-2017, hello2morrow

Technische Schulden

Category of TD	Repair Cost	Visible Impact	Maintainability Impact
Programming			
Testing			
Local/Global Metrics			
Architecture			

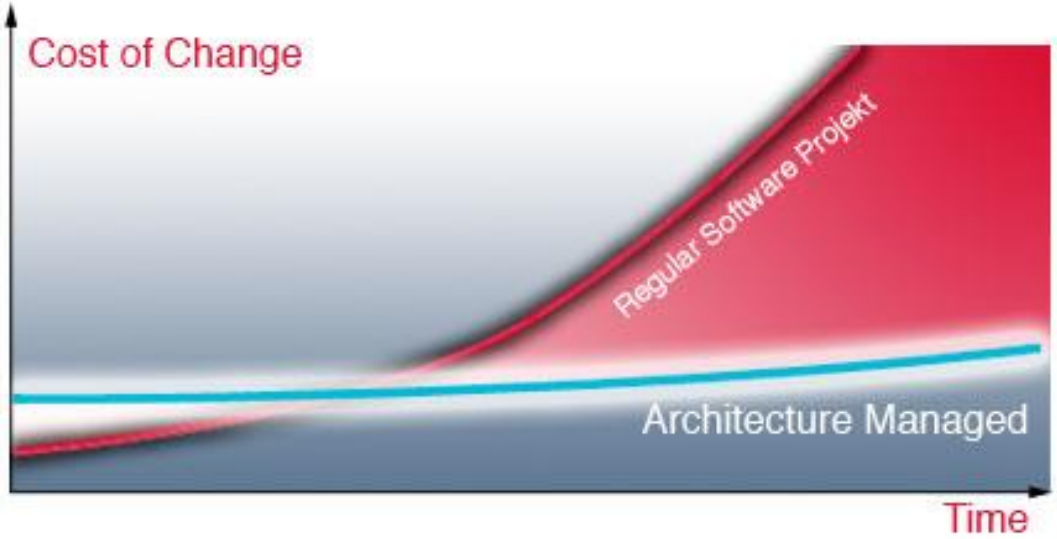
Zur Erinnerung

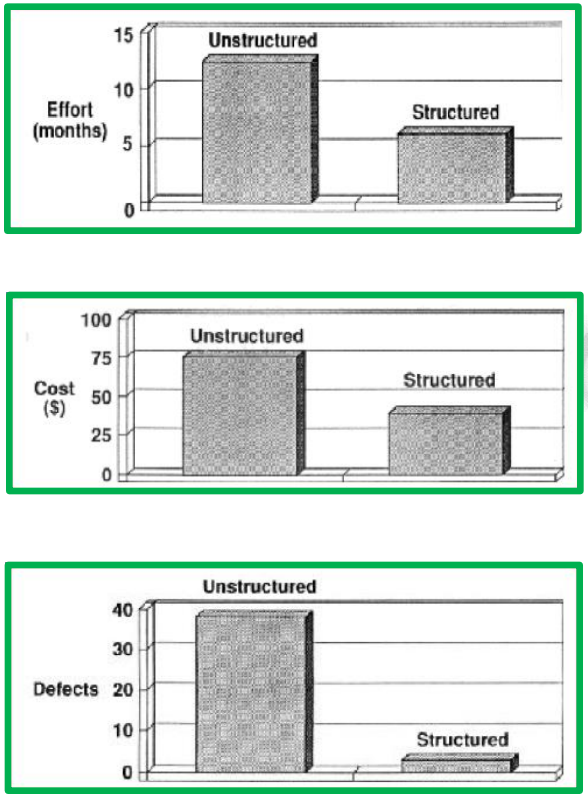
- Definiere den logischen Rahmen der Anwendung (keine Zyklen zwischen den logischen Blöcken)
- Definiere ein Abbildungsschema um die Zuordnung der Components vorzunehmen
- Erlaube keine Zyklen auf Namespace/Directory Ebene
- Vermeide Zyklen auf Component Ebene

