How to Organize your Code

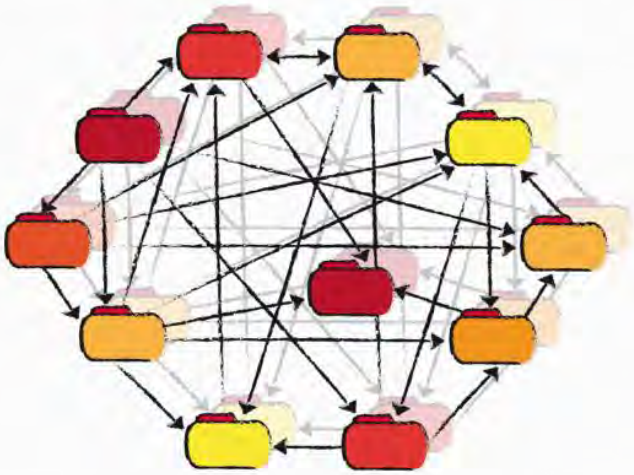
Alexander v. Zitzewitz
a.zitzewitz@hello2morrow.com
blog.hello2morrow.com
@AZ_hello2morrow

Code Organization Equals Dependency Management

The single best thing you can do for the long term health, quality and maintainability of a non-trivial software system is to carefully manage and control the dependencies between its different elements and components by defining and enforcing an architectural blueprint over its lifetime.

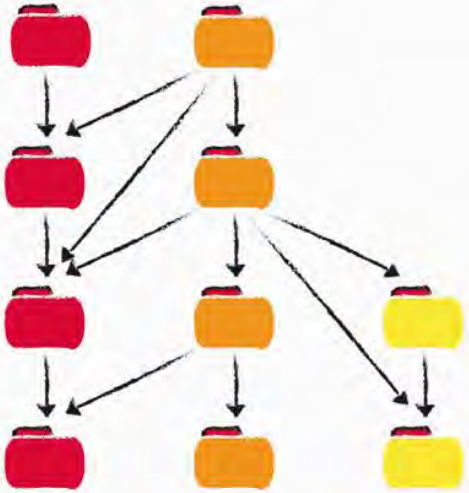
Because, if you don't…

Chances are your code looks like this:



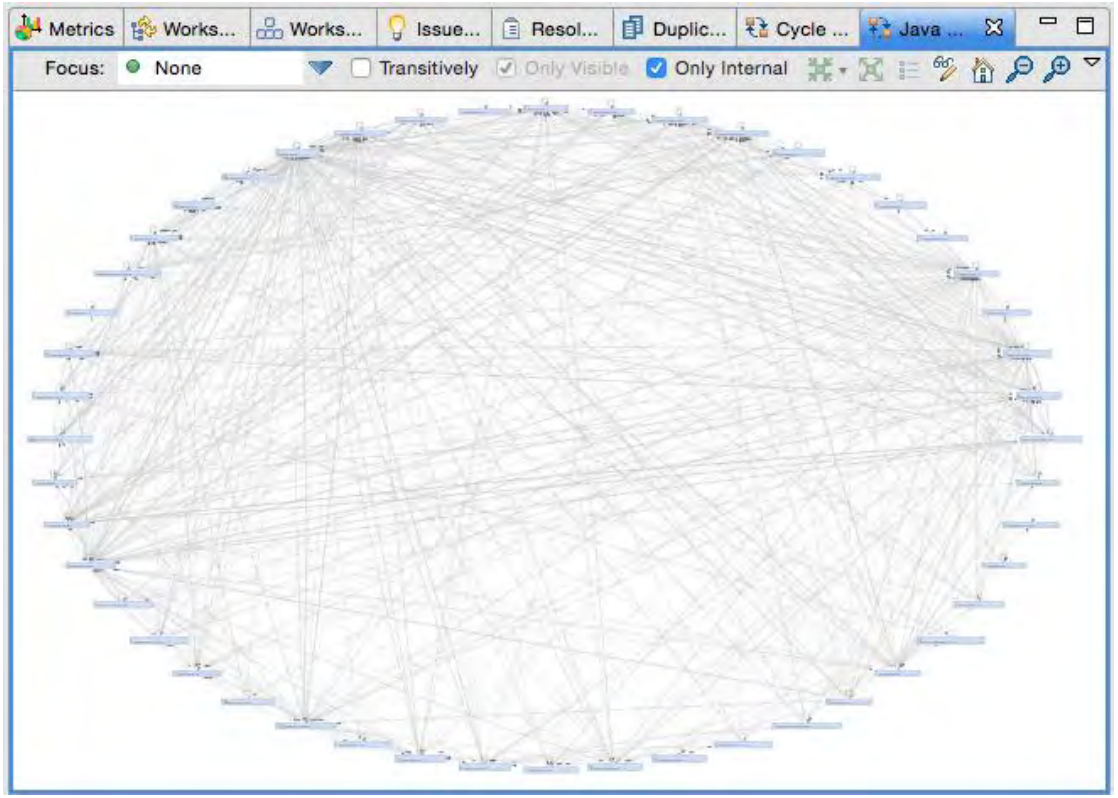
- Much reduced team velocity
- Frequent regression bugs
- Hard to maintain, test and understand
- Modularization is impossible

