Revealing Invisible Technical and Architectural Debt Quality Attributes

Alexander v. Zitzewitz

a.zitzewitz@hello2morrow.com

blog.hello2morrow.com

Invisible Quality Attributes

- The structure of your software, aka Architecture
- Software Metrics

They influence many things:

- How easy/difficult is it to comprehend your software
- Maintainability
- Testability
- Vulnerability through cyber threats

Understanding Technical Debt as a Metaphor

Ward Cunningham first defined the term in 1992:

"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite… The danger occurs when the debt is not repaid. Every minute spent on ***not-quite-right code*** *counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise."*

How to track technical debt

- Define rules that minimize the creation of “not quite right code”
  - Architecture rules and models
  - Metric based rules (thresholds)
  - Programming rules
  - Testing rules
- Count rule violations to measure your debt (automation needed)
- But how to weigh the rules?
- How to make sure that you’re not wasting time with irrelevant rules?

Categories of Technical Debt

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

Architectural Debt…

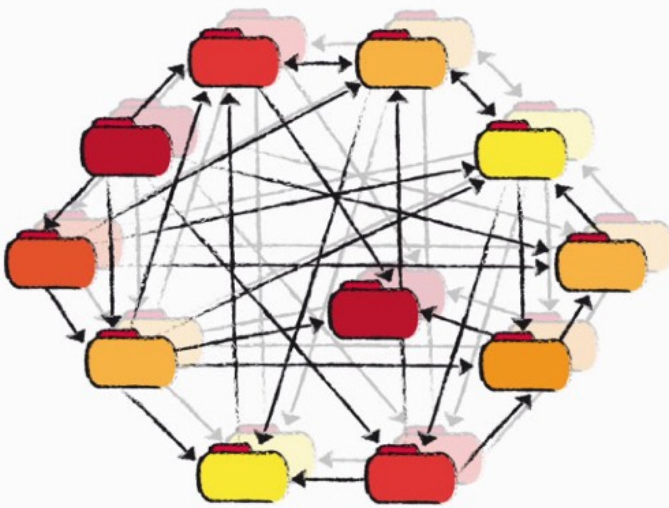
Is just a very toxic form of technical debt

Therefore avoiding it becomes crucial to keep software in good shape.

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.

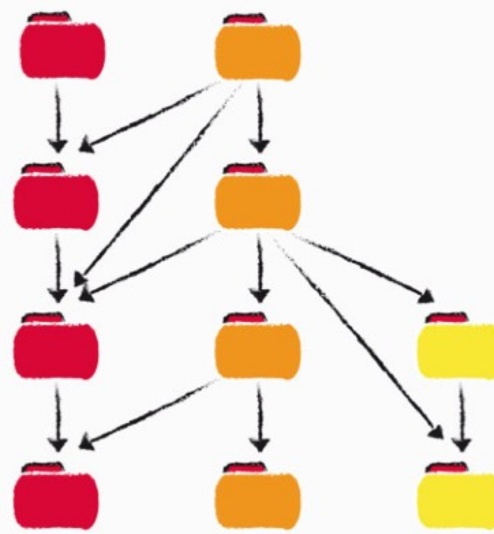
Because, if you don't…

CHANCES ARE YOUR CODE LOOKS LIKE THIS:



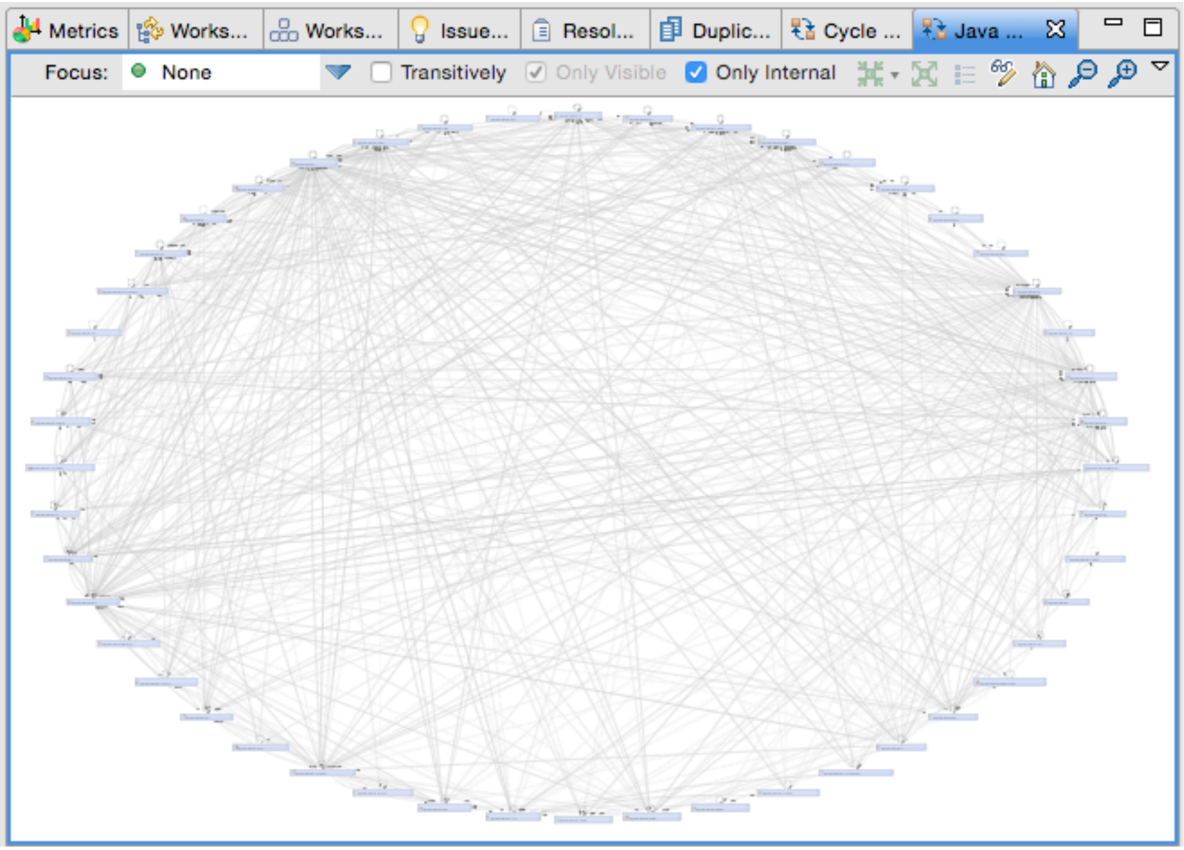
- Much reduced team velocity
- Frequent regression bugs
- Hard to maintain, test and understand
- Modularization is impossible

ORGANIZED CODE LOOKS MORE LIKE THIS:



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

Another example for 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
- Micro-Services increases complexity significantly (e.g. possible points of failure, network problems etc.)

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

Early Warning Metrics for Architectural Erosion

- Structural Debt Index
  - Can be computed for packages/namespaces or components (source files)
  - Number measures number of changes necessary to disentangle cyclic dependency groups
  - Cyclic dependencies are a good indicator for structural erosion
- ACD (Average Component Dependency, John Lakos)
  - Measures overall coupling
  - Value usually grows with system size
  - NCCD is a normalized version of this metric
- Those metrics can be computed by our free tool Sonargraph-Explorer
  - Can be integrated with SonarQube/Jenkins

