BETTER CODE BOOK 2015

FEATURING NDEPEND'S MVPS
ON CODE QUALITY

Introduction

Dear Reader,

First of all, I want to thank you for downloading this booklet of collected works.

I started making NDepend because code quality and standards were important to me. As the world is increasingly relying on technology and software to function, solutions have become more and more complex. It is more important now than ever for companies and individuals to create beautiful code.
Writing has been important to both the success of NDepend and the spread of code quality and standards as an integral part of development. We, the NDepend team, are grateful and humbled by the contributors of this booklet, and all the people who have written about NDepend and code quality. As a writer myself, I know it is difficult sometimes to find time and energy to write something great, so thank you.

Contained herein are some of the best written articles from our community. I hope you will find them as inspiring as I have.

Again, I want to thank you all for taking the time to create exceptional works.

Patrick Smacchia

Table of Contents

The Importance of Static Code Analysis

Anthony Sciamanna

Static code analysis is a critical tool for development teams who value code quality and continuous improvement. My most recent experience with static code analysis tools is with NDepend for .NET. So I will specifically discuss experiences I've had with that tool. However, a lot of the ideas in this article can apply to a majority of the static analysis tools that are currently available.

If you have read what I have written about coding conventions, you know how important I think coding conventions are to a development team. Once conventions have been documented, the next step is to enforce them via a static analysis tool.

Rule Overload

One of the first things you will notice is that the majority of these tools come pre-configured with a set of static analysis rules. I recommend that you look at all of them closely. You may find that you are missing some critical rules and want to revise your team's coding conventions accordingly. While I think this is a great idea, be mindful that the coding conventions should be short. So resist the urge to go back and add every rule into your coding conventions. Not all of the rules will provide the same benefit to your team. Some rules will not provide any benefit. You should focus on the rules that will provide the maximum benefit, and like everything else in software development, iterate on your static analysis configuration. NDepend in particular does a great job of providing a comprehensive set of critical and non-critical rules out of the box. But you will still want to modify them and create your own to match your team's coding conventions.

Critical and Non-Critical Rules

NDepend has the concept of critical and non-critical rules. Critical rules are ones,

that if violated, will break your build. These should be reserved for your team's coding conventions and other serious code quality offenders. The non-critical rules should still be enabled so that your team can continue to monitor them without failing the build. Non-critical breaches that continue to increase are a problem and you will want to address those accordingly.

This difference highlights two of the ways that you should be using the tool to get the maximum benefit for your team. The critical errors should fail the build immediately and require a developer change before there will be another successful build. The non-critical rules along with other metrics collected via your static analysis tool, cyclomatic complexity and coupling for example, shouldn't necessarily fail the build, but be part of a report that the team examines regularly. Armed with this information, the team can focus their refactoring and clean-up efforts in a way that addresses the most problematic parts of the codebase first.

An Early Warning System

Bryan Helmkamp (founder and CEO of Code Climate) gave a fantastic talk at Baruco 2013, Building a Culture of Quality. In his talk he describes that the natural trajectory for a software project's quality is down. Because of this, we developers need to employ several techniques to prevent this from happening. One recommendation is to implement an early warning system. Part of your early warning system should be a static analysis tool.

There are several factors that can create environments where code quality suffers. These include schedule pressure, critical bugs that need to be fixed and deployed to production quickly, and changes in the development team members, just to name a few. Furthermore, refactoring is a challenging skill to learn so teams may struggle to make the code better when adding features or fixing bugs. Even the best teams with agreed upon coding standards can suffer this fate. By putting developers who believe code quality is subjective, or have the "just get it done" attitude on teams who have no coding conventions you are creating a recipe for disaster.

This early warning system, your critical static analysis rules, is your safety net against code quality deteriorating over time. You will be notified as soon as you breach a critical rule, which is the optimal time to fix the issue. You can fix these issues as they arise as part of your daily development process. Otherwise, these issues will accumulate until they become a much bigger problem which will cost

your organization a lot more time, money, and skill to reverse.

Overwhelmed by Legacy Code

If you are in a situation where you are working on a team that has ownership of a large amount of legacy code, static analysis tools will help focus your efforts to improve the code quality. Legacy code has several definitions and I will start with Michael Feathers'

> *To me, legacy code is simply code without tests.*
> *— Michael Feathers, Working Effectively with Legacy Code*

However, it is important to mention that often, a lack of unit tests and code that is untestable go hand in hand. If there aren't any unit tests or if the ones that do exist are terribly complicated, you can be fairly certain that the code under test is poorly designed and implemented.

You may find yourself in a situation where you now have ownership of a large amount of code that is tightly coupled, not cohesive, has no unit tests, contains large classes, the classes contain large methods, and there are a lot of static global classes and methods making it even harder to modify. And all of this code exists in the context of a larger system without an architecture where there are no boundaries or separation of concerns. This has happened to me more than once and it can be overwhelming. Being in these situations can quickly increase the team's stress level and decrease the team's morale as they feel like improving the quality is an impossible task.

It is in these situations that a static analysis tool can help your team determine a path to start chipping away at the worst code first. The tool can give you instant feedback as to the progress of your team and can start changing that stress and low morale into a feeling of accomplishment and forward progress.

