METHODS & TOOLS

Practical knowledge for the software developer, tester and project manager **ISSN 1661-402X**

Winter 2015 (Volume 23 - number 4) www.methodsandtools.com

Successful Project Failures

Software development projects have always been challenging. The most referenced source for metrics about project success is the Standish Group yearly CHAOS report. According to this source, around 30% of software development project were considered as a "failure" in the recent years. The failure rate of larger projects is a little bit higher than the average and the rate of failure of Agile projects a slightly lower than average.

Initially, some people thought that Agile would the "cure" for all software development projects issues. I remember reading that if an Agile project failed, it was because it wasn't truly Agile. The fact is that one of the key benefits of Agile should be a quicker feedback loop. It will however not remove the fact that you might be working with unskilled product owners or developers, that the chosen technology is not adequate or that the expected value of the project doesn't materialize. Even Google that is supposed to employ some of the brightest people in the technology area can produce a project like Google Glass that went from hyperhype to silent death when it hit the real world.

For small organizations with limited resources, measuring value creations and detecting issues or failures quickly is a matter of survival. This is however not necessary the case in larger organizations where failures are still mainly considered as bad things and generate blame. The goal of the project manager is to deliver results, not to kill projects. We can still dream of a software development world where you would be proud of having on your resume the fact that you stopped a project after two months and saved your company a lot of money. Most people will only consider the money that was spent for "nothing". There are however many cases where admitting failure could be the greatest success for a software development project.

I wish all the Methods & Tools readers and their family a healthy 2016. I would also like to thank the authors and the advertisers for their contribution and their support that help Methods & Tools to start with confidence its 24th year of existence. You can help us also us as a reader by telling your colleagues or social network relationship that you have read something interesting in Methods & Tools.

[illegible]

Inside

Meet the Social Side of Your Codebase

Adam Tornhill, @AdamTornhill, http://www.adamtornhill.com/

Let’s face it - programming is hard. You could spend an entire career isolated in a single programming language and still be left with more to learn about it. And as if technology alone weren’t challenging enough, software development is also a social activity. That means software development is prone to the same social biases that you meet in real life. We face the challenges of collaboration, communication, and team work.

If you ever struggled with these issues, this article is for you. If you haven’t, this article is even more relevant. The organizational problems that we will discuss are often misinterpreted as technical issues. So follow along, learn to spot them, react, and improve.

Know Your True Bottlenecks

Some years ago I did some work for a large organization. We were close to 200 programmers working on the same system. On my first day, I got assigned to a number of tasks. Perfect! Motivated and eager to get things done, I jumped right in on the code.

I soon noticed that the first task required a change to an API. It was a simple, tiny change. The problem was just that this API was owned by a different team. Well, I filed a change request and walked over to their team lead. Easy, he said. This is a simple change. I’ll do it right away. So I went back to my desk and started on the next task. And let me tell you: that was good because it took one whole week to get that “simple” change done!

I didn’t think much about it. But it turned out that we had to modify that shared API a lot. Every time we did, the change took at least one week. Finally I just had to find out what was going on - how could a simple change take a week? At the next opportunity, I asked the lead on the other team. As it turned out, in order to do the proposed change, he had to modify another API that was owned by a different team. And that team, in turn, had to go to yet another team which, in turn, had the unfortunate position of trying to convince the database administrators to push a change.

No matter how agile we wanted to be, this was the very opposite end of that spectrum. A simple change rippled through the whole organization and took ages to complete. If a simple change like that is expensive, what will happen to larger and more intricate design changes? That is right - they’ll wreak havoc on the product budget and probably our codebase and sanity too. You don’t want that, so let’s understand the root causes and see how you can prevent them.

Ranorex Automated Testing Tool - Click on ad to reach advertiser web site

Ranorex®

Automated Testing of

Desktop. Web. Mobile.

Robust Automation

Broad Acceptance

Seamless Integration

Quick ROI

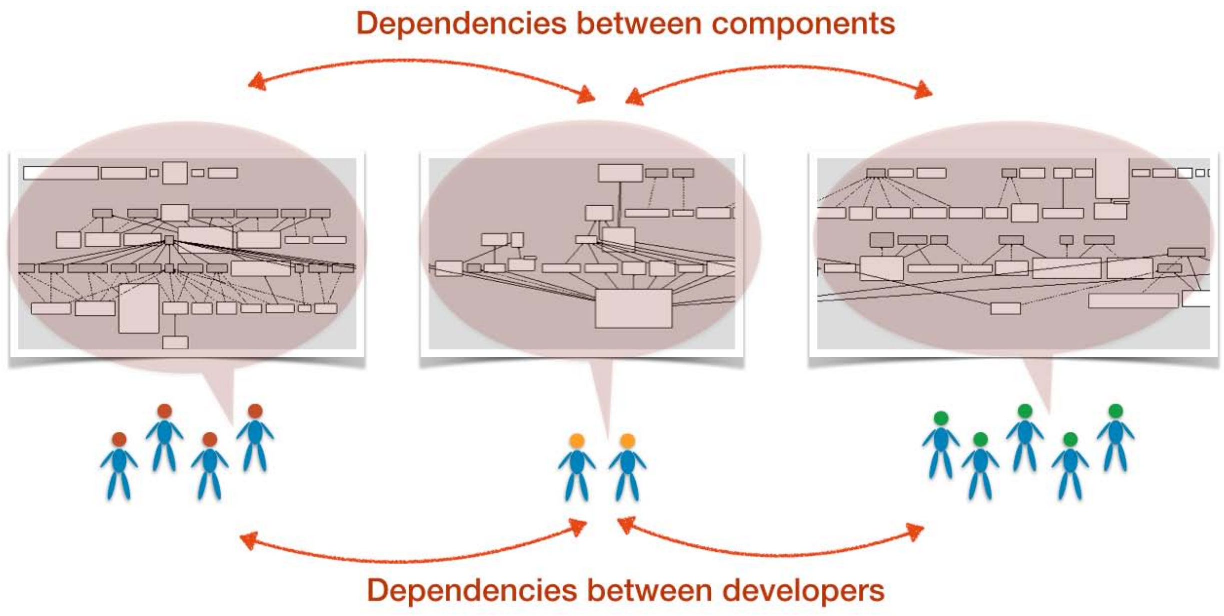
Why Use Ranorex

www.ranorex.com/why

Award-winning test automation tools provide seamless testing of a wide range of desktop, web and mobile applications.
Android | iOS | HTML5 | IE | FF | Chrome | Safari | WPF | Flash/Flex | Silverlight | Qt | SAP | .NET | MFC | Delphi | 3rd Party Controls | Java

Understand the Intersection between People and Code

In the story I just told, the problem wasn't the design of the software, which was quite sound. The problem was an organization that didn't fit the way the system was designed.


Dependencies between components
Dependencies between developers

When we have a software system whose different components depend upon each other and those components are developed by different programmers, well, we have a dependency between people too. That alone is tricky. The moment you add teams to the equation, such dependencies turn into true productivity bottlenecks accompanied by the sounds of frustration and miscommunication.

Such misalignments between organization and architecture are common. Worse, we often fail to recognize those problems for what they are. When things go wrong in that space, we usually attribute it to technical issues when, in reality, it is a discrepancy between the way we are organized versus the kind of work our codebase supports. These kinds of problems are more severe since they impact multiple teams and it is rare that someone has a holistic picture. As such, the root cause often goes undetected. That means we need to approach these issues differently. We need to look beyond code.

Revisit Conway's Law

This common problem of an organization that is misaligned with its software architecture is not new. It has haunted the software industry for decades. In fact, it takes us all the way back to the 60's to a famous observation about software: Conway's Law. Conway's Law basically claims that the way we are organized will be mirrored in the software we design. Our communication structure will be reflected in the code we write. Conway's Law has received a lot of attention over the past years, and there are just as many interpretations of it as there are research papers about it. To me, the most interesting interpretation is Conway's Law in reverse. Here we start with the system we are building: given a proposed software architecture, what is the optimal organization to develop it efficiently?

When interpreted in reverse like that, Conway's Law becomes a useful organizational tool. But, most of the time we are not designing new architectures. We have existing systems that we keep maintaining, improving, and adding new features to. We are constrained by our existing architecture. How can we use Conway's Law on existing code?

SpiraTeam Complete Agile ALM Suite - Click on ad to reach advertiser web site

Are You Tired of Having Separate Tools for Requirements, Testing, Bug-Tracking and Planning?

It's time to try a better way.

spiraTeam®

The most complete yet affordable Agile ALM suite on the market today.

Learn more at: inflectra.com/spiraTeam

www.inflectra.com
sales@inflectra.com
+1 202-558-6885

inflectra®

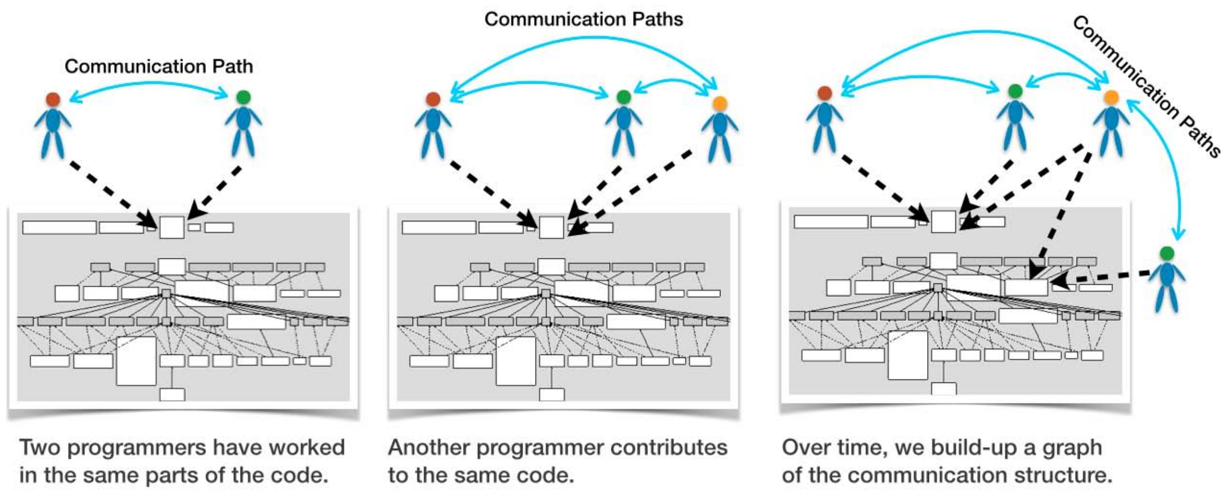
Let Conway's Law Guide You on Legacy Systems

To apply Conway's Law to legacy code, your first step is to understand the current state of your system. You need to know how well your codebase supports the way you work with it today. It is a tricky problem - we probably don't know what our optimal organization should look like. The good news is that your code knows. Or, more precisely, its *history* knows.

Yes, I'm referring to your version-control data. Your version-control system keeps a detailed log of all your interactions with the codebase. That history knows which developers contributed, where they crossed paths, and how close to each other in time they came by. It is all social information, but we are just not used to thinking about version-control data in that way. That means that we can mine our source code repositories to uncover hidden communication structures. Let's see how that looks.

Uncover Hidden Communication Paths

According to Conway [1], a design effort should be organized according to the need for communication. This gives us a good starting point when reasoning about legacy systems. Conway's observation implies that any developers who work in the same parts of the code need to have good communication paths. That is, if they work with the same parts of the system, the developers should also be close from an organizational point of view.


Communication Path
Communication Paths
Communication Paths
Two programmers have worked in the same parts of the code.
Another programmer contributes to the same code.
Over time, we build-up a graph of the communication structure.

As you see in the figure above, we follow a simple recipe. We scan the source code repository and identify developers who worked on the same modules:

1. Every time two programmers have contributed to the same module, these developers get a communication link between them.
2. If another programmer contributes to the same code, she gets a communication link as well.
3. The more two programmers work in the same parts of the code, the stronger their link.

Once you've scanned the source code repository, you'll have a complete graph over the *ideal* communication paths in your organization. Note the emphasis on *ideal* here. These communication paths just show what *should* have been. There is no guarantee that these communication paths exist in the real world. That is where you come in. Now that you have a picture over the ideal communication paths, you want to compare that information to your real, formal organization. Any discrepancies are a signal that you may have a problem. So let's have a look at what you might find.

YouTrack Issue Tracker - Click on ad to reach advertiser web site

YT

YouTrack

ISSUE TRACKER DESIGNED FOR DEVELOPMENT TEAMS

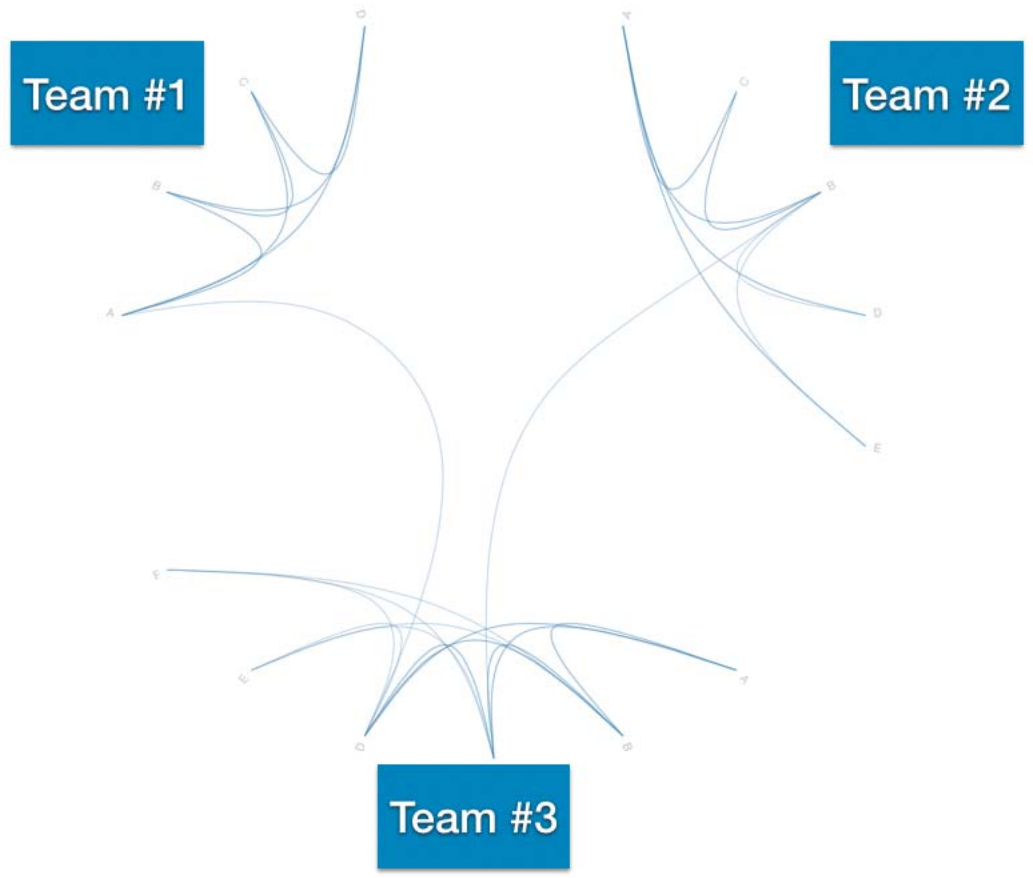
Free fo 10 users

Get it now

JET BRAINS

Know the Communication Paths Your Code Wants

In the best of all worlds, you'll be close to what Conway recommended. Have a look at the following figure. It shows one example of an ideal communication diagram:



The example in the figure above illustrates three teams. You see that most of the communication paths go between members of the same teams. That is a good sign. Let's discuss why.