Organized code looks more like this:



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

Another architecture derailed…



Architecture of Apache-Cassandra (or what is left of it)

Reminds me of…

Why architectures tend to erode…

- Very hard to see from the perspective of the developer
- Software-Architects rarely use tools to visualize and manage dependencies
- If they even describe architecture, it is often informal (PowerPoint, Wiki etc.)
- That means it is hard to check conformity of code to architectural rules
- Rules that are not enforced will be broken
- Often there are no clearly defined quality and architecture standards that must be met for a software to be considered “done”.
- Agile projects consider architecture as a side effect of a user story
- Who has time for this??

Now add Micro-Services to this mix

- Splitting a messy monolith into Micro-Services will move the mess to the network layer
- Dependency management between services becomes even more important
- Avoid service loops – no cyclic dependencies between services
- We will need a way to visualize and restrict dependencies between services
- Static analysis can be useful here

Why architectural debt is so toxic

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

Agile Development and Architecture

- The agile approach does not automatically create maintainable and well architected systems. Often the opposite is true.
- Ongoing management of technical debt is considered to be a **critical success factor** for high quality and maintainable software systems even by promoters of the agile approach
- Architectural debt is a very toxic form of technical debt
- That challenges the idea that software development should almost exclusively be driven by business value
- Project size has obviously an important influence

Architectural Debt – 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*"

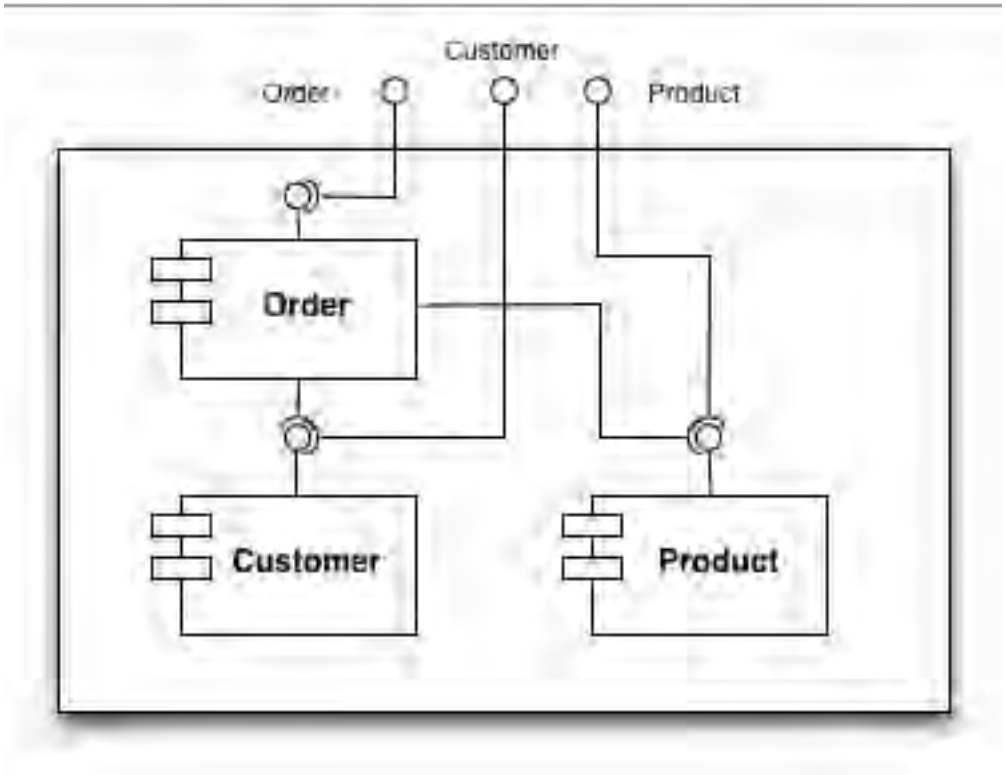
Do you manage Technical/Architectural Debt?