The Boy Scout Rule & Opportunistic Refactoring

Some may think using a static analysis tool in this way works against the Boy Scout Rule (coined, I believe, by Uncle Bob Martin) or Opportunistic Refactoring techniques by Martin Fowler. Both of these techniques describe cleaning up the code you are currently working on. If you are not familiar with the Boy Scout

Rule, Uncle Bob describes it as always checking in the code you are working on a little cleaner than you found it. This is analogous to the Boy Scouts, who leave the campground cleaner than they found it. This same sentiment is echoed in Martin Fowler's writing on Opportunistic Refactoring.

Static analysis tools can be used in conjunction with these other refactoring techniques to optimize your approach to cleaning up the code. While still utilizing opportunistic refactoring techniques, the type and extent of the refactorings can be determined by the static analysis rules that are currently being breached.

Metric Visualization

Static analysis tools use a variety of techniques to visualize metrics. These metric visualizations are a great place to start when trying to determine where to focus refactoring efforts when faced with a large amount of legacy code.

Treemaps in NDepend

NDepend uses treemaps to visualize metrics which I have found to be incredibly useful. A treemap is a visualization algorithm to display data via nested rectangles. These rectangles can represent various code elements in the system (including methods, namespaces, types, and a few others). The size of the rectangle represents one metric (e.g., method length or cyclomatic complexity). The color of the rectangle is used to represent a second metric like test coverage. This enables developers to correlate two metrics and use this information to determine refactoring techniques and priorities.

Pressure to Ease the Rules

If you don't have ownership of your entire codebase or you just took ownership of a large legacy codebase, you may need to relax the rules early in the process. While this isn't ideal, you can use the concept of ratcheting to improve the software to the point where you can enable all of the critical rules.

Ratcheting

Ratcheting is a technique used to ensure that the overall codebase is getting better over time by introducing a practice gradually as described by Jez Humble in his book Continuous Delivery. In his example, he describes that early on in a practice's adoption, the build would not be configured to fail on a single rule breach. His examples include compiler warnings or TODO comments in the code. By employing a ratcheting technique, the software build would fail if the number of these breaches increased as compared to the previous build. If the development team is more aggressive about improving the software quality, the build could instead be configured to pass only if the number of these breaches decrease as compared to the previous build.

You can employ this same technique at a more granular level to determine if a specific rule should break the build or not. For example, you may have a rule that states a class can't be more than 100 lines of code. If a legacy class is 300 lines of code and stays that size or gets smaller, the rule can continue to pass. But if it becomes 301 lines of code, the rule will break the build. It's also important to configure these rules so that all new classes will breach the rule if they are larger than 100 lines of code. NDepend's CQLinq queries allow the creation of these regression type rules.

Don't Go Backwards

Once you have rules in place you may feel the same kinds of pressure I've mentioned earlier which may encourage you to ease the rules. While it may feel like the "pragmatic" thing to do in the moment, I recommend that you fight this urge as you will lose your early warning system.

Conclusion

I will be writing more about static analysis tools and NDepend as I'm barely scratching the surface of the capabilities these tools provide. In the meantime, give one of these tools a try on your team and you'll see that there are significant benefits that can be gained by having detailed analysis of your codebase on every build.

Anthony Sciamanna

Anthony Sciamanna is a software developer from Philadelphia, PA who has worked in the industry for nearly 20 years.

He specializes in leading and coaching development teams, improving development practices for cross-functional teams, Test-Driven Development (TDD), unit testing, pair programming, and other Agile / eXtreme Programming (XP) practices.

He can be contacted via his website: anthonysciamanna.com

Decoupling Legacy Code Using NDepend

Bjørn Einar Bjartnes

Decoupling a Legacy Code Base is Hard

Decoupling legacy code bases is not only hard, but often we don't even have a clear idea of the current couplings that exist in our system. Without a clear overview of the current state we can't make sound decisions on what we should do to improve. This fact leads to refactorings that are subjective, these refactorings might not yield value except for some subjective measure of *"less ugly"*.

I will deliberately leave out a generic discussion on the different types of coupling and cohesion in code, but I will restrict myself to large assemblies with low cohesion that typically comes from splitting up monolithic applications such as ours. They tend to have large dependencies, each with its set of transitive dependencies, many of which you do not need in your application. As legacy code is already hard enough to reason about, the last thing we want to carry around is more code than we need. Also, large shared assemblies with low cohesion causes small changes to require disproportionally large parts of the system to be rebuilt, redeployed and re-tested.

By using static analysis, we can make a map of the current couplings in our system, make a plan for refactorings based on that map, and verify that we have obtained the goals we set out to accomplish by re-running the analysis. This can make refactoring easier, cheaper and yield code-bases that can be proven to be more modular.