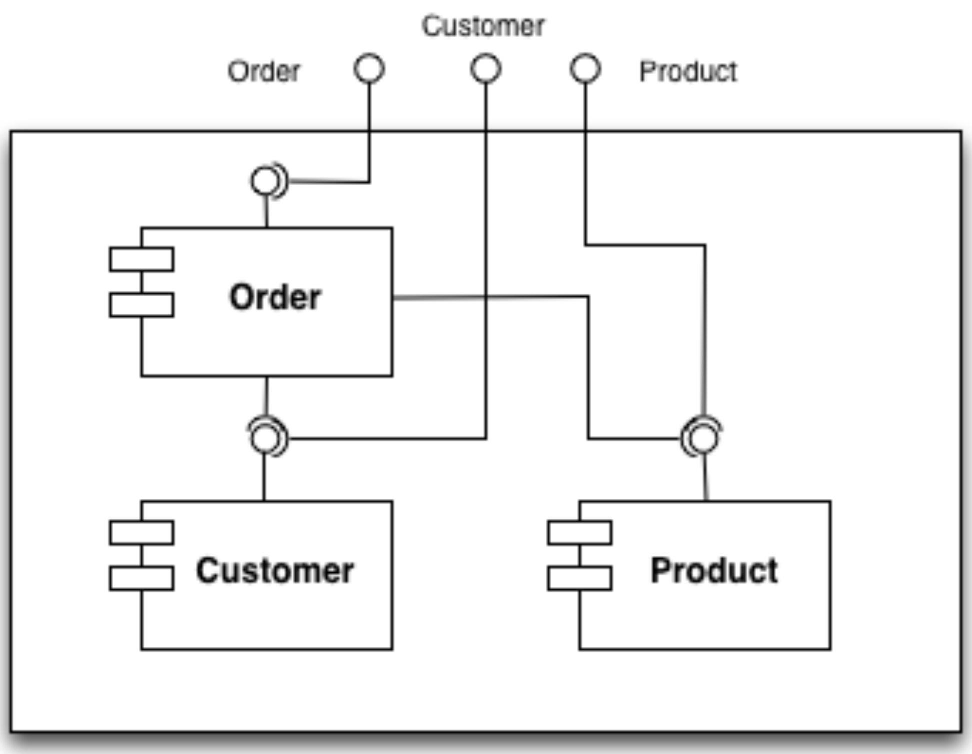
Do you manage Technical/Architectural Debt?

- 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?

How to enforce an Architectural Model

- You need a formal machine readable version of your architectural model
- This allows a tool based approach where the tool can warn developers in their IDE or break the build when architecture violations are introduced

Example: Order System



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/**"
}
```

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
```

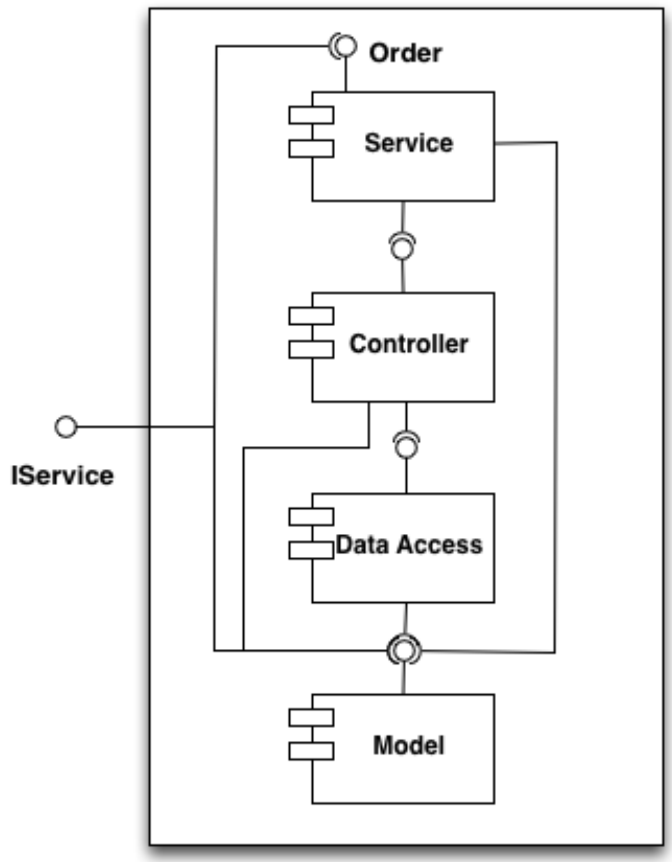
- Patterns address groups of components

```
"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

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
}
```