Remember that a communication diagram is built from the evolution of your codebase. When most paths are between members on the same team, that means the team members work on the same parts of the code. Everyone on such a team has a shared context, which makes communication easier.

As you see in the picture above, there is the occasional developer who contributes to code that another team works on (note the paths that go between teams). There may be different reasons for those paths. Perhaps they're hinting at some component that is shared between teams. Or perhaps it is a sign of knowledge spread: while cohesive teams are important, it may be useful to rotate team members every now and them. Cross-pollination is good for software teams too. It often breeds knowledge.

The figure above paints a wonderful world. A world of shared context with cohesive teams where we can code away on our tasks without getting in each other's way. It is the kind of communication structure you want.

However, if you haven't paid careful attention to your architecture and its social structures you won't get there. So let's look at the opposite side of the spectrum. Let's look at a disaster so that you know what to avoid.

A Man-month is Still Mythical

About the same time as I started to develop the communication diagrams, I got in contact with an organization in trouble. I was allowed to share the story with you, so read on - what follows is an experience born in organizational pain.

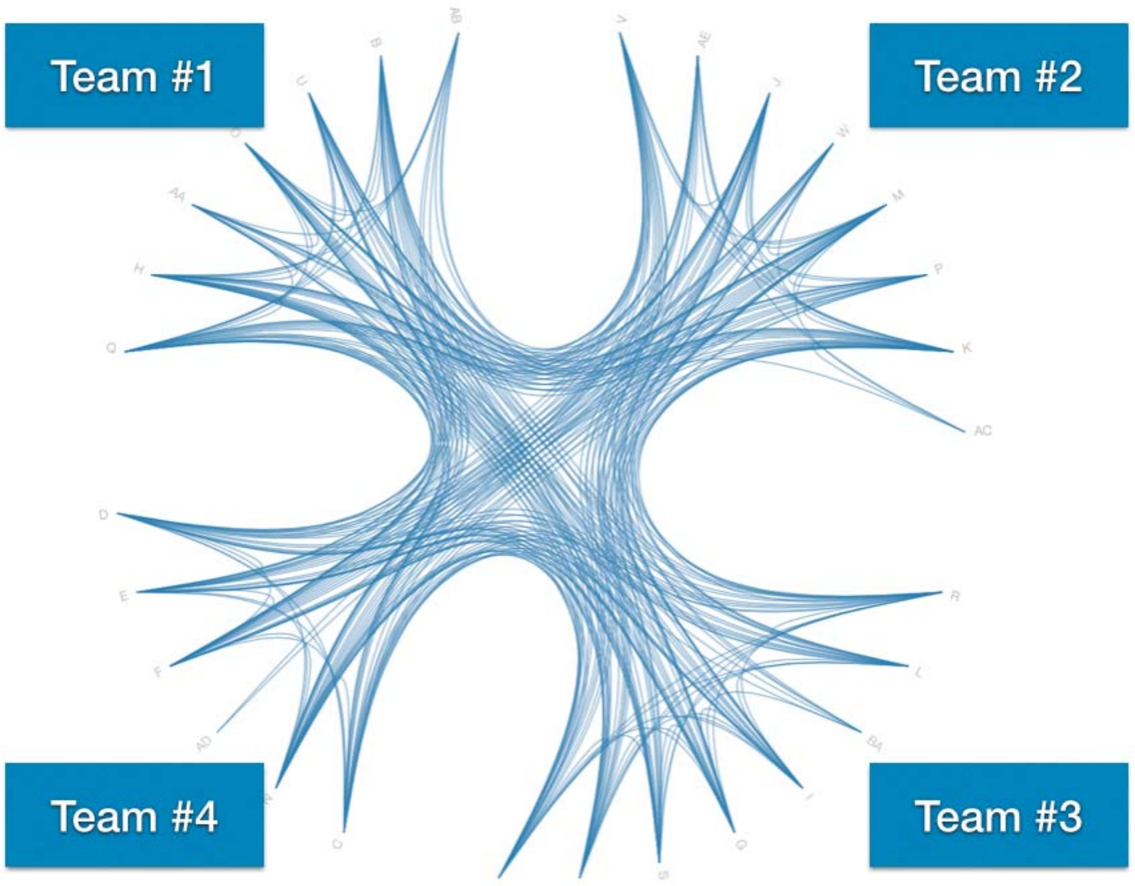
The trouble I met at that organization was a bit surprising since that company had started out in a good position. The company had set out to build a new product. And they were in a good position because they had done something very similar in the past. So they knew that the work would take approximately one year.

A software project that is predictable? I know that it looks crazy, but it almost happened. Then someone realized that there was this cool trade-show coming up in just three months. Of course, they wanted the product ready by then. Now, how do you take something you know takes a year and compress it down to just three months? Easy: just throw four times as many developers at it.

So they did.

The project was fast-paced. The initial architecture was already set. And in shorter time than it would take to read *The Mythical Man-Month*, 25 developers were recruited to the project. The company chose to organize the developers in four different teams.

What do you think the communication paths looked like on this project? Well, here they are:



It is a cool-looking figure, we have to agree on that. But let me assure you: there is nothing cool about it in practice. What you see is chaos. Complete chaos. The picture above doesn't really show four teams. What you see is that in practice there was one giant team of 29 developers with artificial organizational boundaries between them. This is a system where every developer works in every part of the codebase - communication paths cross and there is no shared context within any team. The scene was set for a disaster.

Learn from the Post-mortem Analysis

I didn't work on the project myself, but I got to analyze the source code repository and talk to some of the developers. Remember that I told you that we tend to miss organizational problems and blame technologies instead? That is what happened here too.

The developers reported that the code had severe quality problems. In addition, the code was hard to understand. Even if you wrote a piece of code yourself, two days from now it looked completely different since five other developers had worked on it in the meantime.

Finally, the project reported a lot of issues with merge conflicts. Every time a feature branch had to be merged, the developers spent days just trying to make sense of the resulting conflicts, bugs, and overwritten code. If you look at the communication diagram above you see the explanation. This project didn't have a merge problem - they had a problem that their architecture just couldn't support their way of working with it.

Of course, that trade show that had been the goal for the development project came and went by without any product to exhibit. Worse, the project wasn't even done within the originally realistic time frame of one year. The project took more than two years to complete and suffered a long trail of quality problems in the process.

Simplify Communication by Knowledge Maps

When you find signs of the same troubles as the company we just discussed, there are really just two things you can do:

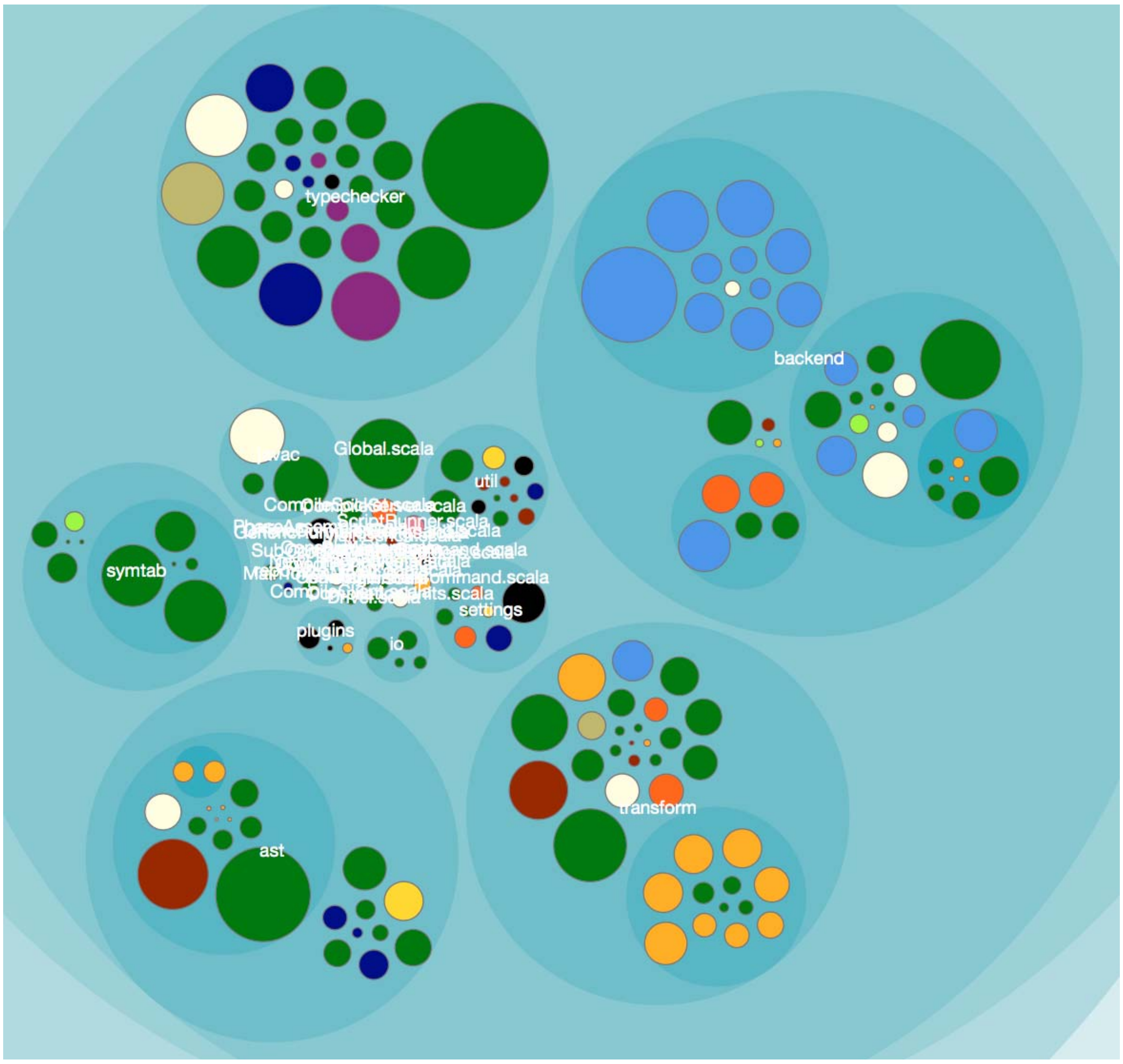
1. Change your architecture to support your way of working with it.
2. Adapt your organization to fit the way your architecture looks.

Before you go down either path you need to drill deeper though. You need to understand the challenges of the current system. To do that efficiently, you need a knowledge map.

Build a Knowledge Map of Your System

In Your Code as a Crime Scene [2], we develop several techniques that help us communicate more efficiently on software projects. My personal favorite is a technique I call Knowledge Maps.

A knowledge map shows the distribution of knowledge by developer in a given codebase. The information is, once again, mined from our source code repositories. Here is an example of how it looks:



In the figure above, each developer is assigned a color. We then measure the contributions of each developer. The one who has contributed most of the code to each module becomes its knowledge owner. You see, each colored circle in the figure above represents a design element (a module, class, or file).

You use a knowledge map as a guide. For example, the map above shows the knowledge distribution in the Scala compiler. Say you join that project and want to find out about the *Backend* component in the upper right corner. Your map immediately guides you to the light blue developer (the light blue developer owns most of the circles that represent modules in the backend part). And if she doesn't know, it is a fairly good guess the green developer knows.

Knowledge maps are based on heuristics that work surprisingly well in practice. Remember the story I told you where a simple change took a week since the affected code was shared between different teams? In that case there were probably at least 10 different developers involved. Knowledge maps solve the problem by pointing you to the right person to talk to. Remember - one of the hardest problems with communication is to know who to communicate with.