$\Rightarrow$ Prüfe die Konformität automatisiert

© 2005-2017, hello2morrow

Positive Effekte



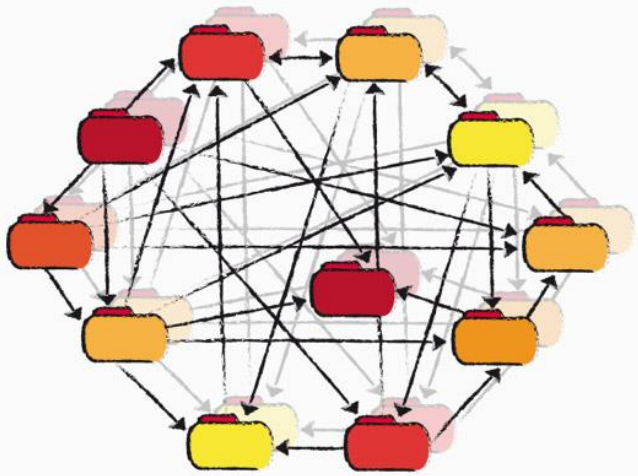


Barry M. Horowitz, DoD Study 1994

© 2005-2017, hello2morrow

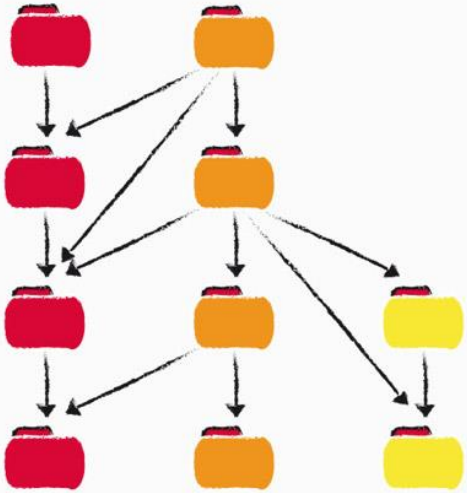
Because, if you don't…

Chances are your code looks like this:



- Much reduced team velocity
- Frequent regression bugs
- Hard to maintain, test and understand
- Modularization is impossible

Organized code looks more like this:

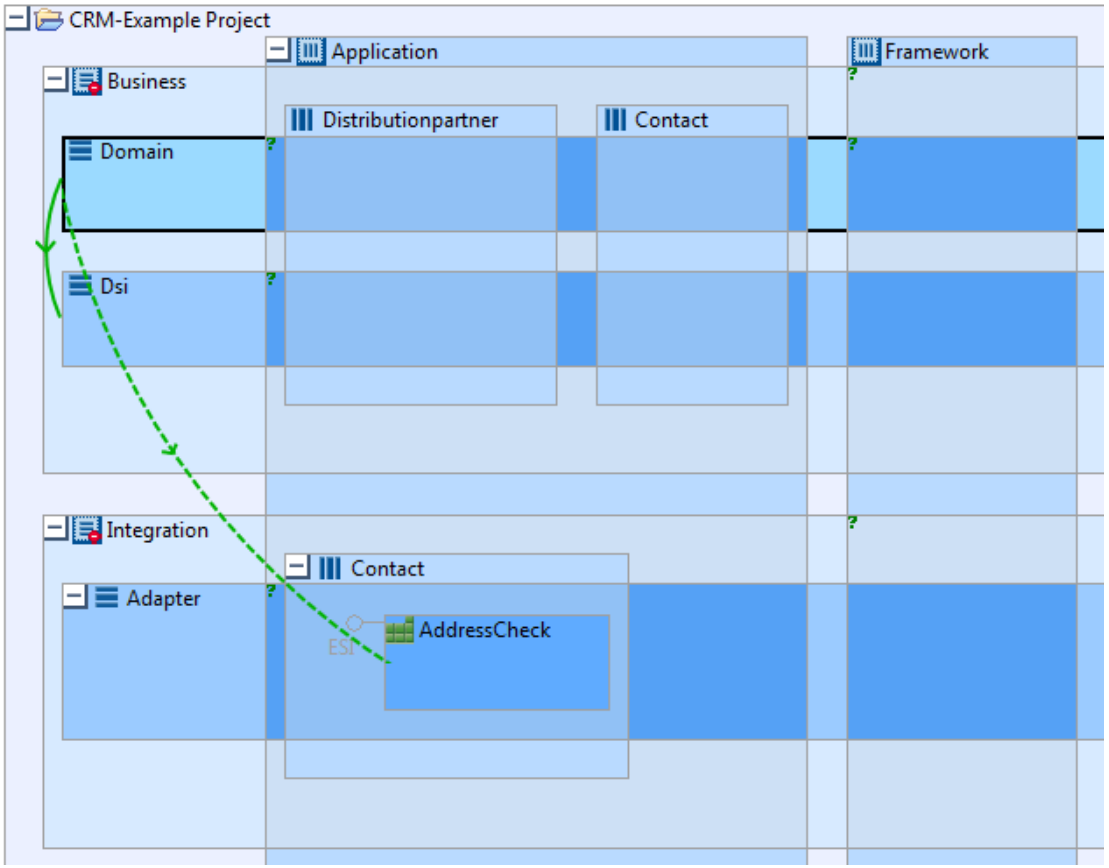


- Much lower cost of change
- Easier to maintain, test and understand
- Improved developer productivity
- Lower risk

© 2005-2017, hello2morrow

Sonargraph

Sonargraph 7.x Architektur Modell



© 2005-2017, hello2morrow

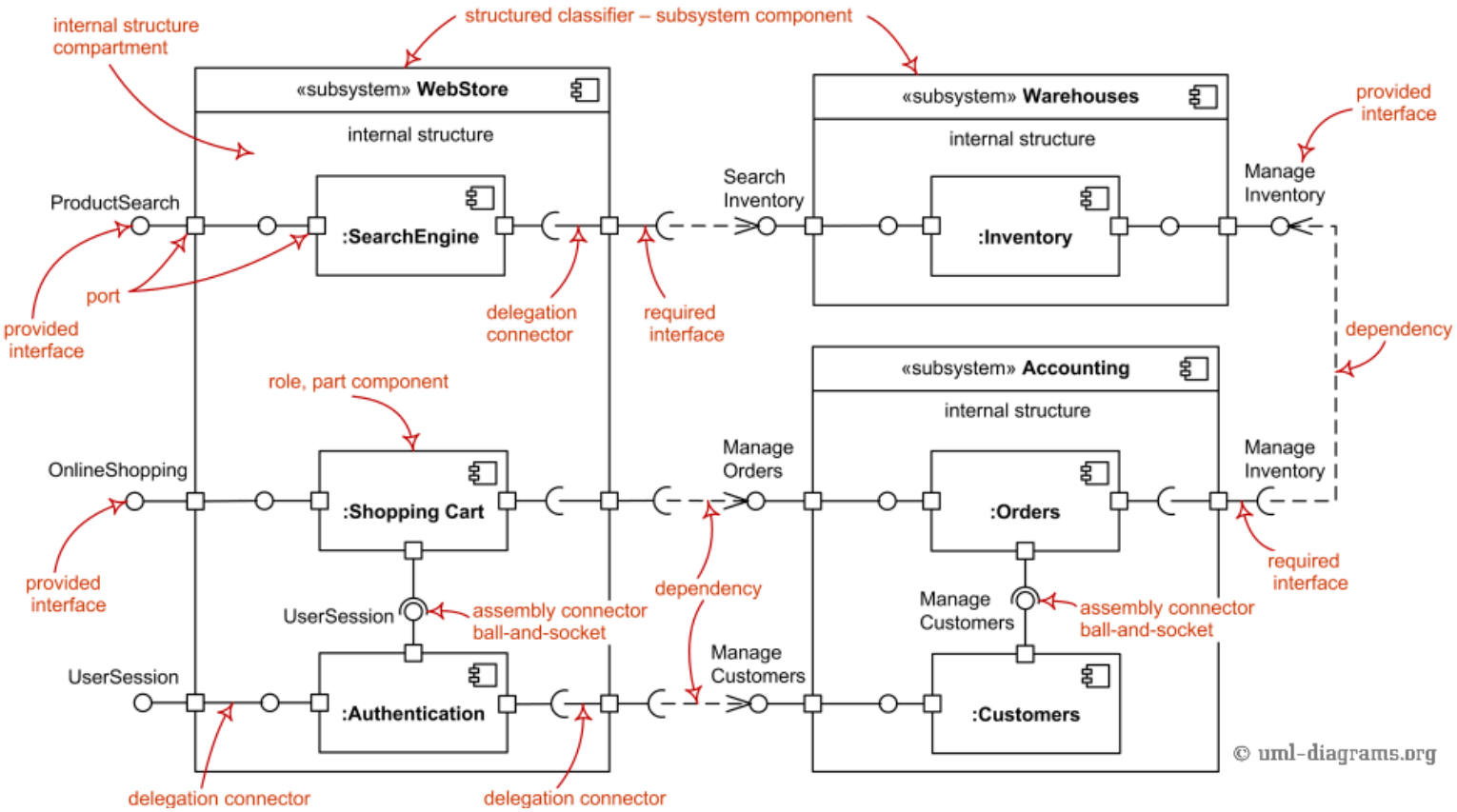
Einschränkungen (Sonargraph 7.x)