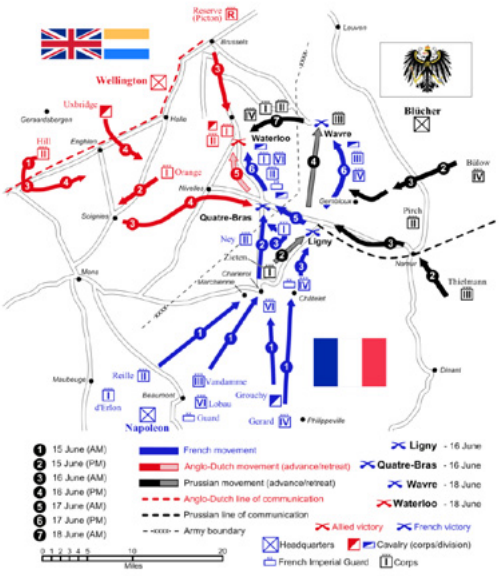
Refactoring Strategies and Maps

There are many strategies that one can use to obtain a more maintainable code-base, for example utilizing a domain-driven approach, reduce unwanted coupling, refactor to meet some code-metric or refactor according to SOLID principles, all described in the literature.

I am a big fan of the strategies that lend themselves well to drawing maps because they put me in the role of a general. Maps visualize the current situation and they provide an opportunity to make high-level design decisions and prioritize where to focus our efforts.

The life of a general is a nice break from the daily life in the trenches. In the trenches, we battle legacy code line-by-line, class by class, in hand-to-hand combat, armed with only a keyboard and our cunning friend ReSharper, always there to suggest that any problem can be solved by hitting ALT+ENTER.

In the trenches, it's developer against code in a messy, brutal fight that leaves both sides bleeding with infected wounds in a muddy field. A general, on the other hand, can enjoy a hot beverage, miles away from any messy action on the ground.



Strategic maps must always be true to the facts. Beauty should come from truth – not because undesirable facts are left out. Don't try to make the map look like something from a book – it should reflect reality. I find inspiration in military maps, sometimes showing rivers, bridges, towns and other details of the real world that affect different strategies. Rain and mud affected the outcome of Waterloo, illustrating that it's not easy to decide which details to leave out and which to leave in.

Static analysis

> *Static program analysis is the analysis of computer software that is performed without actually executing programs*

We use NDepend to do static analysis. It plugs easily into Visual Studio and TeamCity and provides many types of analysis with interesting code metrics. I'll stick to the analysis of dependencies and I'll also stick to the dependencies between assemblies, and leave modules as namespaces out for now.

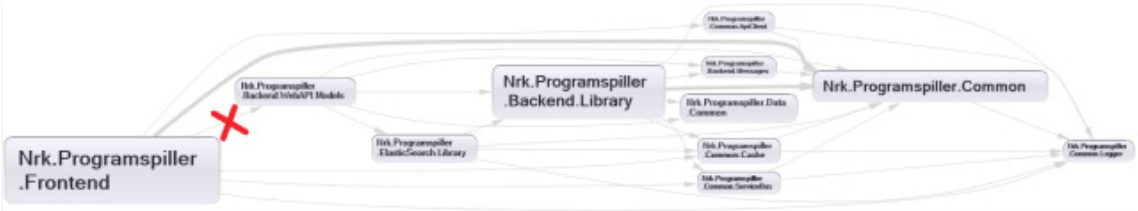
NDepend will automatically provide an analysis and make diagrams of your code based on simply scanning all assemblies in your build folder. The only thing I've done to make these diagrams is filter the assemblies with "Nrk" and selecting "Include application assemblies only", leaving frameworks- and external libraries out of the diagrams.

Splitting frontend and backend

We have a solution where the former monolithic tv.nrk.no has been split into a front-end (tv.nrk.no) and a back-end (psapi.nrk.no). The front-end does not access databases directly anymore, but accesses all data through psapi.nrk.no over HTTP, just as if it was a smart-TV or mobile client. This is good!