Testuff Test Management Tool - Click on ad to reach advertiser web site

“Simplicity is the Ultimate sophistication”
(Leonardo Da-Vinci)

testuff

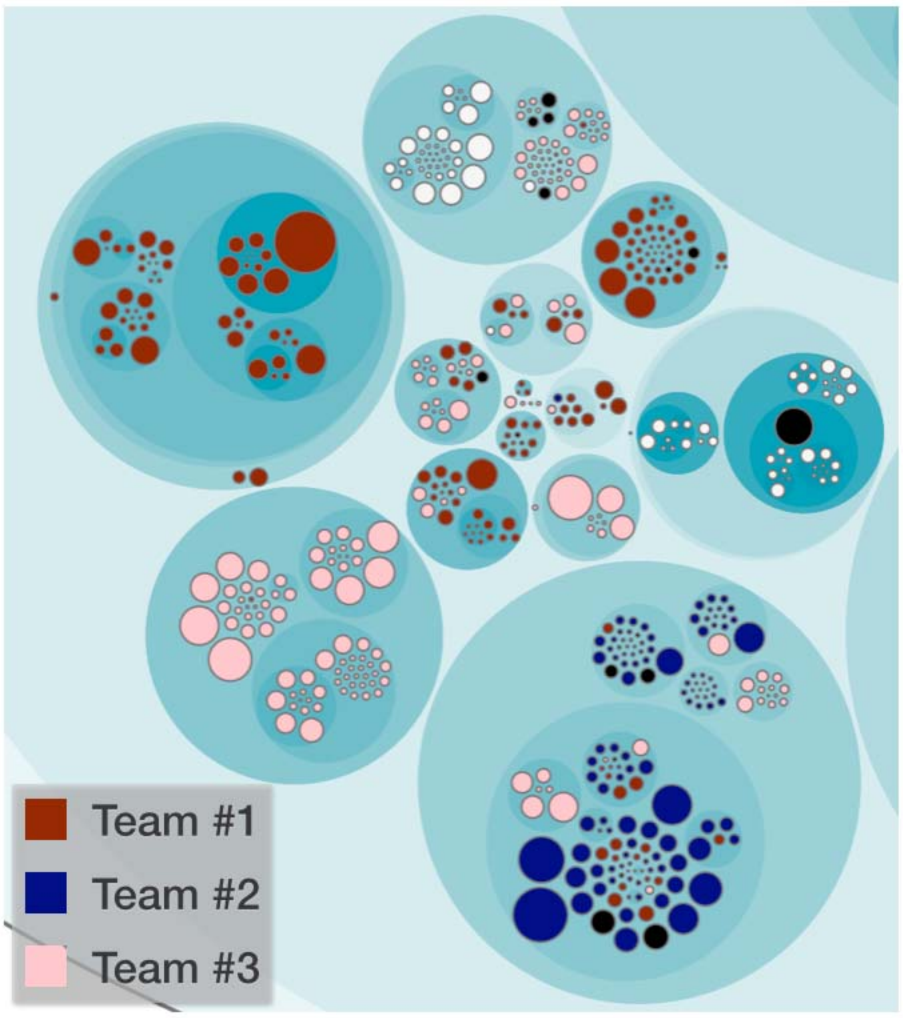
Test Management Tool

Simply #1

www.testuff.com

Scale the Knowledge Map to Teams

Now that we have a way of identifying the individual knowledge owners, let’s scale it to a team level. By aggregating individual contributions into teams, you’re able to view the knowledge distribution on an organizational level. As a bonus, we get the data we need to evaluate a system with respect to Conway’s Law. How cool is that?

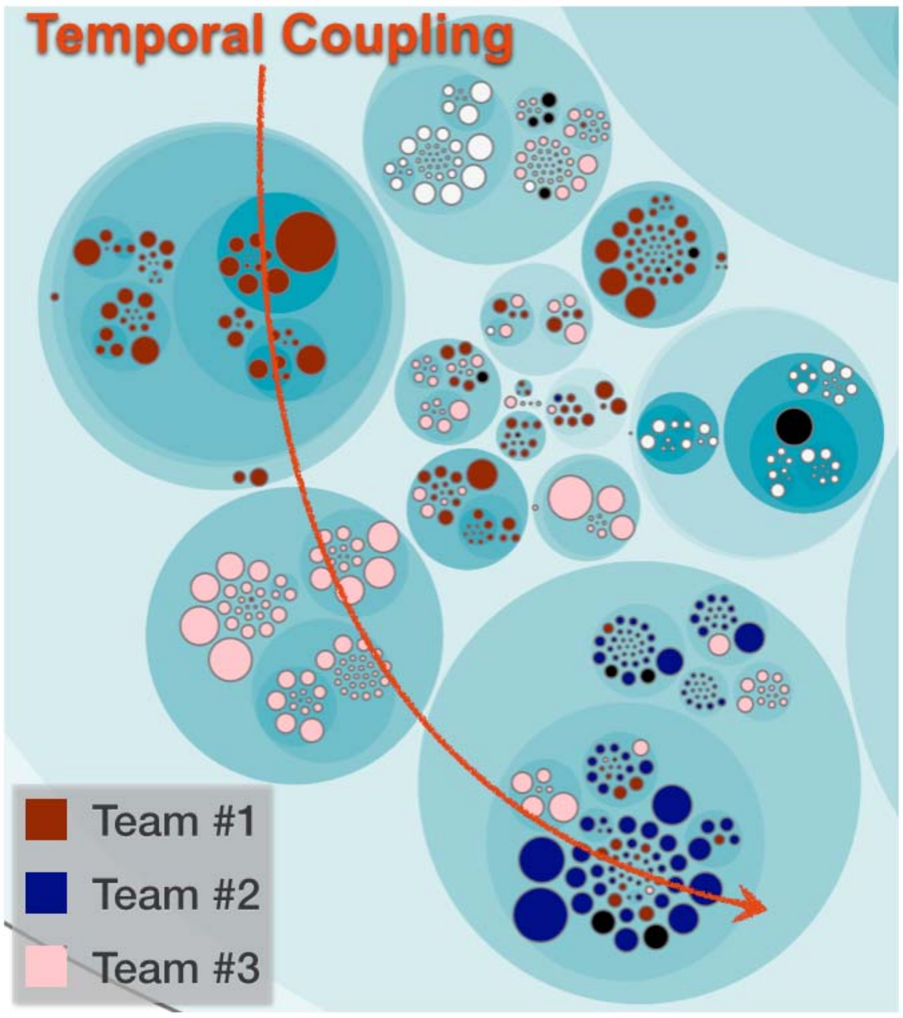


From the perspective of Conway, the map above looks pretty good. We have three different teams working on the codebase. As you see, the Red team has its own sub-system. The same goes for the Pink team (yes, I do as the mob boss Joe in *Reservoir Dogs* and just assign the colors). Both of these teams show an alignment with the architecture of the system.

But have a look at the component in the lower right corner. You see a fairly large sub-system with contributions from all three teams. When you find something like that you need to investigate the reasons. Perhaps your organization lacks a team to take on the responsibility of that sub-system? More likely, you’ll find that code changes for a reason: If three different teams have to work on the same part, well, that means the code probably has three different reasons to change. Separating it into three distinct components may just be the right thing to do as it allows you to decouple the teams.

Identify Expensive Change Patterns

Mapping out the knowledge distribution in your codebase is one of the most valuable analyses in the social arsenal. But there is more to it. What if you could use that information to highlight expensive change patterns? That is, change patterns that ripple through parts of the code owned by different teams. Here is how it looks:



The picture above highlights a modification trend that impacts all three teams. Where does the data come from? Well, again we turn to our digital oracle: version-control data.

There is an important reason why I recommend the history of your code rather than the code itself. The reason that dependencies between multiple teams go unnoticed is because those dependencies are not visible in the code itself. This will just become even more prevalent as our industry moves toward distributed and loosely coupled software architectures as evidenced by the current micro-services trend.

The measure I propose instead is *temporal coupling*. Temporal coupling identifies components that change at (approximately) the same time. Since we measure from actual modifications and not from the code itself, temporal coupling gives you a radically different view of the system. As such, the analysis is able to identify components that change together both with and without physical dependencies between them.

Overlaying the results of a temporal coupling analysis with the knowledge map lets you find team productivity bottlenecks. Remember the organization I told you about, the one where a small change took ages? Using this very analysis technique lets you identify cases like that and react on time.

Explore the Evolution

We are almost through this whirlwind tour of software evolutionary techniques now. Along the way, you've seen how to uncover the ideal communication paths in your organization, how to evaluate your architecture from a social perspective, and how you can visualize the knowledge distribution in your codebase.

We also met the concept of temporal coupling. Temporal coupling points to parts of your code that tend to change together. It is a powerful technique that lets you find true productivity bottlenecks.

As we look beyond the code we see our codebase in a new light. These software evolutionary techniques are here to stay: the information we get from our version-control systems is just too valuable to ignore.

I modestly recommend checking out my new book "Your Code as a Crime Scene" if you want to dive deeper into this fascinating field. The book covers the details of these techniques, lets you try them hands-on, and witness them in action on real-world codebases. You'll never look at your code in the same way again.

This article was originally published in April 2015 on
http://www.adamtornhill.com/articles/socialside/socialsideofcode.htm

References

1. http://www.melconway.com/Home/Conways_Law.html

2. Your Code as a Crime Scene, Adam Tornhill, The Pragmatic Programmers, ISBN 978-168050038, https://pragprog.com/book/atcrime/your-code-as-a-crime-scene

REST API Test Automation in Java with Open Source Tools

Vladimir Belorusets, PhD

Introduction

Nowadays, REST API testing has gained much deserved popularity due to the simplicity of verifying backend functionality. Availability of many open source tools and libraries supporting REST API development and testing also added incentives to automate these tests before the GUI development is complete.

REST API regression test automation includes generating code for HTTP calls and comparing the server’s actual response with the expected one. In my article "A Unified Framework for All Automation Needs - Part 2" [1], I described how to use the open source Spring Framework to generate REST calls and map JSON and XML responses to Java classes.

In this article, I will describe the automation process in detail. The intended audience is test automation engineers who need practical guidance in navigating through the large number of various tools available on the market. The article does not pretend to be a comprehensive overview of all the tools, but it describes a subset of the tools sufficient for successful and easy regression test automation.

The automation process consists of the following steps:

1. Compose a REST call manually with the help of a REST client tool.

In automating any test case, the first step is to perform that test case manually to see that the application produces the outcomes as expected. Only after that it makes sense to start writing the automation code. REST clients provide a fast and simple way to execute the REST calls manually and observe the responses in a convenient format.

2. Model the endpoint response into an object class using any object-oriented programming language of your choice.

That class will represent the JSON/XML tree structure and provide an easy access to the response values. In the next step, the object of that class will be used to store the response data in a file for future comparisons.

An alternative is to process the REST response as a string without mapping it to an object. Then, you need to apply a parser to that string to extract individual field values. This will make the response validation test much more complex and cumbersome.

3. Generate the same REST call programmatically and serialize the response object into a file as the expected result.

The serialized object represents the reference point. The regression test validates the condition that even after modifying the web service code, the REST call previously implemented continues to return the same data as in the stored object.

TimeShift Time Testing Solution - Click on ad to reach advertiser web site

Introducing TimeShiftX

Your **Time Travel Testing** Solution!

Validate

Leap Year Billing Pensions Rates
Policies Enrollment Financials

Learn more today!

www.vornexinc.com

vornex

4. Create a script that issues the REST API call and compares the response object with the expected object after deserialization.

The comparison between actual and expected results is done via xUnit assertions.

Let’s go through the set of tools that supports each step of the process outlined above.

REST Clients

There are plenty of the REST client tools for any taste. They allow generating REST calls without using a programming language and observe the response in JSON or XML format.

We can classify these tools in the following categories:

- Desktop REST client applications
- Web browser based REST clients
- Command line REST clients

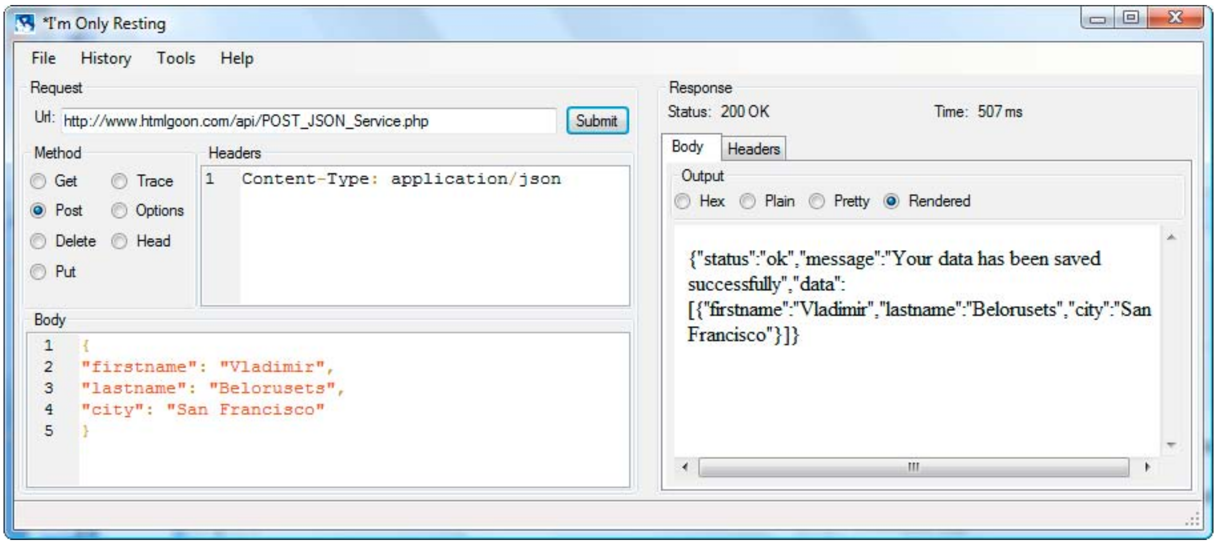
Most of the REST clients allow saving REST calls for future reuse either as files or favorite links. They also support common HTTP request methods such as GET, POST, PUT, HEAD, DELETE, OPTIONS, etc.

Desktop Rest Clients.

OS	**Tool Name**	**Web Site**
Windows	I’m Only Resting	http://www.swensensoftware.com/im-only-resting
Mac OS	CocoaRestClient	http://mmattozzi.github.io/cocoa-rest-client/
Any OS	WizTools.org RESTClient	https://github.com/wiztools/rest-client

Table 1. Desktop REST Clients.

The tools differ mostly in their GUI layout. As an example, ‘I’m Only Resting’ is presented here.