- Starres Modell mit vorgegebener Schachtelungstiefe
- Logische Architektur gebunden an Projektstruktur
- Monolithischer Ansatz – eine einzige Datei

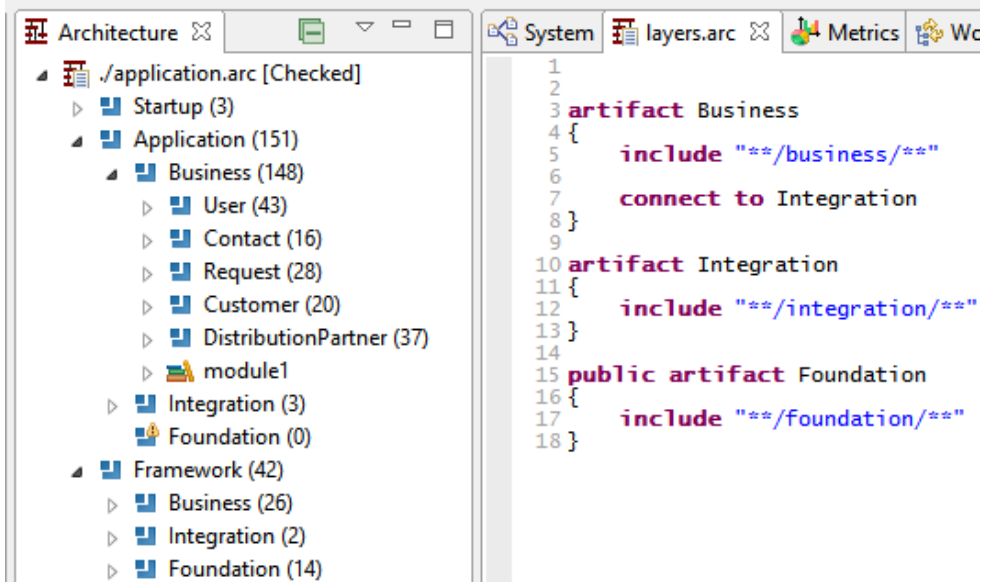
⇒ Wiederverwendung nicht möglisch
⇒ Kapselung nicht ausreichend unterstützt
⇒ Entstehen von komplexen Grafiken
⇒ Skaliert nicht