- Do you have binding rules for code quality?
- Do you measure quality rule violations on a daily base?
- Is your architecture defined in a formal way?
- Do you measure architecture violations on a daily base?
- Does quality management happen at the end of development?
- Does your current QM lead to sustainable results?
- Are there incentives for writing great code?

Code organization equals architecture

- Define your architecture in a formal and enforceable way (i.e. use a DSL to describe it)
- Use tools to check for violations of architecture rules, ideally with every commit or directly in the IDE
- Broken architecture rules have to be fixed while it is still easy to fix them
- Avoid cyclic dependencies between packages or higher level artifacts
- Invest about 20% of all development and maintenance effort into code hygiene and architecture

Example: Order Microservice



First step: think about package naming

Use functionality as top-level discriminator

com.hello2morrow.ordermanagement.order
com.hello2morrow.ordermanagement.customer
com.hello2morrow.ordermanagement.product

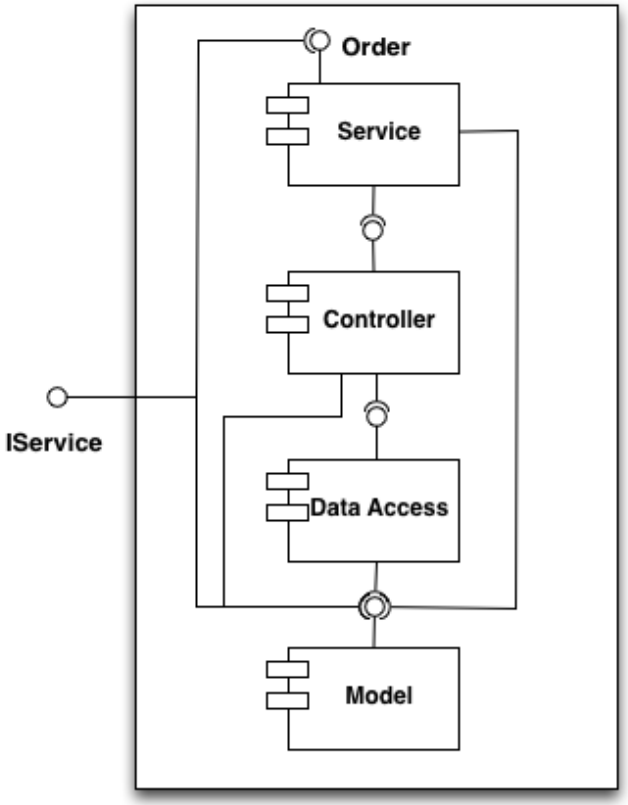
Step 2: High level architecture (in DSL)

```
artifact Order
{
    include "**/order/**"
    connect to Customer, Product
}

artifact Customer
{
    include "**/customer/**"
}

artifact Product
{
    include "**/product/**"
}
```

Step 3: Layering of major elements



Formal description of Layering:

```
// Layering.arc
artifact Service
{
    include "**/service/**"
    connect to Controller
}

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

require "JDBC"

artifact DataAccess
{
    include "**/data/**"
    connect to JDBC
}

public artifact Model
{
    include "**/model/**"
}

interface IService
{
    export Service, Model
}
```