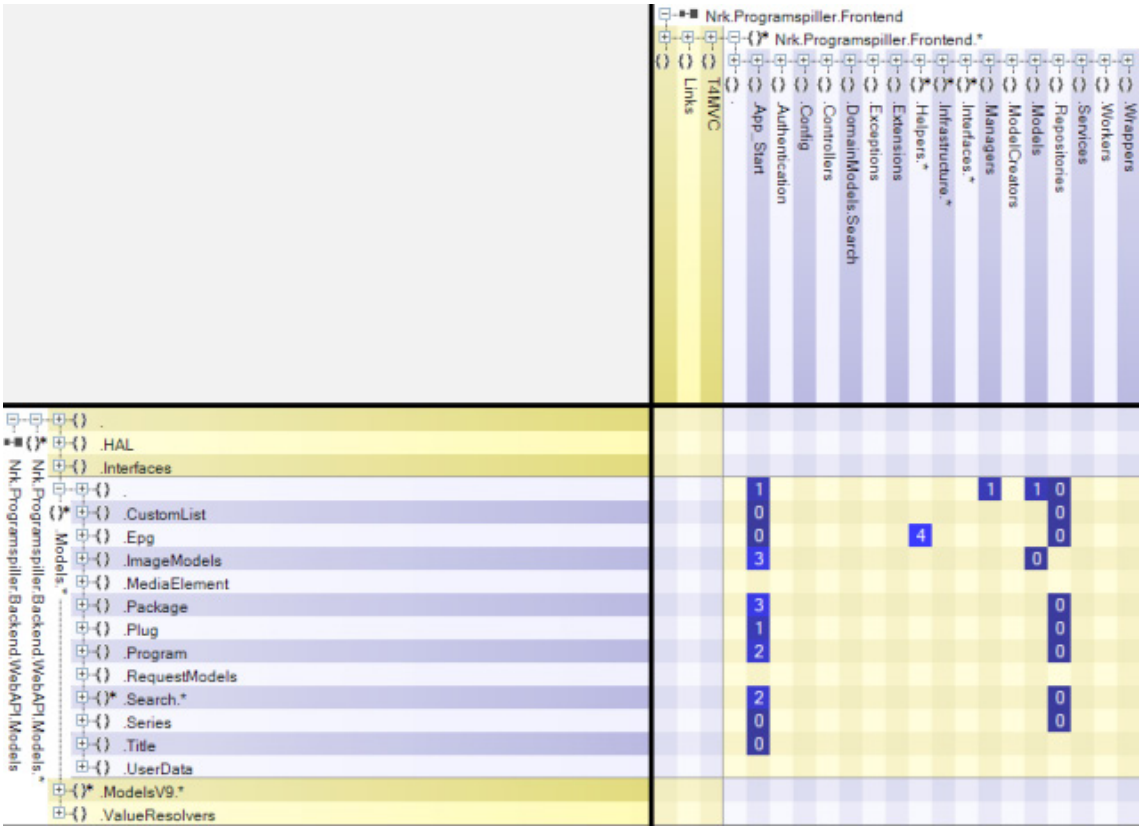
In the team, we talk about the front-end and the backend as separated, but when looking at the dependency graph, we could see that five assemblies in the front-end project contained backend in the name, or where referenced transitively by backend assemblies. We wanted the backend to be separated, but in fact it was an uncompleted "Siamese twins" operation. Couplings were still in place. Contrasted with our view of the system driven by illusions and feelings, static analysis doesn't care too much about your hopes and plans for the refactoring, but mercilessly maps out all dependencies between assemblies, and internal dependencies between namespaces in your assemblies.

The strategy is pretty clear from the map – the red cross (added by the Strategic Command) marks the dependency we want to get rid of. Getting rid of Backend.WebAPI.Models will also get rid of four transitive dependencies.

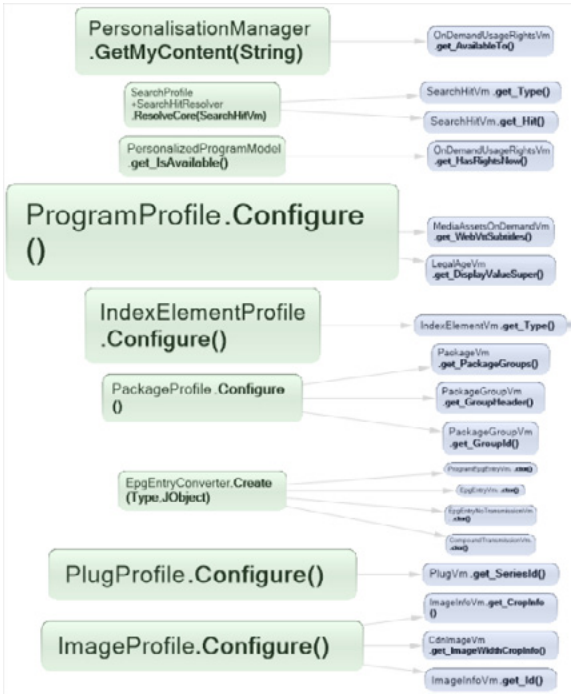


NDepend analysis before refactoring. The red cross added by the author

This map serves as a basis for implementing our strategy of reducing coupling. This is our high level plan for our first refactoring campaign. NDepend will help us further by listing the exact couplings that must be removed to remove the reference.



Coupled namespaces – the 0's is due to indirect couplings, NDepend marks the connection but you can select methods/ types/ indirect references etc. as a basis for the numbers



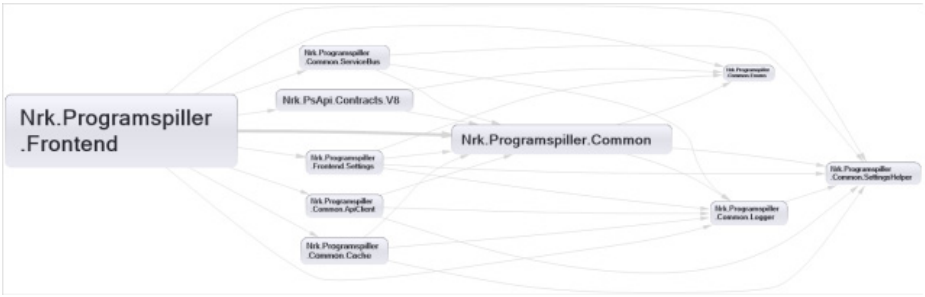
Code connecting the assemblies

For a more concrete view, we can list the connections directly. This is well suited for components that aren't too tightly coupled. This is also the point at which you realize being a general is easy, but being in the trenches and cutting through enemy lines is painful and will leave you scarred.

Still, this to-do list of things to remove is much more comfortable to work with than being all alone in enemy territory, with no plan and no map. We don't have to manually read and analyze the entire code-base, we can forget the bigger picture and simply focus on each objective one by one, knowing that when we have completed our list we have made a consistent set of refactorings.