© 2005-2017, hello2morrow

UML Composite Structure Diagram



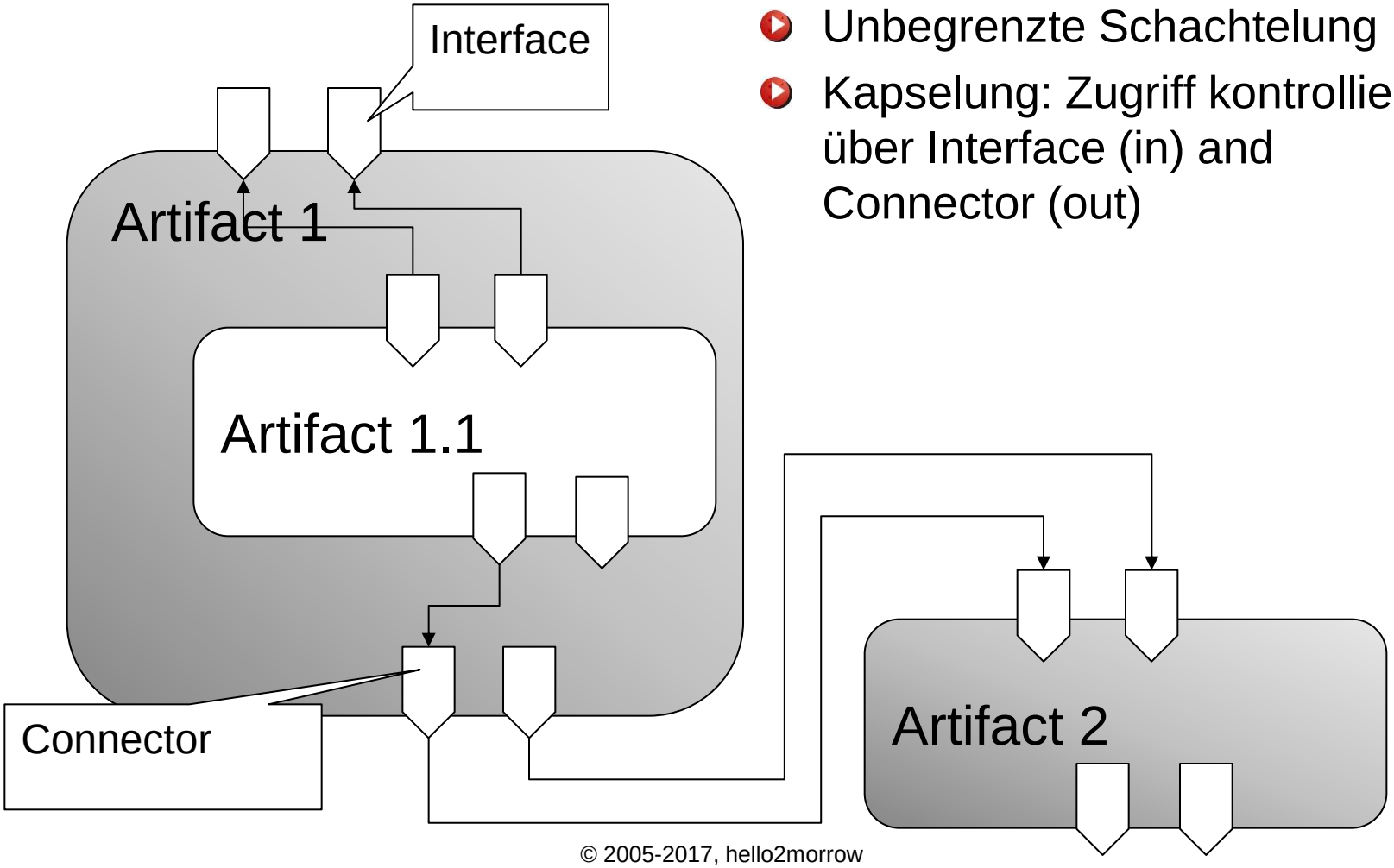
Sonargraph 8.x Architektur Modell



- Text basierte DSL zur Artifact Definition
- Anwenden von n Architektur Aspekten (n Dateien) möglich

© 2005-2017, hello2morrow

Artifacts



- Unbegrenzte Schachtelung
- Kapselung: Zugriff kontrolliert über Interface (in) and Connector (out)

Artifacts, Interfaces and Connectors (1)