Url: http://www.htmlgoon.com/api/POST_JSON_Service.php

Method: [✓ Post]

Headers:

```
Content-Type: application/json
```

Body:

```
{
"firstname": "Vladimir",
"lastname": "Belorusets",
"city": "San Francisco"
}
```

Status: 200 OK   Time: 507 ms

Output: [✓ Rendered]

```
{"status":"ok","message":"Your data has been saved successfully","data":
[{"firstname":"Vladimir","lastname":"Belorusets","city":"San Francisco"}]}
```

Figure 1. ‘I’m Only Resting’ HTTP Client

There are also REST client plugins for popular IDEs that allow you to speed up code development by testing REST API in a separate panel without leaving your IDE.

IDE	Tool Name	Web Site
Eclipse	RESTClient Tool	https://marketplace.eclipse.org/content/rest-client
IntelliJ IDEA (Community Edition)	RESTClient	https://code.google.com/p/restclient-idea-plugin/

Table 2. IDEs REST Plugins

The IntelliJ IDEA plugin above imitates the desktop WizTools.org RESTClient.

Browser REST Clients.

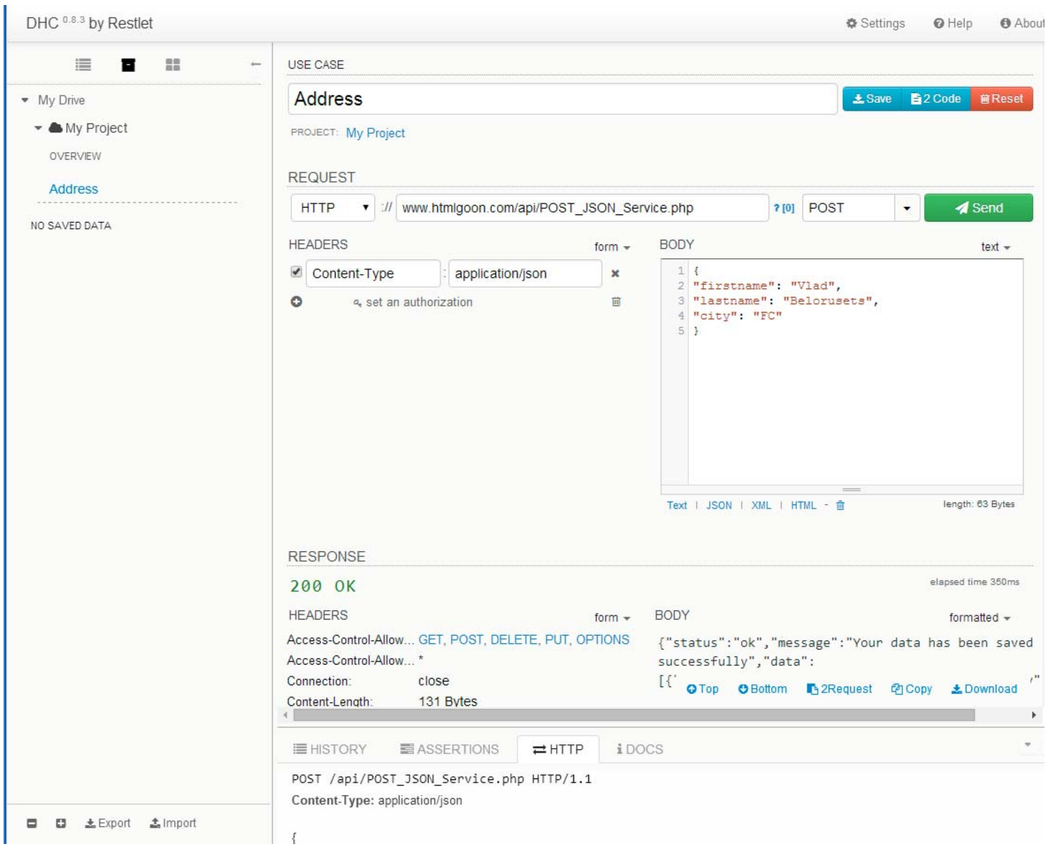
Popular browsers have numerous open source REST clients developed as their plug-ins. Some of these are listed below.

Browser	Tool Name	Web Site
Firefox	RESTClient	https://addons.mozilla.org/en-US/firefox/addon/restclient/
Firefox	REST Easy	https://addons.mozilla.org/en-us/firefox/addon/rest-easy/
Chrome	Advanced REST Client	http://chromerestclient.appspot.com/
Chrome	Postman	https://www.getpostman.com/
Chrome	DHC - REST/HTTP API Client	http://restlet.com/products/dhc/

Table 3. Browser REST Clients

Firefox clients are presented as buttons on the browser's toolbar. All Google Chrome clients are accessible via the Chrome App Launcher.

As an example, the Chrome DHC client is presented below.



Address

PROJECT: My Project

REQUEST

HTTP :// www.htmlgoon.com/api/POST_JSON_Service.php POST Send

HEADERS form

Content-Type : application/json

set an authorization

BODY text

```
{
"firstname": "Vlad",
"lastname": "Belorusets",
"city": "FC"
}
```

length: 63 Bytes

RESPONSE

200 OK

elapsed time 350ms

HEADERS form

Access-Control-Allow... GET, POST, DELETE, PUT, OPTIONS
Access-Control-Allow... *
Connection: close
Content-Length: 131 Bytes

BODY formatted

```
{"status":"ok","message":"Your data has been saved
successfully","data":
[{'
```

```
POST /api/POST_JSON_Service.php HTTP/1.1
Content-Type: application/json

{
```

Figure 2. Chrome DHC Client.

<u>Command Line REST Clients</u>.

Tool Name	**Web Site**
CURL	http://curl.haxx.se/
HTTPie	https://github.com/jkbrzt/httpie

Table 4. CLI REST Clients

The most widely used CLI client for REST calls is cURL which stands for ‘Client for URLs‘. It allows getting or sending files using URL syntax. In the example below cURL is used for HTTP POST in Windows.

```
C:\Users\Vlad>curl -H "Content-Type: application/json" -X POST -d "{\"firstname\
":\"Vlad\",\"lastname\":\"Belorusets\",\"city\":\"Foster City\"}" http://www.htm
lgoon.com/api/POST_JSON_Service.php
```

Figure 3. An Example of Using cURL for HTTP POST.

HTTPie client is a user-friendly cURL replacement featuring intuitive commands, JSON support, and syntax highlighting. The same REST call we used with cURL is presented with HTTPie syntax on Figure 4. By default, the Content-Type header set to application/json, and data fields in the message body are presented in JSON format. When a request contains data fields, HTTPie sets the request method to POST.

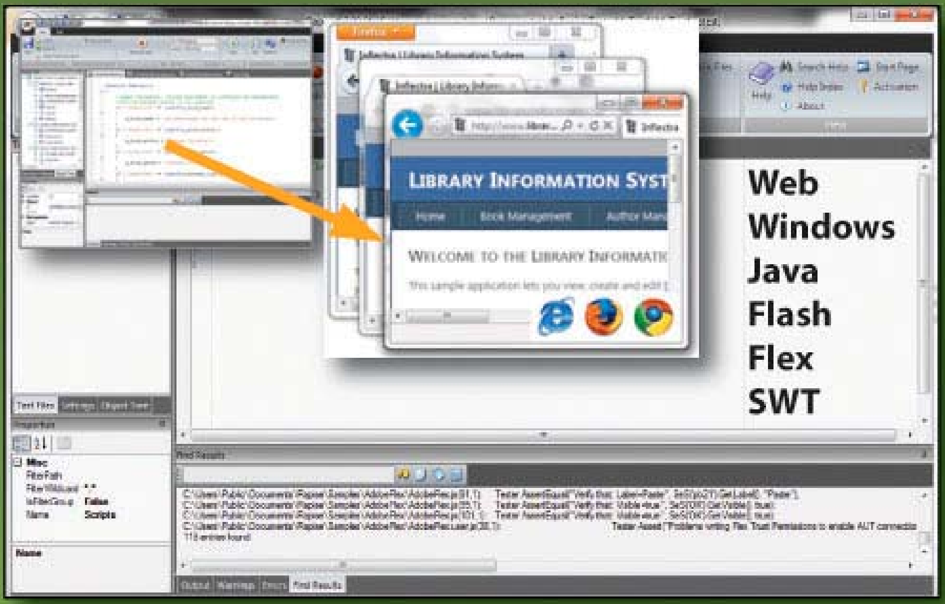
```
C:\Users\Vlad>http www.htmlgoon.com/api/POST_JSON_Service.php firstname="Vlad" l
astname="Belorusets" city="Foster City"
```

Figure 4. An Example of Using HTTPie for POST.

Rapise Rapid & Flexible Test Automation - Click on ad to reach advertiser web site

Need to Test Your Application on Multiple Environments?

Writing Test Scripts Too Slow?


Library Information System
Welcome to the Library Information
Web
Windows
Java
Flash
Flex
SWT

It's time to try a better way.

Rapise

RAPID & FLEXIBLE TEST AUTOMATION

Learn more at: inflectra.com/rapise

www.inflectra.com
sales@inflectra.com
+1 202-558-6885

inflectra®

Mapping JSON Response

Now it is time to develop the automated tests. Since Java is today the most popular programming language in the world [2], the implementation details are given in Java. The first thing to consider is how to validate the response data with assertions. There are many Java libraries that generate HTTP requests and get the responses as strings: java.net package in Java SE and Apache HttpComponents (http://hc.apache.org/), just to name a few. However, string comparison is not a solution since some data may contain dynamic values, such as ids and dates. Each time you produce the REST calls, these data in the response body will be different and your assertions will fail.

One option is to parse the JSON response string and extract the interesting individual name/value pairs. A popular open source library that provides this capability is JSON.simple (https://code.google.com/p/json-simple/). In this approach, the test case will consist of multiple assert methods. This is not a good design. It makes the test code longer and harder to perceive. If developers later add new fields, the test code must be updated, which should be avoided since the test logic stays the same and only the data have changed. The pattern to use is a single assertion on an expected object instead of one assertion per each object field [3]. It implies mapping the response to a Java class called POJO (Plain Old Java Object), with the class members corresponding to the JSON fields and using this object in assertion, *assertEquals(expectedResponseObject, actualObject),* given that assertion is customized to compare objects.

To process JSON response as a POJO, I recommend using the open source Spring Framework (http://projects.spring.io/spring-framework/). It comes with the built-in converters that map the web service responses into the relevant Java classes. *MappingJackson2HttpMessageConverter* class converts JSON into a POJOs using the Jackson library (http://wiki.fasterxml.com/JacksonDownload), and *GsonHttpMessageConverter* class does the same with the *gson* library (https://github.com/google/gson). Both converters are very similar. For the rest of this article, I imply using Jackson. It provides simple data binding by converting JSON to Java data types, such as maps, lists, strings, numbers, booleans and nulls.

After reviewing the JSON response by using one of the REST clients described in the previous section, you need to create a POJO object that Jackson can convert the data to. Jackson requires having getters and setters for each response field. The Spring Framework contains *org.springframework.web.client.RestTemplate* class that makes submitting REST calls and converting responses to Jackson POJOs very easy. For each HTTP request method, Spring offers corresponding *RestTemplate* method that includes Java class for expected JSON response as a parameter: *getForObject(), postForObject(),* etc. You can create a POJO manually, which can be time consuming for the complex JSON responses. Open source tools that simplify this procedure are presented below. They generate POJOs online after you paste the JSON string. An example of the jsonschema2pojo web page is presented on Figure 6.

Tool Name	**Web Site**
Jsonschema2pojo	http://www.jsonschema2pojo.org/
JSON to POJO	http://boldijarpaul.github.io/jsontopojo/
Convert XML or JSON to Java Pojo Classes	http://pojo.sodhanalibrary.com/
Json to Java	http://jsontojava.sinaapp.com/

Table 5. JSON to POJOs Online Converters

Mobile & IoT Dev + Test Conferences - Click on ad to reach advertiser web site

Be On the Lookout!

2 Conferences
1 Location

Back in 2016!

Mobile Dev + Test

A TECHWELL EVENT

The Mobile Dev + Test Conference is jam-packed with keynotes from top industry thought-leaders, full- and half-day tutorials, concurrent sessions on hot mobile topics, an Expo that is full of industry solution providers in the mobile space, and a full-day Mobile Innovation and Leadership Summit. Join us to find out where the future of smart and mobile software is headed.

- Mobile Development for iOS and Android
- Cross Platform Mobile Development
- Mobile App Testing
- Mobile Performance Testing
- User Experience (UX) Design
- Mobile App Security

NEW for 2016!

IoT Dev + Test

A TECHWELL EVENT

The challenges of developing and testing in the new digital frontier of Internet of Things are sometimes unique. The newly-added IoT Dev + Test conference focuses specifically on developing and testing for IoT and embedded systems. You will have the opportunity to learn from experts in the field where the future of IoT is headed and how to position yourself to be a part of this revolution.