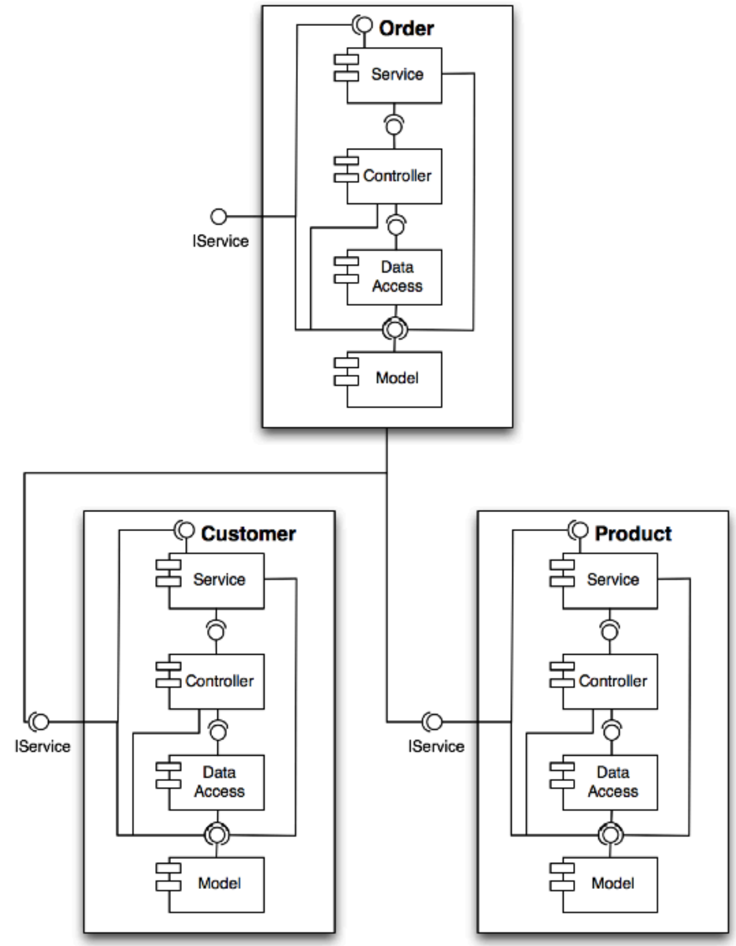
Step 4: Putting everything together

```
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/**"
}
```

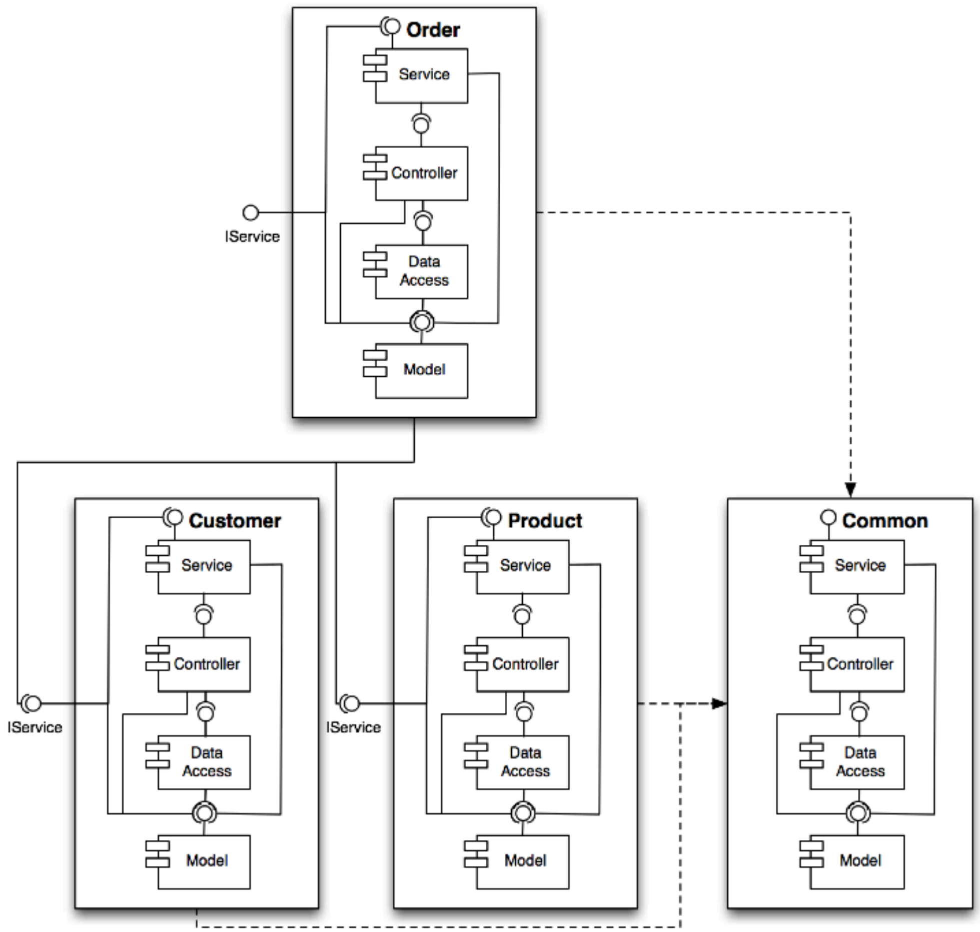
Where do modules fit?