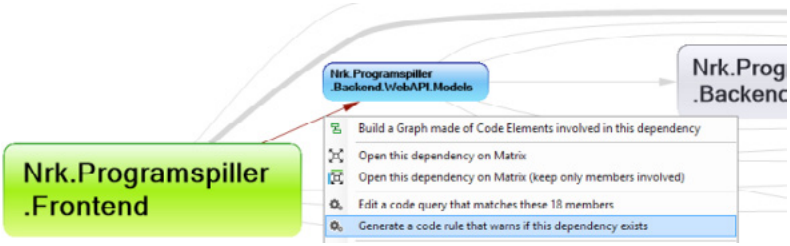
This is the time for team-discussions - before we start on how to best remove the coupling. It could be moving code between assemblies, duplicating code, not using helper methods from other assemblies, deprecating functionality, splitting assemblies etc., but it requires careful analysis to figure out what to do on a case-by-case basis. Principles such as SOLID could help facilitate these tactical design discussions and help to see the design challenges from multiple perspectives.

After the refactoring the size of the deployment package was reduced from 288 to 113 MB. The backend references are gone. Our work still isn't over though, the thick arrow to the Common module means that there are still a lot of shared code that should be looked at. However, having cut through the enemy lines, annihilated their supply lines to the backend and isolated them completely at the front, we should relax and celebrate a little before we push forward and refactor the remains.



Next steps

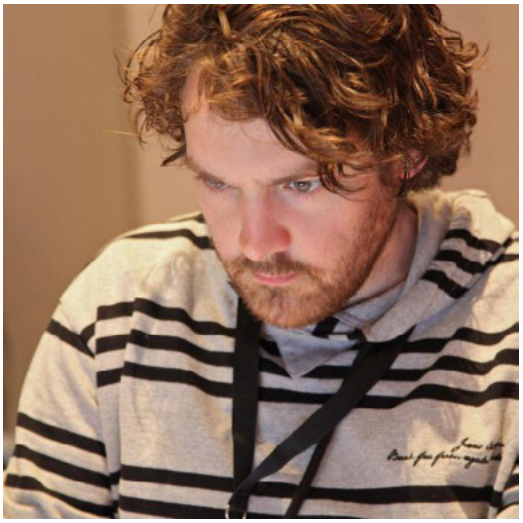
The next step from here is to add rules that warns if these dependencies come back. NDepend makes it fairly straightforward to make rules such as a rule that breaks the build if a dependency is made on an assembly containing the name "Backend". I like to think of these rules as tripwires, efficiently stopping unwelcome intruders from attempts at recovering the occupied territory.



Secure refactorings with tripwires

Our experience has been that this strategy works quite well, we have also applied it to other parts of the system. When we split functionality into smaller services, we make sure the entire monolith is not pulled in by accidental coupling.

Bjørn Einar Bjartnes



Bjørn Einar Bjartnes is a developer at NRK TV, the Norwegian Broadcasting Corporation. His current role is a backend developer at the API team, serving web, mobile, TV clients and more metadata about programs- and video-streams. He holds a MSc in Engineering Cybernetics and has a background from the petroleum industry, which has probably shaped his view on systems design. Also, Bjørn is active in the local F# Meetup and a proud member of the lambda club, playing with all things useless related to computers.

He has recently begun writing for NRKbeta.

Why Code Metrics Are Important, and how NDepend can help You!

Jack Robinson

Before we start, a small disclaimer - I don't often write pieces about software, so this is unknown territory for me. In my usergoup talk, I gave a whirlwind tour of the NDepend Universe, something I hope to replicate here, but it is in no means a full spread of the capabilities of NDepend - just the other day we managed to string together a code query to determine the classes that have fields that implement System.Collections, before ranking them on their equivalent Google PageRank - the Swiss army knife of capabilities on your hands with this program is insane.

NDepend?

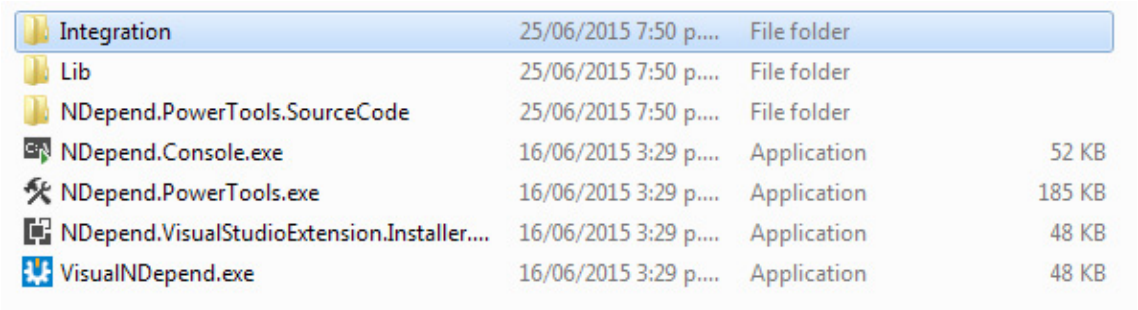
NDepend is a Static Analysis Tool for all things .NET. Developed by Patrick Smacchia and his team and NDepend, it runs and analysis over your code base, and provides a number of metrics and visualizations, as well as allowing you to run queries over your code in the form of LINQ statements, called CQLinq.