- Smart Home and Office Apps
- Embedded Systems Development
- Wearable Technology
- Embedded System Security
- IoT and DevOps
- IoT Testig and Performance

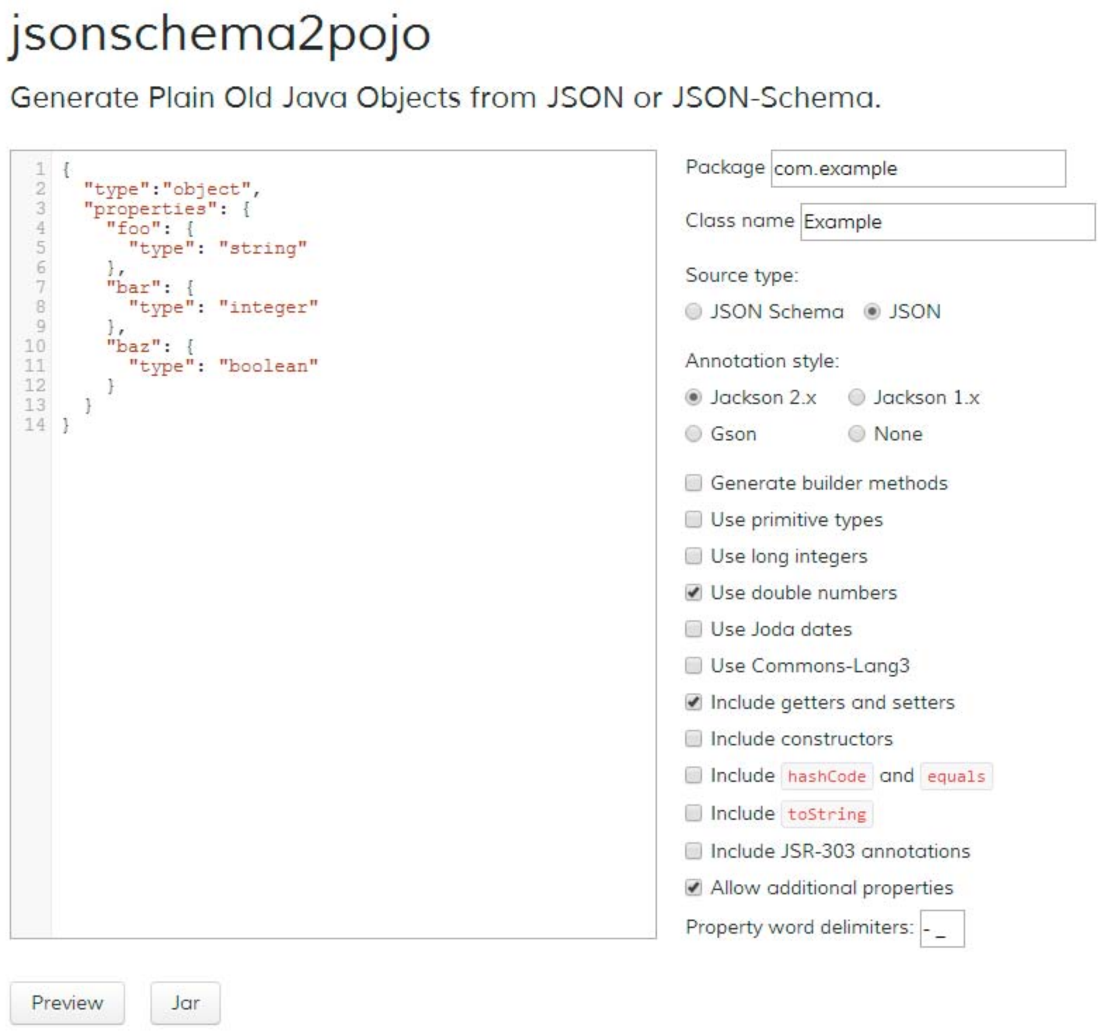
EXPLORE THE PROGRAMS

April 17–22, 2016 | San Diego, CA
Westin San Diego

SAVE UP TO AN ADDITIONAL $200 OFF with code MDTMT

Mobile-IoT-DevTest.TechWell.com

The Jackson project also provides classes that programmatically convert the JSON string into a POJO (https://github.com/FasterXML/jackson-docs/wiki/JacksonSamplePojoMapper).



```
{
  "type":"object",
  "properties": {
    "foo": {
      "type": "string"
    },
    "bar": {
      "type": "integer"
    },
    "baz": {
      "type": "boolean"
    }
  }
}
```

Package: com.example
Class name: Example
Source type: JSON Schema [✓ JSON]
Annotation style: [✓ Jackson 2.x] Jackson 1.x Gson None
- [ ] Generate builder methods
- [ ] Use primitive types
- [ ] Use long integers
- [x] Use double numbers
- [ ] Use Joda dates
- [ ] Use Commons-Lang3
- [x] Include getters and setters
- [ ] Include constructors
- [ ] Include `hashCode` and `equals`
- [ ] Include `toString`
- [ ] Include JSR-303 annotations
- [x] Allow additional properties

Property word delimiters: - _

Figure 6. jsonschema2pojo Online POJO Generator.

Serializing Response Objects

The next step is to perform an end-point call programmatically and serialize the JSON response as the POJOs in files for future references and comparison. However, as I mentioned before, some of the response fields contain data varying from call to call.

To identify those fields, you can issue the same call a few times using one the REST clients. Then, you can compare JSON responses for differences with the help of the online tools listed below.

Tool Name	Web Site
JsonDiffPatch	http://benjamine.github.io/jsondiffpatch/demo/index.html
JSON Diff	http://tlrobinson.net/projects/javascript-fun/jsondiff/
JSON Diff View (Firefox)	https://addons.mozilla.org/en-us/firefox/addon/json-diff-view/
JSON Diff	http://jsondiff.com/

Table 6. Tools for Finding JSON Differences

There is no reason to store those data because the tests will always fail in assertions. A solution is to use the *transient* keyword for the corresponding POJO variables. The values of the transient fields will not be serialized. But we cannot ignore those fields completely since some of them are mandatory, and an important part of the tests is to guarantee that they are not nulls. Field validation can be performed on the backend as well as on the frontend.

To address that issue on the client side, we can use the open source Hibernate Validator library (http://hibernate.org/validator/). It provides annotation-based constraints on class variables and transparent integration with the Spring Framework. The annotations of interest are: *@NotNull*, *@Size(min=.., max=...)*, *@Min(...)*, and *@Max(...)*. You need to decorate mandatory POJO class variables with these annotations and test the response object using *javax.validation.Validator*.

Response Verification

The last step in the automated test includes comparison of the deserialized expected object from the file with the actual JSON response mapped to the POJOs. In JUnit, verification is accomplished by using assertions, such as *assertEquals()*. However, if the fields in the response class contain other objects, out-of-the-box assertions fail even when the objects being compared contain the same values. The reason is that comparison is conducted by using '==' operator applied to the reference variables without recursively checking all object fields. I recommend two libraries to compare object hierarchy field by field in one statement.

Library Name	**Web Site**
Unitils	http://unitils.org/
Shazamcrest	https://github.com/shazam/shazamcrest

Table 7. Recursive Assertion Libraries

Unitils *assertReflectionEquals(expectedObject, actualObject)* loops over all fields in both objects and compares their values using reflection. If a field value itself is an object, it will be recursively compared field by field. Reflection assert automatically excludes *transient* variables from comparison. You can also ignore non-transient values by specifying *ReflectionComparatorMode.IGNORE_DEFAULTS* mode in the *assertReflectionEquals()*. This mode excludes all fields that designated as *nulls* in the expected object.

Shazamcrest is a library that extends the functionality of Hamcrest assertions with a special matcher *sameBeanAs()*: *assertThat(actualObject, sameBeanAs(expectedObject).* The comparison is done field by field including fields in object variables. It also provides capability of ignoring fields from matching by specifying which fields to ignore as follows: *sameBeanAs(expectedObject).ignore("field1").ignore("field2").*

Code Samples

Let's assume as a result of GET request to http://www.myREST.com/person, JSON response will be mapped to the class Person using the Jackson converter. The explanation comments are provided within the code.

```
 public class Person
{
// using Hibernate Validator annotation
    @NotNull
    private transient String id;
    private String name;
    private Address address;

    public Person(String id, String name, Address address)
    {
        this.id = id;
        this.name = name;
        this.address = address;
    }
// getters and setters must be added
}

public class Address
{
    private String street;
    private String city;
    private int zip;

    public Address(String street, String city, int zip)
    {
        this.street = street;
        this.city = city;
        this.zip = zip;
    }
// getters and setters must be added
}

@Test
public void testREST()
{
// use Spring Framework for generating GET request
  RestTemplate restTemplate = new RestTemplate();
  Person actualPerson = restTemplate.getForObject(
       "http://www.myREST.com/person", Person.class);

// let's assume for illustration that we received the
// following response object
  actualPerson = new Person(null, "John Doe",
          new Address("1 Main St", "San Mateo", 94404));
// then validation using Hibernate Validator fails
  assertTrue("Some required fields are nulls",
      RestUtils.validateObject(actualPerson));
/*
   the following error message will be displayed:
        java.lang.AssertionError: Some required fields are nulls <2 internal calls>
            at TestPerson.testREST(TestPerson.java:15) <23 internal calls>
        id may not be null
*/
// create your methods serialize to and deserialize from the file to extract the
// expected POJO
  Person expectedPerson = Utilities.deserialize("person.ser");
// let's assume that after de-serialization we get the following object
  expectedPerson = new Person("1234", "John Doe", new Address("1 Main St",
"San Mateo", 94404));

// both assertions pass since they ignore transient id
```

```
  assertReflectionEquals(expectedPerson, actualPerson);
  assertThat(actualPerson, sameBeanAs(expectedPerson));

// if the expected object is as follows
   expectedPerson = new Person("1234", "John Doe", new Address("1 Main St",
"San Mateo", 94107));

   assertReflectionEquals(expectedPerson, actualPerson);
/*
    the following error message will be displayed applying Unitils assertion
        junit.framework.AssertionFailedError:
        Expected: Person<, name="John Doe", address=Address<street="1 Main St",
city="San Mateo", zip=94107>>
        Actual: Person<, name="John Doe", address=Address<street="1 Main St",
city="San Mateo", zip=94404>>

        --- Found following differences ---
        address.zip: expected: 94107, actual: 94404
*/
   assertThat(actualPerson, sameBeanAs(expectedPerson));
/*
    the following message will be displayed applying Shazamcrest
        java.lang.AssertionError:
        Expected: {
            "name": "John Doe",
                    "address": {
                        "street": "1 Main St",
                        "city": "San Mateo",
                        "zip": 94107
            }
        }
        but: address.zip
        Expected: 94107
        got: 94404
       <Click to see difference>
*/
}

public class RestUtils
{
    public static <Type> boolean validateObject(Type object)
    {
        Validator validator =  ...........................................

       Validation.buildDefaultValidatorFactory().getValidator();
        Set<ConstraintViolation<Type>> violations = validator.validate(object);

        if (violations.size() > 0)
        {
            for (ConstraintViolation cv : violations)
            {
                System.out.println(cv.getPropertyPath() + " " + cv.getMessage());
            }
            return false;
        }
        return true;
    }
}
```

Summary

The article describes a set of open source tools that give testers of all levels ability to conduct REST API verification. Manual testers can use REST clients that provide convenient GUI for hitting the end-points while test automation engineers with Java experience can develop comprehensive tests fast. Although the choice of tools is subjective, I hope you enjoyed using them as I did.

References

1. Belorusets, Vladimir. A Unified Framework for All Automation Needs - Part 2. Testing Experience, Issue No 27, pp. 9-13, 2014.

2. TIOBE Index for October 2015 (http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html)

PYPL PopularitY of Programming Language (http://pypl.github.io/PYPL.html)

3. Meszaros, Gerard. *xUnit Test Patterns*. Addison-Wesley. 2007.

Database Continuous Integration

Alex Podlesny, http://dbtestdriven.com/

Continuous Integration (CI) and Continuous Deployment (CD) are two common trends in software development. How can you integrate the database changes in this process? This article explores the different aspects of continuous integration from a database perspective.

1.The Software Development Life Cycle (SDLC)

Power of the SDLC

In small startups and big corporations, the software development process follows internal lifecycle adopted by the company. It can be a one- or two-step process, it can be manual, it might be evolving. The benefits of the SDLC are hard to underestimate since they provide standard guidelines, minimize risk, improve quality, provide consistency and play a significant role in safeguarding actual business and its products.

Flow of code

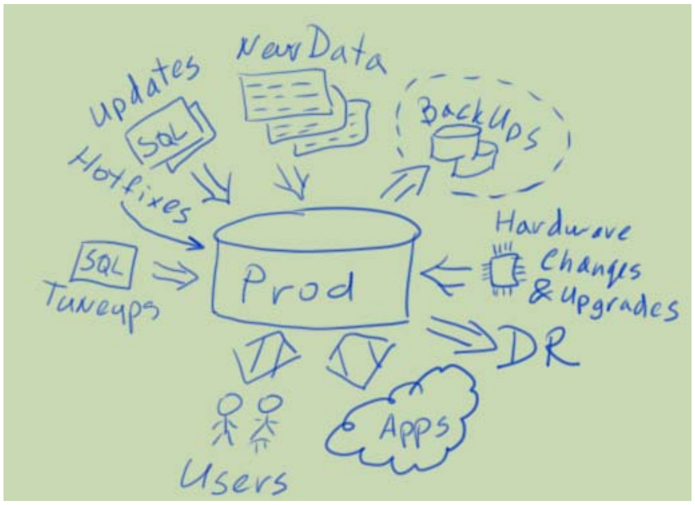
Everything starts in the development camp. It is where changes are made, queries are written, tables created, and databases designed. All changes, no matter how big or small, eventually flow to a testing, staging, or other non-production environments. Over there, those changes are reviewed by someone other than the original creator, tested, checked for compliance, verified, documented, accepted and scheduled for a production release. Scheduled releases are double-checked, triple-checked and eventually deployed to a production environment. In production, the newly introduced changes are confirmed, checked, rolled back when necessary, documented and left to be forgotten.

As database projects mature, development priorities are slowly overtaken by the invisible support and operation priorities; changes to the database design might still happen, but they are happening on a very limited scale. Many new changes tend to follow the SDLC process, but that does not happen all the time.

The life of a production database

Life turns into a real adventure in a production environment. Here, a new code is released and updates are installed. Hardware is changed. Performance is tuned. Backups are configured. Recovery is exercised to verify a backup process. Availability is insured. New data is loaded. Hardware-software-network issues are resolved; configuration is changed; security and permissions are updated. After a database becomes live, the flow of requests never stops.

For many organizations, the changes to production databases might follow different SDLC (formal or informal) and never trickle back to the development environment. After a while, the number of differences grows between the development and production environments.



Development environments

Oh yes, those development environments…

The Development environment is where changes are happening. Regardless of the importance this environment has very little love. It is not the favorite environment for developers, neither it is the preferred environment for testers nor it is an ideal environment for administrators. So why do we have this absence of love? I suspect that it is for the following reasons:

- This environment is refreshed with production data only once in a while;
- It is most likely underpowered and useless for query tuning;
- Developers can delete each other’s code and data at any time, well, mostly unintentionally;
- It is unreliable - too many changes in data and logic, too many contentions;
- It has always bugs and unfinished features;
- Testing activities leave too much rubbish behind.