- Artifacts match components
- A component can only be a child of the first matching artifact relative to the top-level architecture file
- Artifacts can have nested artifacts
- Artifacts have interfaces and connectors – only these are connected
- The default interface and connector contain all matched components
- The default interface and connector may be redefined by the user
- Additional interfaces and connectors may be defined

© 2005-2017, hello2morrow

Artifacts, Interfaces and Connectors (2)

```
artifact name
{
    // include and exclude filters
    // nested artifacts
    // interfaces and connectors
    // connections
}
interface iname
{
    // include and exclude filter
    // exported nested interfaces
}
connector cname
{
    // include and exclude filters
    // included nested connectors
}
```

NOTE: The order of the child elements is important!

© 2005-2017, hello2morrow

Adding Interfaces and Connectors

- With export you can include nested artifacts or interfaces of nested artifacts in an interface.
- The counterpart of export for connectors is the keyword include. It will include nested artifacts or connectors from nested artifacts in a connector.

```
artifact Business
{
    include "Core/**/business/**”
    hidden artifact Reflection
    {
        include "External*/*/java/lang/reflect/*"
    }
    interface Refl
    {
        export Reflection
    }
}
```

© 2005-2017, hello2morrow

Artifact's Definition and Component Matching

- '//' and '/* */' are used for comments

A component will only be assigned to the first matching artifact in the artifact hierarchy seen from the top-level architecture file.

- '*' matches everything except a slash
- '**' matches over slash boundaries
- '?' matches a single character

```
artifact Business
{
   include "Core/**/business/**" //all is a shortcut for "**"
   exclude "**/*java*" //excludes from included – only if needed
}
```

Note: To force a component match deep in the hierarchy use strong include

© 2005-2017, hello2morrow

Artifact's Stereotypes (1)

The following stereotypes exist and are mutually exclusive:

- strict – artifact may use only the artifact defined directly after itself
- relaxed – artifact may use all artifacts after itself
- unrestricted – artifact may use all artifact siblings

⇒ Using none of the above – allowed dependencies must be defined with connect to

NOTE: The order of the artifacts is important – when used with an artifact's stereotype strict or relaxed

© 2005-2017, hello2morrow

Artifact's Stereotypes (2)

- local – excluded from the default connector
- hidden – excluded from the default interface
- optional – can be empty without generating a warning message
- deprecated – supposed to be empty, generates a warning if not

© 2005-2017, hello2morrow

Weiterführende Informationen zur DSL

Die präsentierten DSL Features stellen nicht den kompletten Umfang dar. Für den kompletten Funktionsumfang sind folgende Quellen interessant:

- Architecture DSL Online Manual
- http://blog.hello2morrow.com

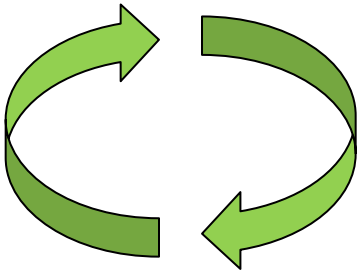
© 2005-2017, hello2morrow

Demo

Quality and Development Workflow

Role “Architect”

Defines the architecture, defines thresholds for coding metrics, identifies „hot spots“, creates tasks