It comes in two different versions, Build Machine and Developer:

- The Developer version is priced at $325 USD, and offers almost full functionality of the GUI component. You'd generally use this if you wish to run the analysis on the whim of the, well, developer.

- The Build Machine license, at $488 USD is more for Continuous Integration environments, where you want the results to be visible to your entire dev team, through custom API wrappers, or through a tool like Team City.

On the mention of the NDepend API, I should probably describe what comes out of the box when you get NDepend:



There are five main components given on download:

- **Visual NDepend:** The GUI portion of NDepend, and perhaps the program you will use the most

- **NDepend Visual Studio Plugin:** I ran the Visual Studio plugin for v5, however, it seemed like it was the NDepend GUI inside of Visual Studio, so in that sense, I won't go into depth on this - I believe most of what is achievable in Visual NDepend is available through the VS plugin.

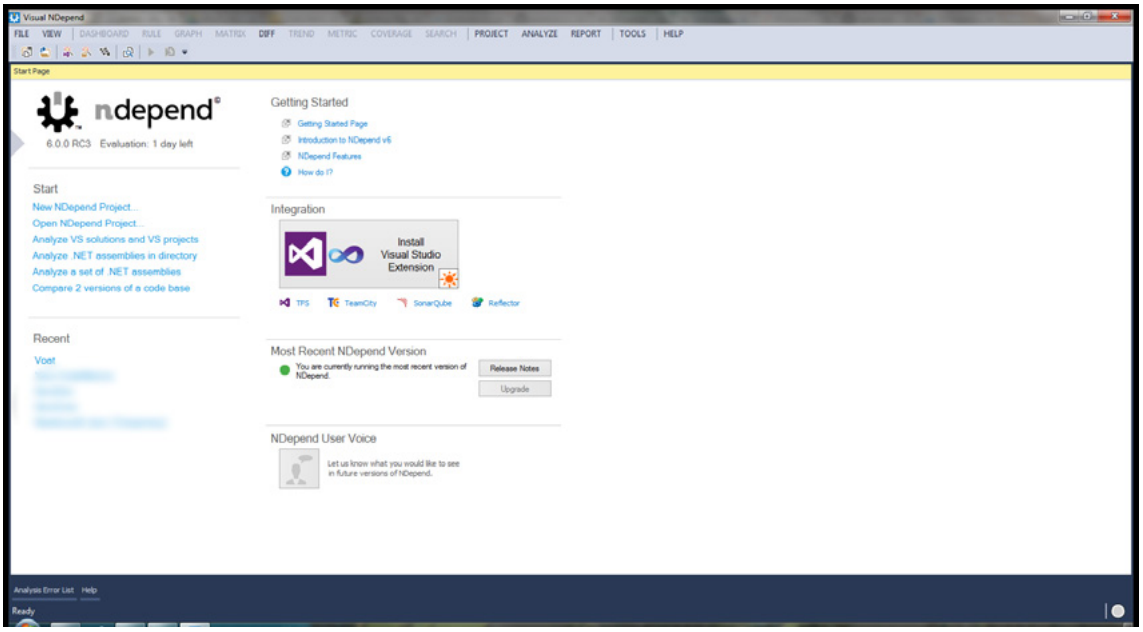
- **NDepend API:** Exposes a large number of the functionality in NDepend via a C# library. On a quick look, it seems like the API is the meat and potatoes of the show, and tools like the Visual NDepend make use of it. The NDepend Console also looks like a Command Line interface for the API, which is very useful in applications where you require the base functionality of NDepend without writing your own wrapper.

- **NDepend.PowerTools:** is an open source collection of examples using the NDepend API to act as a tutorial on the API functionality.

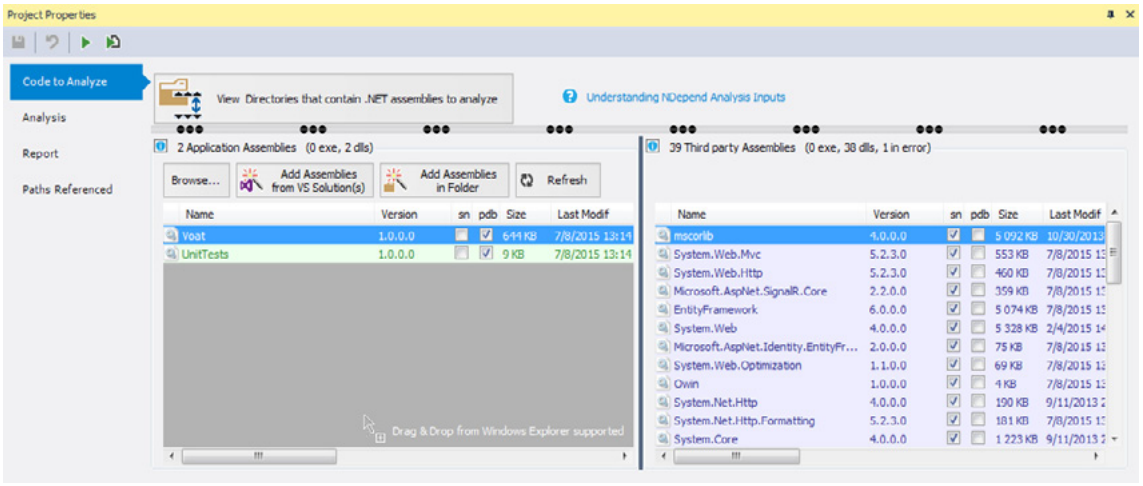
- **Integration:** This is a new addition in v6, and unfortunately I was unable to have a play around with it (I'm not too familiar with SonarQube, another metrics platform, nor TFS, since I generally use Github). Now, there is also supported integration with TeamCity, which excited me, and no doubt will play with it in the near future.