Competition for database resources is one of the primary causes that hold development teams back. Competing needs affect the quality, stretch timeline, pose multiple limitations and consume a mountain of team’s time to sort out all unintentional impact.

The Testing environment gets a little more love. It is updated more often. It is where developers would like to go to make sure that testers do not find their bugs. Continuous impact from testing efforts is anticipated, but this is mitigated by frequent refreshes.

The Pre-Production or User Acceptance Environment (because that is where users can try all new changes) is also an environment where developers and testers want to be. Pre-production environment is a favorite of database administrators (DBAs) because it is as close to production as it can get. From a hardware perspective, it is a sandbox for performance tuning and a final resort before delivering new changes to a live production database.

2. Define a Product

Let's define what is that we are trying to integrate continuously. At the first glance, it might seem obvious that we are talking about just the one thing - the database, and that is our product of integration.

Do we really talk about one thing?

When we say - "database" - here is what people might think:

- The DBA would think of physical servers with database software instances (like Oracle or SQL Server) installed on a few servers, a database created with files located on physical and logical drives, and populated with data;
- The developer who works with database centered application might think of a few tables that exist somewhere in a database this application have access to;
- Database designer might think of relationship, integrity, access control, particular database objects that achieve the required functionality;
- User would think simply of data. Data loaded from raw files, collected from sensors, manually entered, bought or acquired in any other way. User needs data.

The magical "database" term removes obstacles and allows that all stakeholders in their different roles work together. At the same time, it carries very high level of ambiguity. Let's try to remove ambiguity, set expectations and to focus on actual CI implementation for a particular database product.

Please welcome the “DNA and The Creature” example:

DNA		**The Creature**
On one side there are developers and designers who create databases, write database code, create packages and procedures	The Terminology Wall	On the other side, there are Users who use data in a database and DBAs who support that database

Let say, for example, we have designed a database that describe a creature, the "The Creature" database. Two tables is all we need:

```
CREATE DATABASE TheCreature
 GO
CREATE TABLE CreatureAttributes(
      AttributeID int,
      Name nvarchar(50),
      Value nvarchar(50)
      )
GO
CREATE TABLE SignificantEvents(
      EventID int,
      EventDate datetime,
      EventDescription nchar(10)
      )
GO
```

Done - we got a product. So… Coding is done; testing is done, we got approval from the client; we saved our code in source control system as one small file called **A_Creature_DNA.sql**

Then, this product (the SQL file) was distributed to many divisions of the company. Many departments installed this product internally on their production servers and started using their own “The Creature” databases.

Let say one office accumulated following data:

CreatureAttributes table

AttributeID	Name	Value
1	Class	Mammal
2	Family	Rabbit
3	Name	Bunny
4	Diet	Herbivore
5	Lifespan Limit	12 years
6	Birthday	2012-01-01

SignificantEvents table

EventID	EventDate	EventDescription
1	2014-01-01 4:00 PM	New creature is born of class Mammal
2	2014-01-01 4:01 PM	New creature chooses to be a Rabbit
3	2014-01-03 1:00 AM	Creature tried to eat grass - and it likes it
4	2014-01-03 2:00 PM	We named it Bunny
5	2015-02-03 7:00 PM	Bunny ran away from Fox. Very scary...
6	2016-04-04 8:00 AM	Bunny met Dolly, they are friends now
7	2019-05-08 9:00 PM	Bunny got to a hospital
8	2019-05-09 5:00 AM	Bunny feels better and going home
9	2020-08-23 9:00 AM	Bunny goes to school

There also might be other department, who grew their rabbits or lions, or horses, or other unknown chimeras. From a developer Joe standpoint, who invented “The Creature” database, the final database product is the SQL code for those two tables. From user Jill standpoint, who uses this product daily, the “The Creature” database is in the actual database installed on the server and populated with real data. Both Joe and Jill have different needs from the database. Both use a “database” term in their way. In the end, they are working with two different database products - **the database code and the database that this code has created**.

3. What is Continuous Integration?

Continuous Integration - What is it? How can we get there? Can we buy it off the shelf?

Continuous integration (CI) is the practice of making sure that changes are integrated continuously into your product so that many issues, integration problems, and impact can be discovered and avoided earlier in the SDLC process.

The main principles of continuous integration process are:

- Maintain source code repository;
- Commit changes often, never longer than a day;
- Every commits to the master branch should be built;
- Employ build automation;

- Make sure that artifacts are available as installation packages;
- Automated unit testing;
- Build process should be fast;
- Keep the set of pre-production environments. Configure automated deployments to them;
- Make build results available;
- Tune the process to the point of continuous deployment to production.

If you choose a single tool, you may find over time that some of the steps are hard to achieve, or that they become an impediment to your business, slowing it down, impacting cost. If it is where you are: look for other tools or use multiple tools or different solutions. Be open. To get continuous integration right, you need to persist and be a leader! You will have a long path to team building and nurturing a CI culture. You will spend much time selling the idea to your peers, superiors, and the team. Know that not every team member adopts a plan. Take one step at time.

Getting CI working for your company can be a fun quest; it should not be bloody conquest. Don't get upset, give everybody his or her time, persist, and become an example of excellence. CI sound cool, but it might not be good for your project. You have to decide if you want to use it or not.

4. Looking through development lenses

From an application standpoint, the database is essentially a resource. As long as the application has access to a given database it will work. Looking back to the second section of our discussion, essentially this resource is some “Creature” created base "Creature DNA".

So, if we can create brand-new-fresh-and-clean database, backup this database and keep files as artifacts, then these files can be used by other processes to restore and reuse that brand-new-fresh-and-clean database in the target environment.

If this were achievable, then data-centric application would have the luxury to use clean non-polluted database with the latest tested code.

Here is how to implement CI practice as a backbone of the development process:

1. Maintain source code repository:

- As the first step, installing in-house (or using a hosted solution) SVN, Git, Mercurial, TFS or any other system would do.
- At the next step, all database code should be scripted. Make scripts that create tables, views, triggers, stored procedures, functions, synonyms, schemas, roles, permissions, data types, constraints, and every other thing that an application might need. Together with scripts that create database objects, you need scripts to populate application-specific metadata and a few seed records.
- Each of those scripts should be checked-in into code repository.
- The goal here is to make the clean database populated with a default set of metadata;

2. Commit changes often, never longer than a day:

- All developers should commit code changes to source code repository as soon as possible, providing comments and explanations why changes made, why new objects added or why they are removed.
- Changes to default metadata should be checked in and documented as well.
- There is no big task that could not be split into committable small sub-parts, developers should not hold features and code in hostage;

3. Every commit to the master branch should be built:

- Out of many branches that developers use in the source code repository, a master branch should always be build when new changes are committed;
- But what does it mean to build? How can it be built?
- These are good questions and we will answer them later. For now let assume that we prepare a package that have all the scripts necessary to create brand-new-fresh-and-clean database;
- To achieve this, we create a set of build scripts that automatically download all committed changes from source code repository, run those database scripts to produce brand-new-fresh-and-clean database, and finally backup this database;

4. Build automation:

- Many build automation tools exist on the market today, like TeamCity, Jenkins, Hudson, and others. Some of them are free. Those tools make the life of automating build processes very easy and enjoyable.

5. Make sure that built artifacts are available as installation packages

- Save time for your fellow team members, instead of writing miles of deployment documentation, create installation packages ready for automated deployments and available to team members.
- What sort of installation package to create? Are there many different kinds?
- These are superb questions, and we cover them in more details below. For now let assume that we prepare an archive with following files:
- a package that have all the scripts necessary to create a brand-new-fresh-and-clean database script that call this package to create brand-new-fresh-and-clean database on the target server Installation package should be easily usable by existing commercial, open source or simple home grown tools

6. The build process should be fast:

- The build server should continuously monitor source code repository for new changes and start the build process as soon as possible.
- Building a product in a few minutes is excellent, in half an hour is questionable, in an hour is unacceptable.
- System might be configured to check for changes, compile the solution, run all unit tests, prepare and publish artifacts - all in One Giant Process.
- Alternatively, it may be configured with many small independent sub-processes that create their artifacts.

7. Automated unit testing:

- After the build is done, product should be checked for conformity to expectations and requirements by running a set of automated unit tests.
- Unit tests should be created by developers, since they are those who build a feature, and they are first who verifying that new functionality. However, that might not always be the case; integrators and testers should automate their testing as well.

- Adopt Test-Driven Development (TDD) practices.
- Unit tests should be stored in the source code repository and be available to the build servers alongside the database scripts used to create database artifacts.

8. Available build results:

- Everyone in a team should know that the build failed. The earlier the team knows, the faster they can fix it.
- Preventing downstream environment from getting corrupted code, and uncovering integration issues is at the core of a continuous integration practice
- Any CI and deployment servers existing on the market would make build results available to users. They can be configured to share the status in many different ways: from email to text message, from the web page to flashing red light in the hallway.
- Everyone should know when a build failed, what code brought it down, when was it fixed and how;

Some might argue that next two steps are not relevant when the product is just a brand-new-fresh-and-clean database intended to be used by developers.

For now here they are the extra steps:

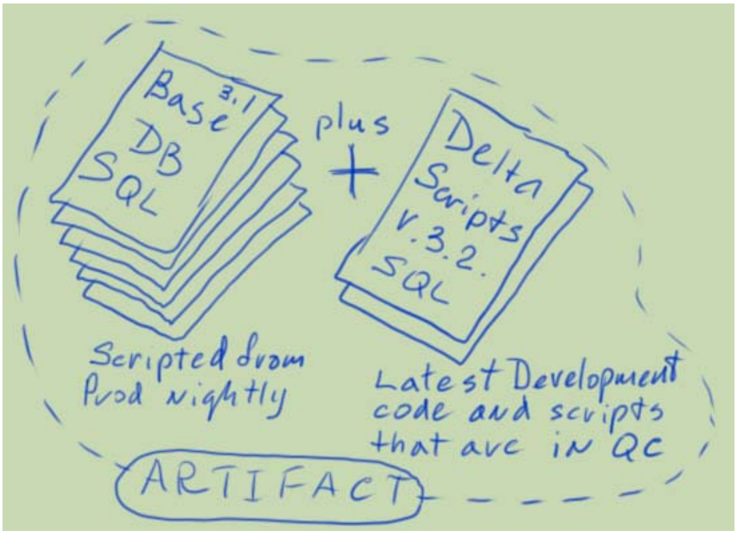
- Keep the set of pre-production environments. Configure automated deployments to them
- Tune the process to the point of continuous deployment to production

5. Artifacts and Installation Packages

Almost at every step of a CI process there would be some sort of a database artifacts generated; they can be different and similar, and all fulfill their individual purpose. Here a few core samples:

SQL Scripts - a group of database SQL scripts. They can be separated in following categories:

1. The complete list of scripted-out database objects and metadata. If run in an empty database server, it creates brand new database will all of its objects and initial set of data;
2. A package that add new functionality or changes existing functionality to an existing database;
3. A package that inserts new or manipulates existing data within existing database.


Base 3.1 DB SQL
plus +
Delta Scripts v.3.2. SQL
Scripted from Prod nightly
Latest Development code and scripts that are in QC
ARTIFACT

An installation, in this case, can be a simple execution of the scripts on target database server. While script execution might not be a viable approach for databases with many objects and metadata, it can be quite efficient to deliver hotfixes and incremental updates, especially to production environments.

Backups or DB Files - different flavors of backup processes are available for various database engines. Some of them work for a particular use case better than other. They usually divide into following categories:

- Cold backup - is done when database server is down, and actual database files can be copied to another location (preferred approach). In Oracle, it might require server shutdown while in the SQL Server it can be achieved by detaching the database in order to gain access to actual database files;
- Hot backup - performed while database is in use.

An installation, in case of backups, can be completed by the set of scripts that perform automated restore on a target database server. While installation would be an excellent solution for application development and automation of unit tests, it has limited use for deploying changes to running production and running test environments.

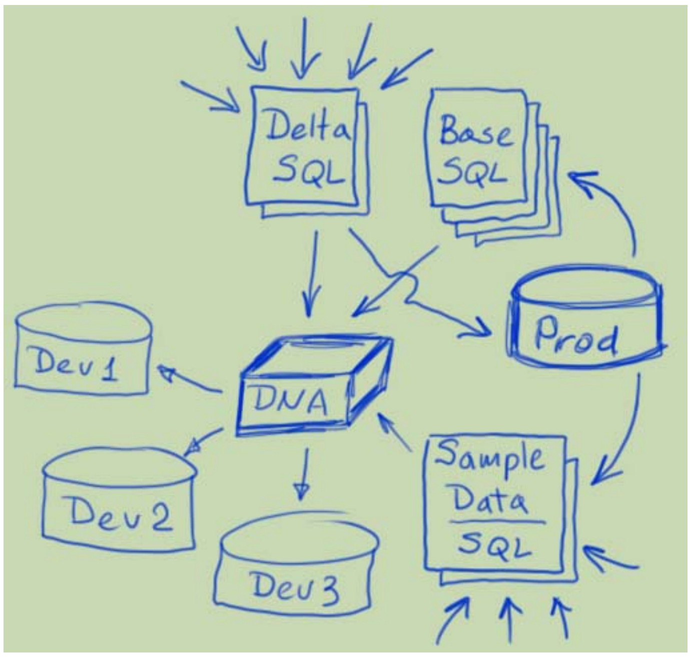
Virtual Servers - a set of fresh virtual clones can be used as a starting point for a CI process to install fresh and clean databases. An installation, in this case, consists of an on-demand virtual machine provisioning by a consuming system. Solution and a type of artifact do not need to be rigid. A mix of the strategies can be chosen to satisfy development and integration tasks that appear at a time.

6. From Dev to Prod and Back

Back in the first section, where we have discussed the SDLC, we have introduced the realities of the production world. Many businesses are built around one database or, perhaps, multiple databases. They grow together, expand and evolve through well know development cycles:

1. Develop new feature
2. Move it to production
3. Use a new feature until it becomes old
4. Come up with new ideas
5. Back to Step 1

The big difference between CI for a product development and CI needed to support corporate production environment is a product of integration: "The Creature DNA" vs. "The Creature" (check section 2 for more information). While "Creature DNA" can be sitting on a shelf and can be assembled and reassembled any time. The actual "The Creature" database could not be turned off, reassembled or changed, especially in the high-availability projects. Ironically over lifetime, very close to real life scenario, the Creature can get new properties that original DNA did not even had. Maintenance, performance tuning, constant flow of new data and hotfixes, hardware-software-network issue resolution, upgrades, configuration changes can quickly introduce mountains of differences in how both behave.