Build Server

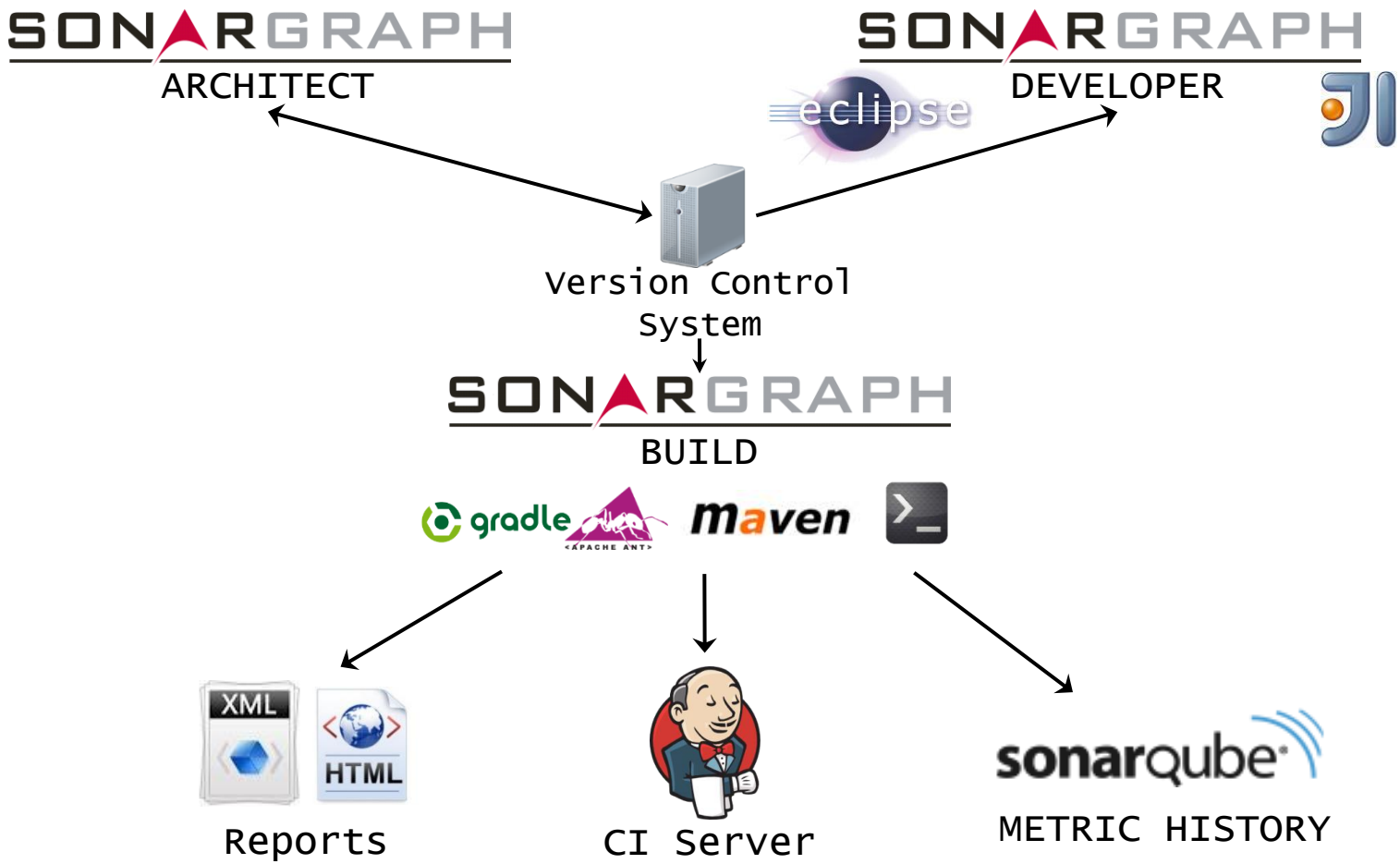
Continuously checks for architecture and threshold violations, cyclic dependencies, etc.

Role “Developer”

Implements use cases respecting the architecture and coding metrics thresholds

© 2005-2017, hello2morrow

Sonargraph Product Family



© 2005-2017, hello2morrow

Weitere Informationen zu Sonargraph

- Manuals/tutorials online:
  - Sonargraph Architect: http://eclipse.hello2morrow.com/doc/standalone/content
  - Sonargraph Build: http://eclipse.hello2morrow.com/doc/build/content/
- Whitepapers, DZone RefCard, etc auf unserer website www.hello2morrow.com
- Technische Fragen: support@hello2morrow.com
- Folgen sie uns: http://blog.hello2morrow.com/, Twitter: @sonargraph
- Kontaktieren sie uns: support@hello2morrow.com
- Kontaktieren sie mich: d.menges@hello2morrow.com

© 2005-2017, hello2morrow