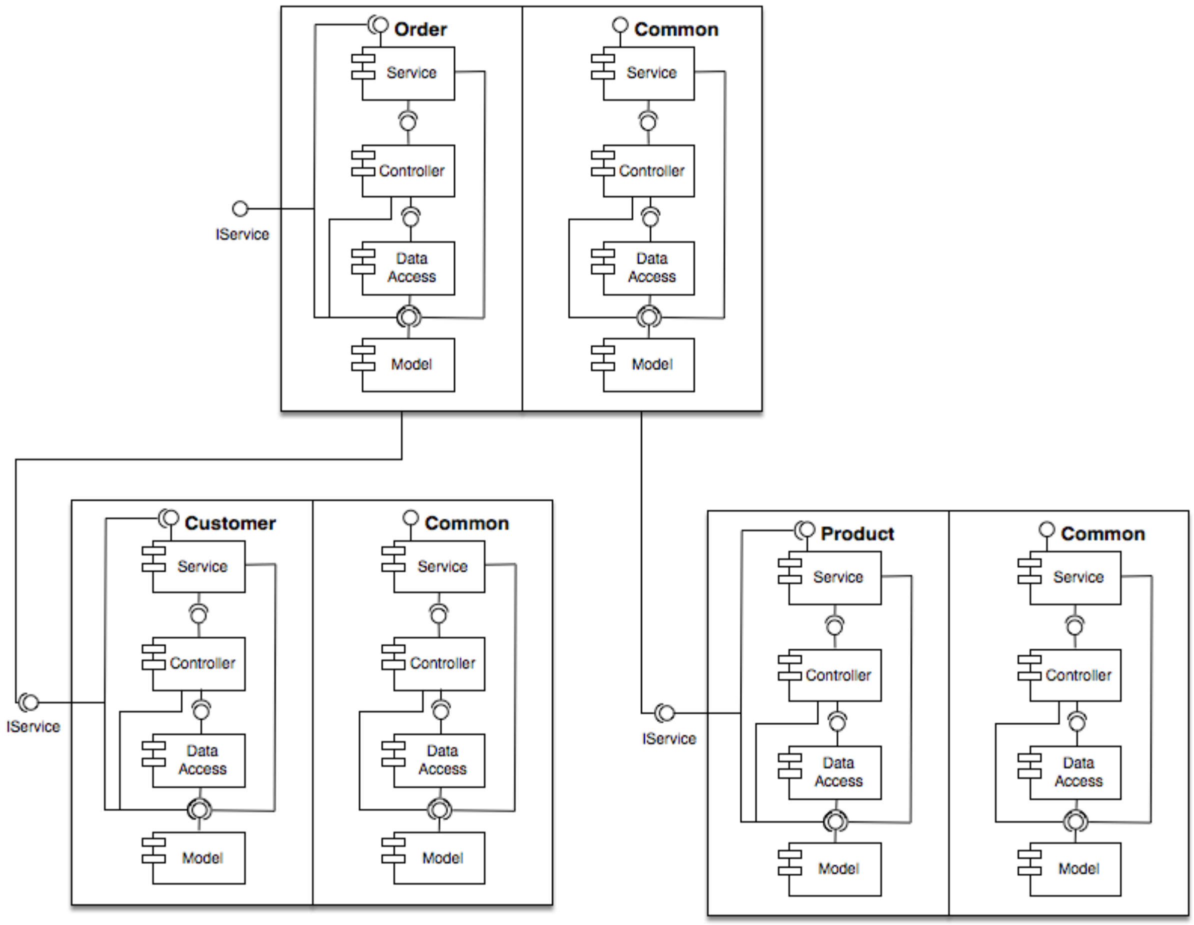
Modules and Architecture

- Modules structure should reflect the topmost level of your architecture
- Limit number of modules to dozen’s at the most
- 100’s of modules always create new problems
- Modules are very limited when it comes to enforce architectural restrictions – no nesting
- Maven is not a module system and is not able to enforce architectural descriptions