If technologists want to deliver new functionality to running, potentially high-availability database, they need to look into the idea of incremental upgrades. Upgrades that slowly transform an existing system into an adaptive and always evolving production environment with new functionality and without impact. When the idea of continuous updates and continuous delivery of changes is accepted, the next step is to set up a backflow of changes, unknown to developers, from the production environment back to the development.

In such a transformed place:

- the product of CI for a production environment can be a simple set of scripts that introduce new functionality - an upgrade
- the goal of CI stay the same - is to make sure that by the time those scripts are ready to be deployed to production, there are:
  - no known integration issues
  - a minimal as possible risk of impact

Production lifecycle should include steps to push all production changes continually back to development environments and close the potential code difference gap between development and productions environments:

Dev --> Prod	**Prod --> Dev**
1. develop new feature 2. move it to production 3. use new feature until it becomes old 4. come up with new ideas 5. back to Step 1	1. recognize a change in production 2. script that change 3. deliver this change back to source code repository in development environment 4. back to Step 1

7. Production CI

Here is how CI practice can be implemented as the backbone of the production process:

1. Maintain a source code repository

- Follow the same principles outlined in section 4, plus:
- Create separate Version Control System (VCS) folder (or branch) to store all objects scripted from a production system. This folder becomes a "Base Source Code" for your database.
- Create separate VCS folder (or branch) for each release that is currently in the queue to go to production, all development changes represent a "Delta Source Code". The Delta is where all developers are contributing their code until feature is ready to go to the next step in the adopted SDLC process.

2. Commit changes often, never longer than a day

- Follow the same principles outlined in section 4, plus:
- Set up automated process that script all production database objects and core metadata in a production database, and then checks all those scripts to the VCS repository on a daily basis.
- Yes, every night an automated process should collect and check-in production code into VCS. Over time, it becomes a history of all production changes that can be used to monitor the state of the production system.

3. Every commit to the master branch should be built

- Follow the same steps outlined in section 4, plus:
- All changes from the production environment should be incorporated into the build to help control assumptions.
- In addition to a brand-new-fresh-and-clean database artifact that needed for development, a production oriented CI process should produce a Delta Release artifact, which later is used to update production system automatically.

4. Build automation

- Build automation should become the backbone of the production delivery process with multi-layer, multi-environment delivery system.

5. Make sure that artifacts are available as installation packages

- Follow the same principles outlined in section 4
- Installation package should be easily usable by existing and homegrown solutions. Keep your options open, tools are changing very rapidly, make sure you have a way to adopt a new approach

6. Build process should be fast:

- Follow the same principles outlined in section 4, plus:
- Production CI process can quickly become multi dependent and layered, look for ways to improve delivery time for each part of the process and for the process as whole;
- Manage the risk of "never building the final product", minimize the impact from future releases that currently are actively developed. Isolate, separate, branch, minimize and simplify each release. Try to achieve a "one deployment a day" goal, when you are there improve too many per day, if you are not there re-think re-engineer, change…

7. Automated unit testing

- Follow the same principles outlined in section 4, plus:
- Unit tests should become paramount requirements. Any changes in code, no matter how small should have a unit test for that change.
- Test-Driven Development anybody? Why not?
- Focus on code coverage; improve it with every commit to VCS.
- Introduce code review and sign-in for every pull request. Please, somebody, check my code!
- Introduce integration tests. Developers cannot mock forever. Test the actual code you build.
- Create and automated process for any steps that run multiple times, those steps should not be executed manually.

8. Available build results

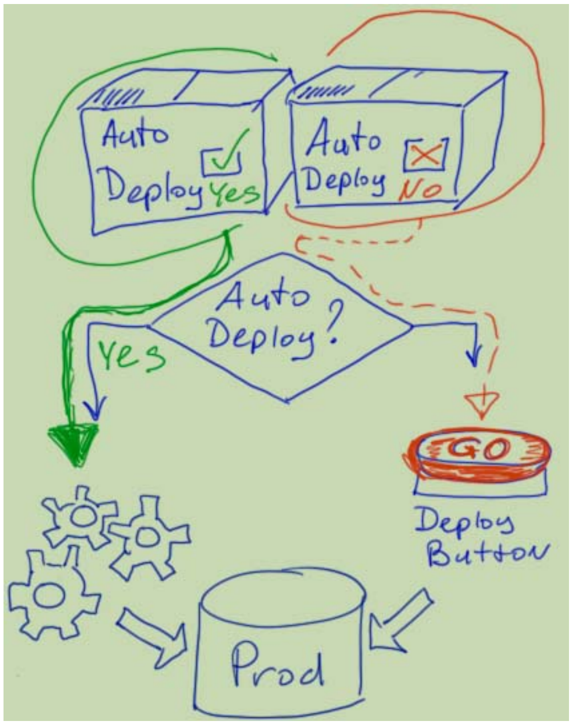
- Follow the same principles outlined in section 4, plus:
- Something fails all the time; it is human to make mistakes.
- Install TVs in the hallways, install them in the kitchen, hang them on an empty wall right by Steve Job's picture to increase visibility. Let your build process inspire your team.

9. Keep the set of pre-production environments. Configure automated deployments to them

- Create as many environments as needed, as many as you can afford;
- Use virtualization, use instances;
- Automate provisioning;
- Reuse existing hardware, move where it is needed, convert it to build agents, if you do not need it - donate to another team;
- Configure automated deployment to all of your environments
- Use the same tools to deploy to production and to pre-production. You want to know what fails before it fails in production;
- Do not release anything if a build fails, even if it is one-insignificant-tiny-tiny unit test, know that this unit test was there for a reason;
- If something failed - stop, think, create a unit test to check for failing conditions, change your process to prevent future failures;

10. What to deploy? What artifacts?

- The primary deployable artifacts in the production CI is the Delta Source Code that contains scripts to load well-tested code or by now well known data.



11. Tune the process to the point of continuous deployment to production

- At first, define what is the "Auto-Deployable Release" that team can accept.
- Create guidelines to identify Manual Release. A manual release is the release that should not be deployed automatically. Make guidelines available for every team member.
- Try to get code reviews, documentation, signoffs, approvals and other paperwork while the new product is moving through development pipeline. By the time, it reaches the user acceptance testing (UAT) environment, you would know if the release is auto- or manually deployable.
- Do not push the release to production if anything fails in a build, even if it is one-insignificant-tiny unit test.
- Use the same tools to deploy to production and to pre-production environments. Deployment tools should use the same technology, the same workflow, and the same process to deliver changes. You want to catch failures before they take down production environment;
- Use the same tools to deploy automated and manually released with the main difference being a step that requires human interaction - push a button to deploy Manual Release to prod.

Be realistic. Not everything could (nor should) be automated. You have just to tune the process to the point of continuous deployment to production of all the changes that can be automated.

PhpDependencyAnalysis

Marco Muths @mamuz_de, www.mamuz.de

PhpDependencyAnalysis is an extendable open source static code analysis tool for object-oriented PHP projects. It builds a dependency graph for abstract datatypes (classes, interfaces and traits) based on their namespaces. Abstract datatypes are aggregatable to units on a higher level, like layers or packages. It detects cycle dependencies and verifies the graph against a custom architecture definition.

Web Site: https://github.com/mamuz/PhpDependencyAnalysis
Version tested: 0.*
System requirements: PHP>=5.3, GraphViz
License & Pricing: Open Source, MIT license, free
Support: https://github.com/mamuz/PhpDependencyAnalysis/issues

Why I needed it?

It is essential in mid-size or large-scale software projects to explicitly manage the dependencies over the complete software lifecycle. Ignoring this approach will lead to a maintenance nightmare (Big Ball of Mud).

Characteristics like testability, expandability, maintainability and comprehensibility are achieved using an active dependency management. Technical quality can't be achieved only with testing. The architects and developers must design the software right from the start to meet some quality criteria. This design is mostly described in an architecture definition based on principles like Separation of Concerns, Law of Demeter and Acyclic Dependencies Principle by cutting the system into horizontally layers and vertically packages and a ruleset of engagements.

After writing and testing code, you have to verify that the implementation meets the architecture definition. Violations have to be solved to ensure a high maintainability for a long software life. For some programming languages you can find tools for this, but not for PHP. This is why I created PhpDependencyAnalysis.

How it works

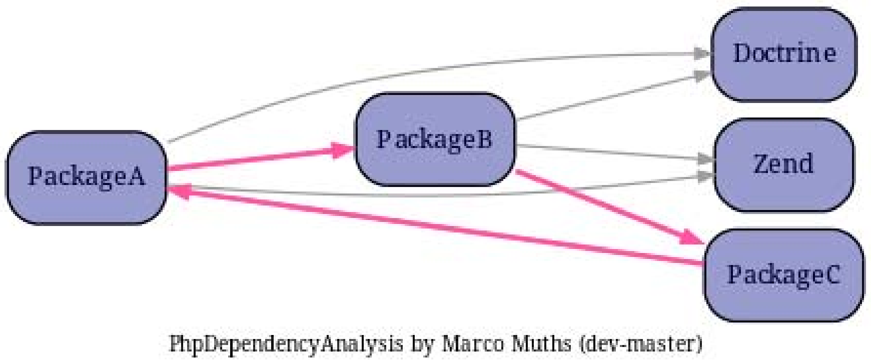
PhpDependencyAnalysis parses PHP-files to extract all containing Abstract Datatypes (ADTs).

An ADT can be an interface, a trait or a class (concrete, abstract or final). Afterwards each ADT is converted to an abstract syntax tree and all nodes of this tree will be traversed. Comments like PHPDocBlocks are also abstracted as a node. During the traversing process, each node is visited by registered collectors (visitor objects). A collector inspects the node to collect a dependency to current ADT.

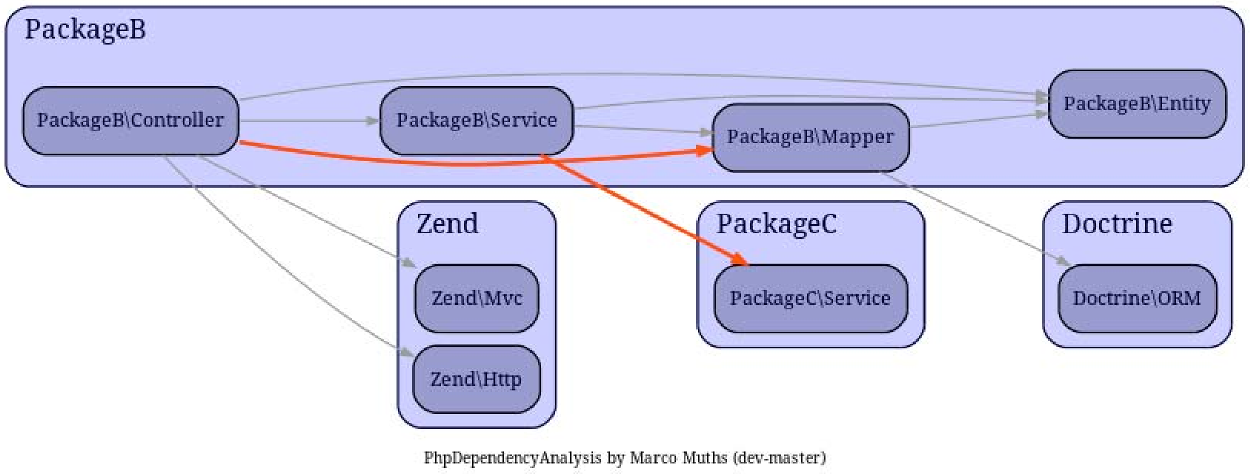
A dependency can be type of inheritance or type of call. Inheritance is defined by implementation, class-extending or a trait-use. Call is defined by a method parameter, by a method return-value or by an instance creation. A dependency is identified by their FQCN (Fully-Qualified Class Name). Those namespaces are sliceable to their namespace parts. This makes possible an aggregation of dependencies to a specific level, such as package-level, layer-level or component-level.

PhpDependencyAnalysis will create a directed graph of all ADTs (nodes) and their dependencies (edges). Cycles (detected automatically) and violations detected by your defined reference validator (Architecture Definition) are highlighted in red.

Example with detected cycle



Example with detected violations by using custom architecture definition



Features

In most cases, large PHP projects generate a big dependency graph that is not readable for humans. To address this issue, PhpDependencyAnalysis is customizable to change the scope of the inspection. For instance you can filter namespaces to have a view on package level or you can parse a specific package only or a specific PHP file. You can also exclude namespaces from the inspection.

Most PHP projects are using IoC-Containers for Dependecy-Injection, like ZF2-ServiceManager or Pimple. All of them provide objects by a string representation, which are not resolvable to dependencies in general. For this reason this tool is extendable with user-defined plugins to meet this requirement. Another useful extension would be a user-defined plugin to detect database tables or REST services as a dependency by parsing string nodes.

List of features:

- Providing high customizing level
- Dependency graph creation on customized levels respectively different scopes and layers
- Supports Usage-Graph, Call-Graph and Inheritance-Graph
- Dependencies can be aggregated to a package, a module or a layer
- Detecting cycles and violations between layers in a tiered architecture

- Verifying dependency graph against a user-defined reference architecture
- Collected Namespaces of dependencies are modifiable to meet custom use cases
- Printing graphs in several formats (HTML, SVG, DOT, JSON)
- Extendable by adding user-defined plugins for collecting and displaying dependencies
- Compatible to PHP7 Features, like Return Type Declarations and Anonymous Classes

Requirements

This tool requires PHP >= 5.3.3. Besides this you need GraphViz (http://www.graphviz.org/) for graph creation on your machine. This is an open source graph visualization software and is available for the most platforms.