Step 4: Putting everything together

```
artifact Order
{
    include "**/order/**"
    apply "layering"
    // Connect to the IService interface of Customer and Product
    connect to Customer.IService, Product.IService
}

artifact Customer
{
    include "**/customer/**"
    apply "layering"
}

artifact Product
{
    include "**/product/**"
    apply "layering"
}

// By using apply we define the artifacts of "JDBC" in this scope
apply "JDBC"
```

Final details

```
// JDBC.arc
artifact JDBC
{
    include "**/javax/sql/**"
}
```

Advantages of a DSL

- Easy to read and understand
- Works well with version control systems and can be diffed
- Can be changed without access to a tool
- More powerful than just drawing boxes
- Different aspects can be described in independent files
- Architecture diagrams can be generated
- Architecture files can be generated from diagrams

Components

- A component is the atomic element of architecture
- Usually a single source file, in C/C++ a combination of header and source files
- Is addressed via the relevant parts of its physical location

```
"Core/com/hello2morrow/Main"                          // Main.java in package com.hello2morrow
"External [Java]/[Unknown]/java/lang/reflect/Method"  // The Method class from java.lang.reflect
"NHibernate/Action/SimpleAction"                      // SimpleAction.cs in subfolder of NHibern
"External [C#]/System/System/Uri"                     // An external class from System.dll
```

- Patterns address groups of components

```
"Core/**/business/**"           // All components from the Core module with "business" in thei
"External*/*/java/lang/reflect/*" // All components in java.lang.reflect
```

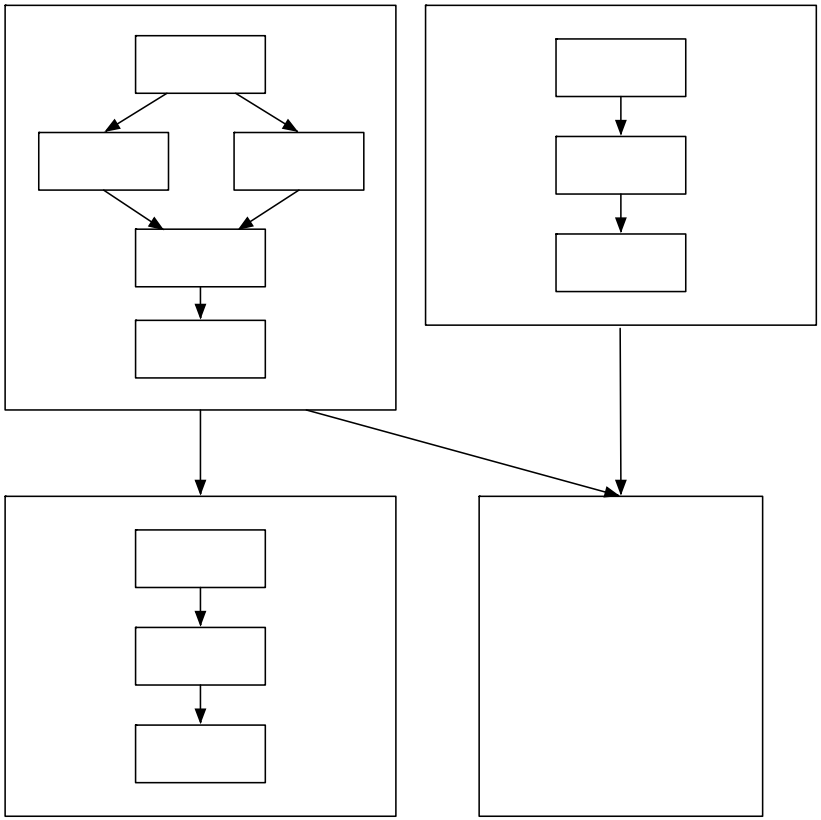
Artifacts

- Artifacts can contain components or other artifacts
- Artifacts have interfaces and connectors
- An interface is an incoming port granting access to a subset of components in artifact
- A connector is an outgoing port that can be connected to an interface of another artifacts
- Connections are only possible between connectors and interfaces
- Each artifact has a default connector and a default interface, both containing all components in the artifacts
- User can restrict the default connector and the default interface