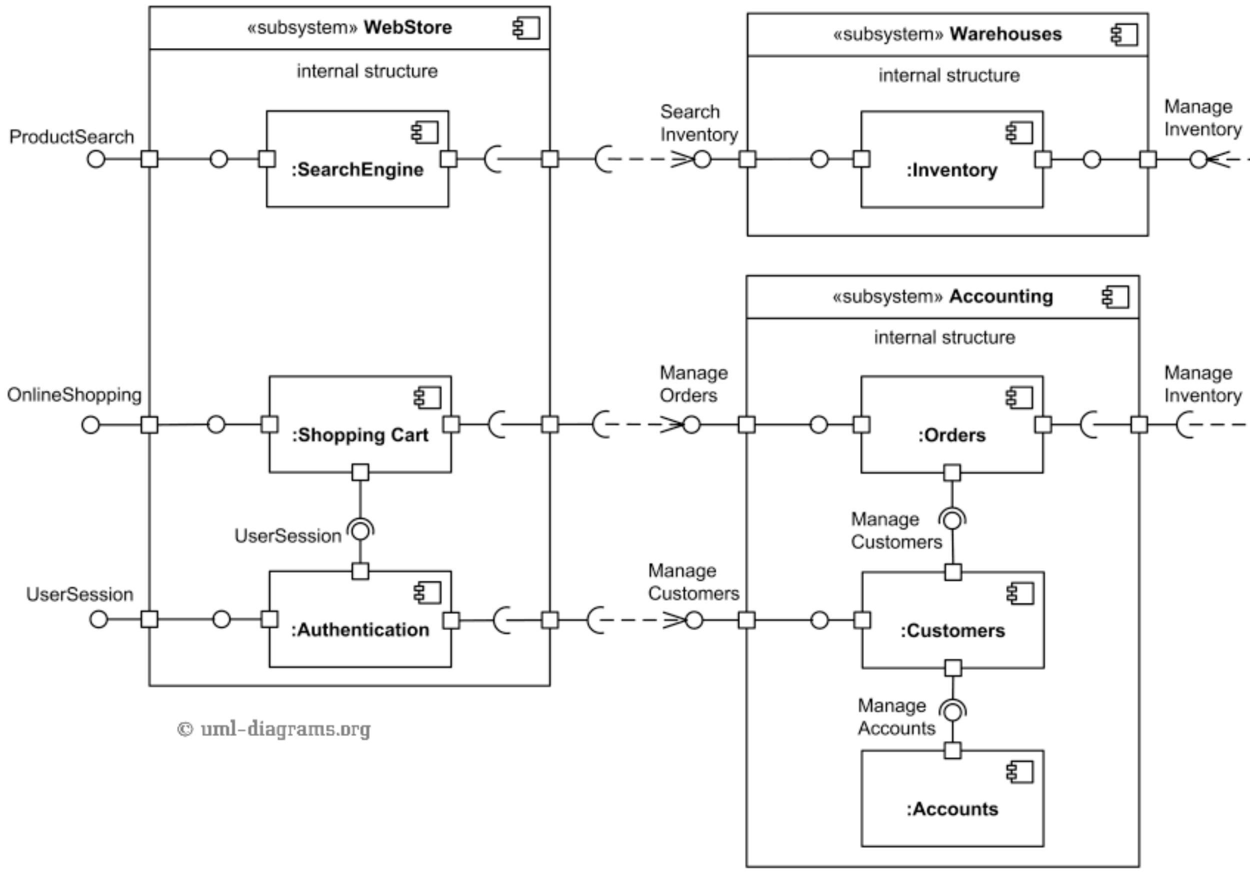
Architecture Pattern: Smart Monolith



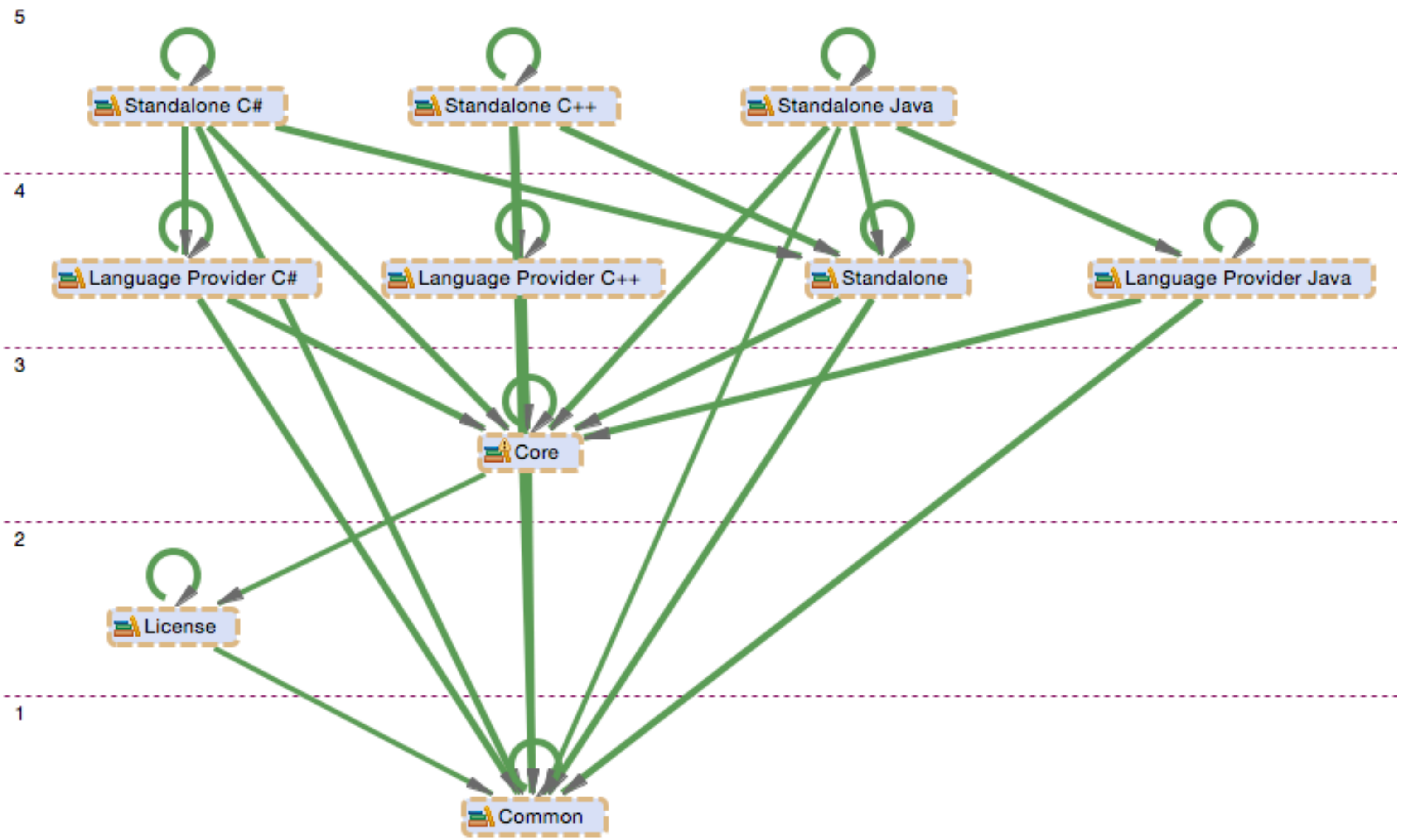
Same Thing as Micro-Services



DSL Implements UML Component Diagrams



Another bigger example live



Q & A

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