PHP is a dynamic language with a weak type system. It also contains a lot of magic statements, which resolves only during runtime. This tool perform static code analysis and thus has some limitations.

Here is a non-exhaustive list of unsupported PHP features:

- Dynamic features such as `**eval**` and `**$$x**`
- Globals such as `**global $x;**`
- Dynamic funcs such as `call_user_func`, `call_user_func_array`, `create_function`

The cleaner your project is, the more dependencies can be detected. Or in other words, it is highly recommend to have a clean project before using this tool.

You should apply naming conventions to have a greater chance for a generic detection of violations between packages and layers. Each package should have its own namespace and this is also true for each layer.

Example for a naming convention in an User-Package

- Namespace for Controllers: `**User\Controller\..**`
- Namespace for Listeners: `**User\Listener\..**`
- Namespace for Services: `**User\Service\..**`
- Namespace for Mappers: `**User\Mapper\..**`

In this case, you can declare classes, which use `**User\Controller**`- or `**User\Listener**`-Namespace, belongs to Application-Layer. Application-Layer is permitted access to Service-Layer (`**User\Service**`-Namespace) and is not permitted access to the Data-Source-Layer (`**User\Mapper**`-Namespace).

Further Reading

Project and Install Guide: https://github.com/mamuz/PhpDependencyAnalysis

Documentation and Examples: https://github.com/mamuz/PhpDependencyAnalysis/wiki

Watai - Integration Testing for Web Applications and Components

Matti Schneider, @matti_sg, mattischneider.fr

Watai (Web Application Testing Automation Infrastructure) is an open-source, declarative full-stack web testing framework. It is both a test runner engine and a set of architectural patterns that help front-end ops-conscious [1] developers create maintainable and solid end-to-end tests. These tests are written as text files that simply describe the components of the application and how the user is expected to interact with them. Watai uses them to automate web application navigation through actual browsers, checking whether such interactions present the expected information to the end users along the way, including through asynchronous operations. This allows automating demonstration of features, detecting regressions when updating code and, since the same tests can be run in almost any browser [2], easily checking cross-browser functional support.

Web Site: https://github.com/MattiSG/Watai/
Version tested: 0.7.
System requirements: Node.js ≥ 0.8, NPM ≥ 2, Java ≥ 1.4 and installed browsers.
License & Pricing: Free & open-source (AGPL).
Support: Community, through GitHub issues and wiki.

Context and vocabulary

“Integration” testing is a testing level in which a combination of individual software modules is tested as a group. In the context of web applications, it means testing the result of combining client- and server-side code, and accessing it through a browser.

Based on this definition, you might have already heard about what I call here “integration testing” under a different name. You might have read about “customer”, “end-to-end” (also abbreviated as “E2E”), “system”, “acceptance”, “validation” or even “functional” testing. These are many synonyms for almost the same reality. For consistency, I will only refer to it in this document as “integration testing”.

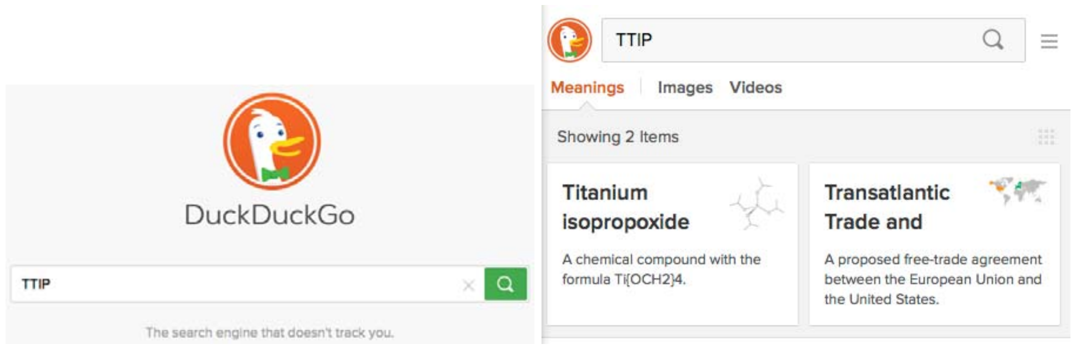
Watai is a tool for integration testing that focuses on making it easy for developers to write user-oriented tests that consider the system under test as a black box, yet are maintainable enough to be worth investing in. These tests are simple JavaScript files that are parsed by Watai and transformed into Selenium (WebDriver) instructions. By using WebDriver to automate browsers, it makes it easy to play tests in any browser, and thus to fight against platform discrepancies, usually referred to as “cross-browser issues”.

Example

We will present a test for the DuckDuckGo search engine. This engine is just like Google, except that it doesn’t track you, and that it has a nice disambiguation feature: the “Zero Click Info box”.

Let’s imagine we are engineers at DuckDuckGo, and we want to test if a Zero Click Info box is shown for an acronym: “TTIP”.

Why are people around the world using this acronym? Are they talking about a chemical compound? Or about a US-EU free trade agreement negotiated in secret? The result page should help our users get a quick answer!



A chemical compound with the formula Ti(OCH2)4.

A proposed free-trade agreement between the European Union and the United States.

Based on the above requirements, we can formalize the following testing scenario:

1. Go to duckduckgo.com (English version).
2. Type “TTIP” in the search field.
3. Submit the search.
4. Ensure that the text “Transatlantic Trade” is presented in the “meanings” header ribbon.

The test suite automating the above scenario is already available at https://github.com/MattiSG/Watai/tree/master/example/DuckDuckGo. We see a *Configuration* file, two *Component* files, a *Scenario* and a *Fixture* file. These are all the parts of a Watai test.

Configuration

The *config.js* file exports a JavaScript object with two mandatory key-value pairs:

- *baseURL*, that describes the URL at which the test should start.
- *browser*, that defines which browser should be used to run the test.

Components

Components are defined in files ending in *Component.js*. They represent a graphical component of the tested web application. A component provides a mapping from DOM elements, identified through selectors, to testing-environment attributes.

Open the ZeroClickComponent.js file: it is a simple JavaScript object property.

```
// example/DuckDuckGo/ZeroClickComponent.js
meanings: '#zci-meanings'   // this is an element definition
```

Our header definition is here as an element. It maps a meaningful, testable name to a CSS selector describing how this element is retrievable in the DOM. We will now be able to retrieve the content of the Meanings section of the Zero Click Info box.

Interface components in modern web apps are not only sets of elements. They can also offer complex actions to the user, triggered by basic interaction events such as clicks or keystrokes on specific elements. This is why a component is a bit more than a simple DOM map: it also abstracts action sequences and makes them callable directly through a meaningful name.

Open the SearchBarComponent.js file: there are once again elements, but there is also an action defined.

```
// example/DuckDuckGo/SearchBarComponent.js
field:        'input[name=q]',            // this is an element
submitButton: '#search_button_homepage',  // this is an element too

searchFor: function searchFor(term) {     // this is an action
    return  this.setField(term)()
                .then(this.submit());
}
```

The *searchFor* key is a JavaScript function that takes a parameter and sequences two basic steps: typing the passed string in a field, and clicking a submit button. With this action, we will be able to search for an ambiguous term, as our scenario needs.

Fixtures

We then need to define what we want to search for, and what we expect to be presented with. Open the ZeroClickFixture.js file: this is where we define such fixture.

```
// example/DuckDuckGo/ZeroClickFixture.js
query = 'TTIP'
expandedAcronym = /Transatlantic Trade/
```

You can notice that we are inputting a specific string, but we only expect results to match a given RegExp rather than defining exactly what they will be. This improves the resilience of our tests.

Using fixtures files to store values is optional, but it is recommended as a way to decouple concerns even more. In our example, if we wanted to test another ambiguous term, there is no hesitation where to define it. You have simply to open the fixture file, replace “TTIP” with the new term, and restart the test. A fixture file is pure JavaScript code, and has to end in “Fixture.js”. All the variables defined inside will be made available to all components and scenarios in the same suite.

Scenarios

So, we have now defined all needed elements, actions and fixtures for our scenario. However, we have not yet defined how these pieces fit together, nor what we are going to check. This is what a scenario is for. Let’s look at the 1 - ZeroClickScenario.js file.

```
// example/DuckDuckGo/1 - ZeroClickScenario.js
description: 'Looking up an ambiguous term should make a Zero Click Info box
appear.',

steps: [
    SearchBarComponent.searchFor(query),
    { 'ZeroClickComponent.meanings': expandedAcronym }
]
```

Just like a component, a scenario is series of object properties, this time with two mandatory keys. The first is *description*, mapping to a String presenting what expected behavior you are testing. The second expected key in a scenario is *steps*. Its value must be an array.

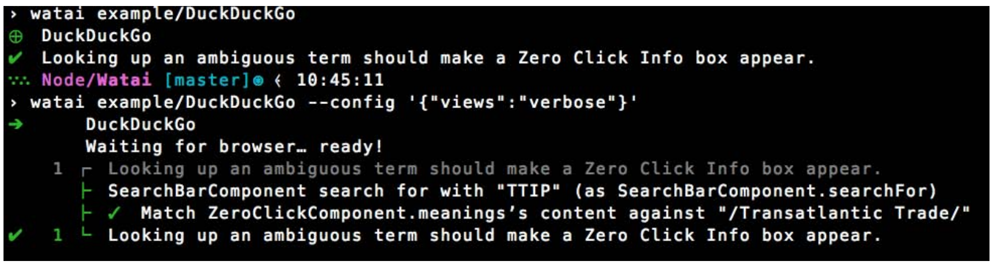
In our example, the first line is simply a call to the action we defined in the *SearchBarComponent*: we are going to *searchFor* the term we defined in the fixture: *query*. You can define as many steps in a scenario as you want. A step can be, like here, an action, a custom function, or a state definition. All these steps will always be executed in order.

We define how our components will be used with scenario steps, but we still don't know how to actually check behavior! Assertions in Watai are implicit. That is, you won't have to assert anything, unless you want to. You simply define an expected state, and the actual state of the current web page will be checked against it. A state definition is simply a hash where keys reference some elements, and the corresponding values are matched against the values of the targeted DOM elements. The only tricky thing is that you can't have an actual reference as a hash key, only its name. So, remember to quote keys when defining states.

Executing the suite

To execute a test suite locally, you will first have to install dependencies. You will need Node.js (installable from nodejs.org), and a Java runtime (java.com/download). Then, simply run `npm install --global watai selenium-server` at a command prompt, and start the Selenium server by running `selenium &`. You can then try the suite we examined above by downloading it as a ZIP (github.com/MattiSG/Watai/archive/master.zip) and running it with `watai <path/to/downloaded/archive/example/DuckDuckGo>`.

A dynamic progress indicator will be displayed in your terminal, and you will be able to follow through the test suite. Several views are available, including a more verbose one, suitable for continuous integration, a WebSocket-based one that allows interaction with external views. The exit code [3] will always be non-0 if a test fails, allowing you to integrate with any pipeline.



```
› watai example/DuckDuckGo
⊕  DuckDuckGo
✔  Looking up an ambiguous term should make a Zero Click Info box appear.
∵∴ Node/Watai [master]● ‹ 10:45:11
› watai example/DuckDuckGo --config '{"views":"verbose"}'
➔      DuckDuckGo
       Waiting for browser… ready!
    1 ┌ Looking up an ambiguous term should make a Zero Click Info box appear.
      ├ SearchBarComponent search for with "TTIP" (as SearchBarComponent.searchFor)
      ├ ✓  Match ZeroClickComponent.meanings's content against "/Transatlantic Trade/"
✔   1 └ Looking up an ambiguous term should make a Zero Click Info box appear.
```

You are now ready to write your own test suites, using this example and the thorough reference documentation available at github.com/MattiSG/Watai/wiki!

A gentle reminder

All levels of testing are important, and doing tests only at one level is not a proper software testing strategy. You should have unit tests for your application, both on the server- and on the client-side. If your application is complex, adding functional-level tests is a good thing. Integration tests validate expected behavior, and Watai largely reduces their barrier to entry, but they still cannot help in pinpointing the origin of a failure as much as unit tests can.

References

1. Front-end ops: http://smashingmagazine.com/2013/06/front-end-ops/
2. The supported browsers are the ones supported by WebDriver, i.e. almost all: http://docs.seleniumhq.org/about/platforms.jsp
3. https://github.com/MattiSG/Watai/wiki/Exit-codes

SQE Training's 2016 ISTQB® Certification Classes In 2016 SQE Training will be offering ISTQB® certification classes for Foundation Level (CTFL), Agile Extension, Advanced Level Test Manager (CTAL-TM), Advanced Level Test Analyst (CTAL-TA), and Advanced Level Technical Test Analyst (CTAL-TTA). SQE Training offers 60+ courses across the SDLC. To see details on all courses, plus dates and locations, view our 2016 catalog on http://www.sqetraining.com/training/demo/2016-training-catalog

The Software Development Jobs Board. Are you looking for a developer, a tester or a project manager? The Software Development Jobs Board is your efficient and low-cost option to reach directly the 43'000+ software development professionals worldwide that subscribe to the Methods & Tools newsletter plus all the people who visit directly the job board. Post your open software development positions for $10 or less on http://www.softdevjobs.com/

SoftwareTestingMagazine.com is dedicated to present industry news, articles, blog posts, book reviews, tools, videos and other resources about unit testing, integration testing, functional or acceptance testing, load or performance testing in software development projects. Your will also find content linked to software quality assurance, test automation and new approaches like Behavior Driven Development (BDD) or Test Driven Development (TDD). Go to http://www.softwaretestingmagazine.com/

Advertising for a new Web development tool? Looking to recruit software developers? Promoting a conference or a book? Organizing software development training? This classified section is waiting for you at the price of US $ 30 each line. Reach more than 50'000 web-savvy software developers and project managers worldwide with a classified advertisement in Methods & Tools. Without counting the 1000s that download the issue each month without being registered and the 60'000 visitors/month of our web sites! To advertise in this section or to place a page ad simply http://www.methodsandtools.com/advertise.php

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 www.martinig.ch
Editor: Franco Martinig ISSN 1661-402X
Free subscription on : http://www.methodsandtools.com/forms/submt.php