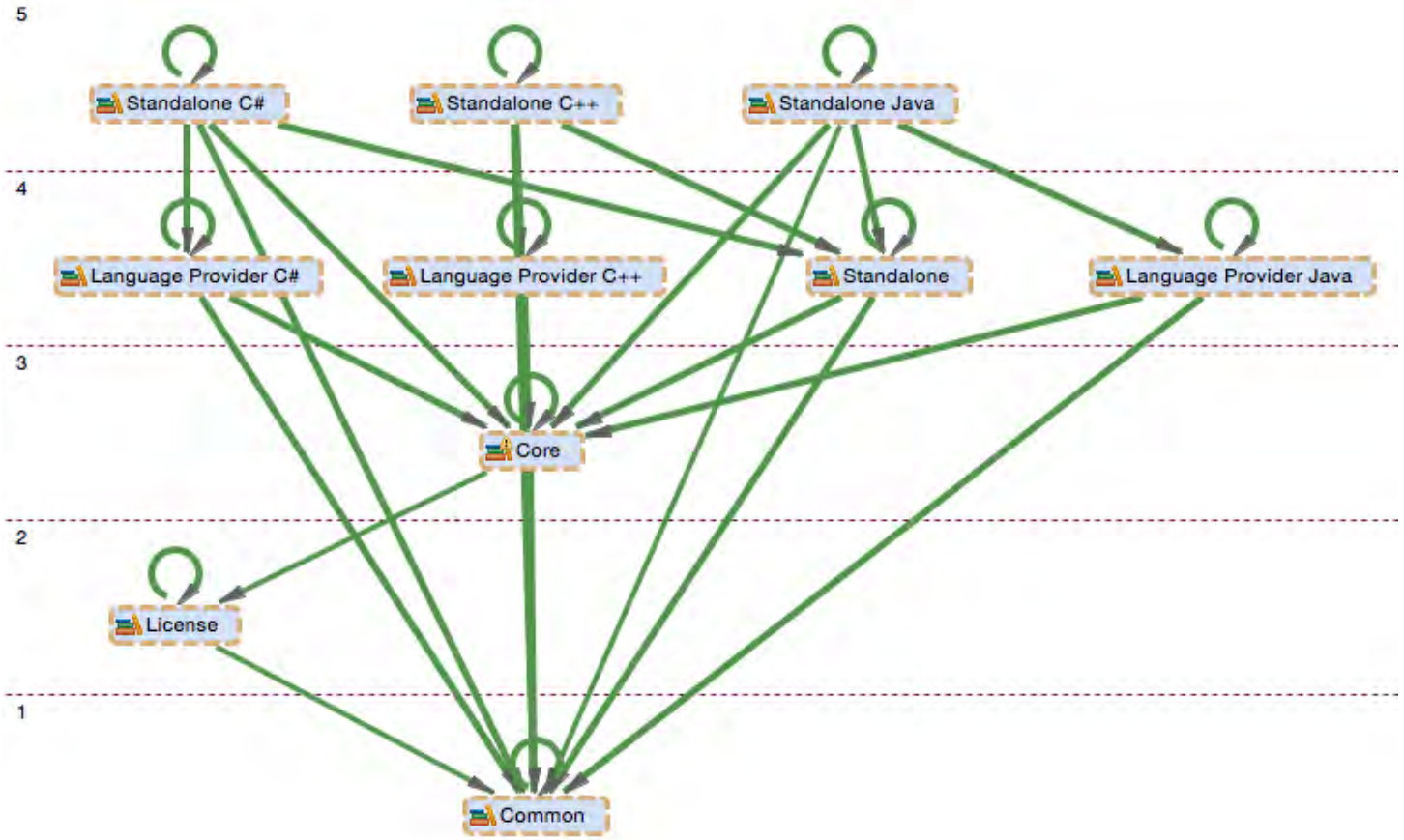
How to design a good architecture?

- Divide and conquer is your friend
- Try to split your system into not more than 10 top level elements
- Then split those elements internally
- Use patterns wherever possible
- Graphs with 7 or less elements are much easier to understand
- Therefore when splitting try to stick around 7 sub-elements or less
- The fewer allowed dependencies the better (increases flexibility)

Good architecture example



Another bigger example live



Q & A

a.zitzewitz@hello2morrow.com
blog.hello2morrow.com
@AZ_hello2morrow