Visual NDepend

First things first, let's have a play with the GUI, since the power of NDepend can easily be seen here.



When you open the program, you're met with this dashboard, giving you the option to open a recent project, or create a new one. Voat is one I created earlier, the source code for the Reddit-clone Voat.co.

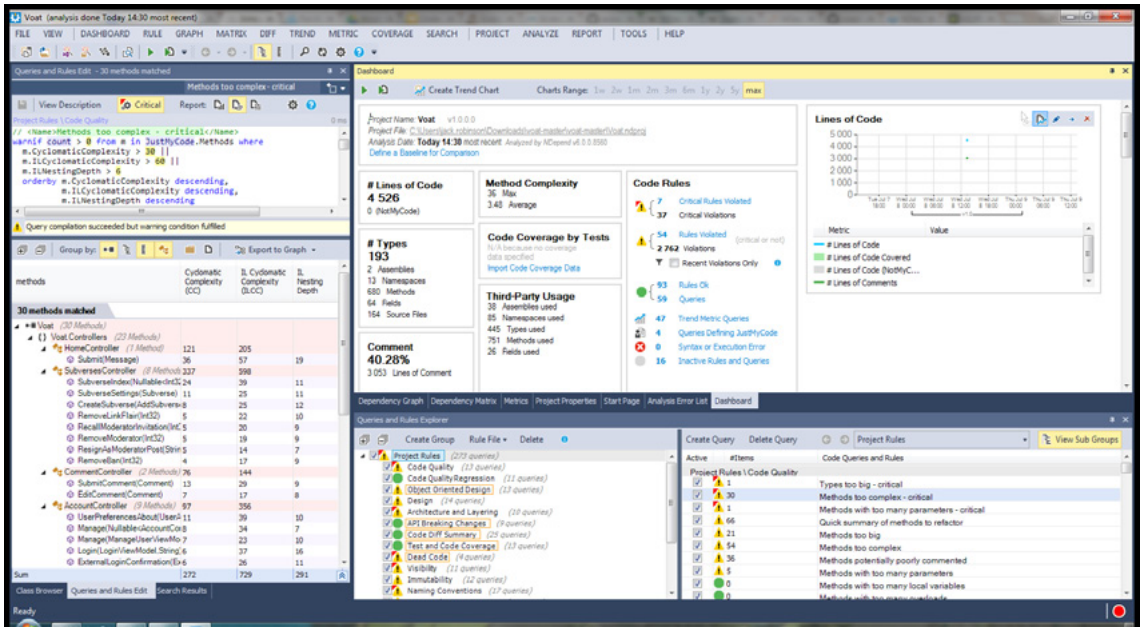


Clicking on "add assemblies from vs solution," then selecting your desired Solution, you are presented with the ability to pick and choose what you track. To the left is your assemblies from your Application, to the right, any assembly that NDepend It's probably important to note it works off of the PDB files, so make sure you build your solution before loading it into NDepend.

From there, you are away laughing. Looking at the top of that panel, you can see two play symbols. The one on the left runs an analysis on your project, while the one on the right generates an HTML based report on the rules you have defined, and presents the information in such a way that you can embed it as a TeamCity artifact.

The Dashboard

Once your analysis is completed, you're presented with this:



Dashboard

The dashboard provides a number of quick-access information to give you a snapshot on the results of your analysis. By default, NDepend comes with more than 150 rules that are run over your code, and each has a warning. If a rule that has been tagged with the **warnif** command, it fires a warning off to alert you. In a build process, you can actually fail a build if one of these critical rules are broken, although you may want to make edits to them if you're adding this to an existing code base.

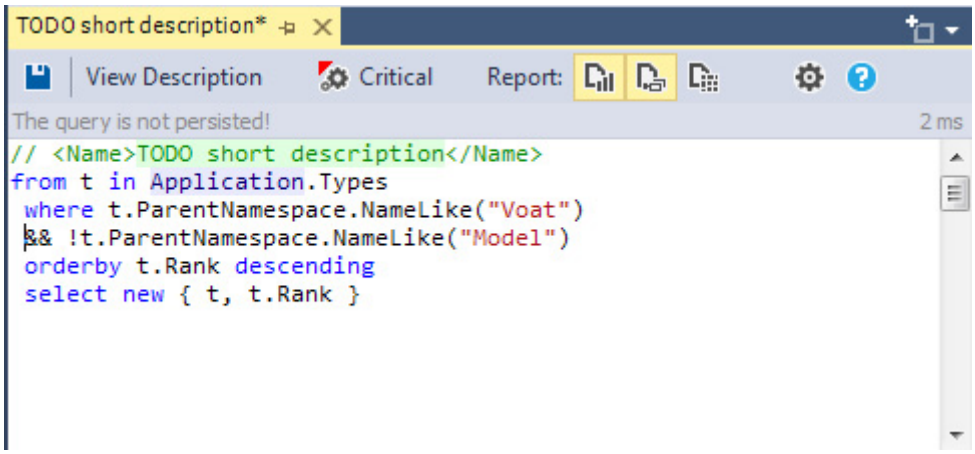
Queries and Rules Explorer/Edit:

Cutting right to the chase, this tool is totally amazing. As a student of Software Engineering, I am completely blown away at the capabilities of the CQLinq system. Be it naivety, or just the fact I'm sometimes easily excited by data, the ability to run queries over a code base, and get the results almost instantly can provide hours upon hours coming up with insights on your code.

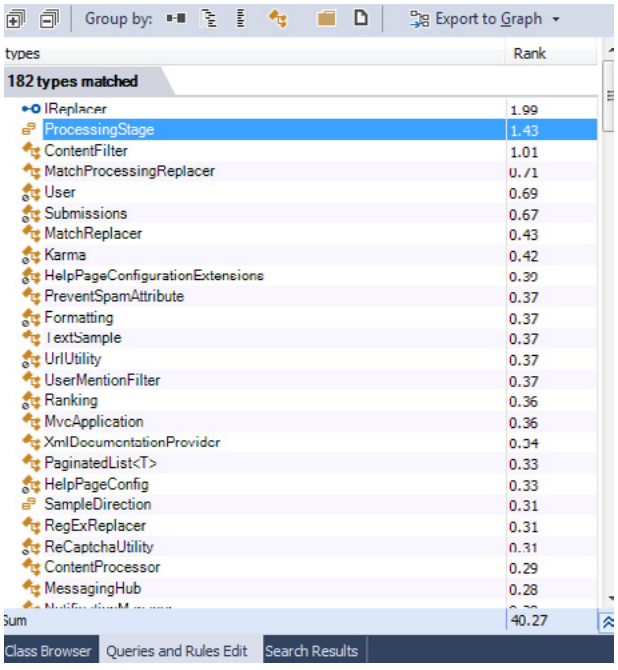
I learnt much of the query language from the default rules provided by NDepend, as I wasn't too familiar with LINQ and Lambdas (I come from a wholly Java education), however, the other developers in my team are able to come over and use their knowledge of Vanilla LINQ to run queries themselves.

In v5, I had a little bit of trouble with this tool, I think because the code base I work on is relatively huge (> 250,000 Lines), however, in v6, the rule editor is much more streamlined, and a joy to use.

As an example, I have this query I wrote up - It gets all the types in the Voat namespace, and orders them by their Google PageRank. This highlights proportions of the code that perhaps are being used by a large number of types, and therefore, should perhaps be looked at to avoid a “God Classes” scenario further down the track. I’ve removed the Voat.Model namespace, as generally Models are going to be used in a lot of places as they are more of an Object representation of the Database, rather than actually meaningful code:

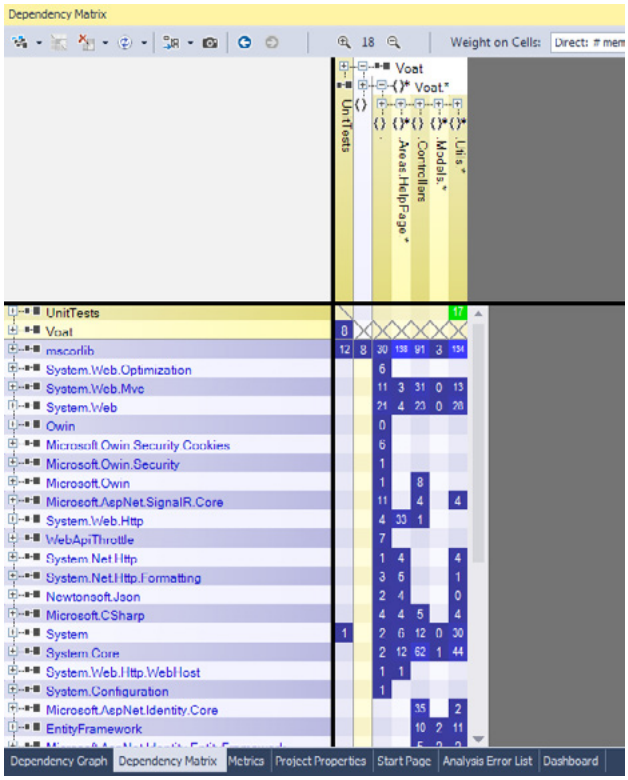


And instantly, as I type, I’m presented with the results. Most results will be instant, and NDepend has a timeout of 2 seconds for any query, so it’s also encouraging you to create well thought out, efficient cost queries:



The cooler thing about this is as soon as you identify culprit classes, you are able to select the result, and then it opens directly into Visual Studio.

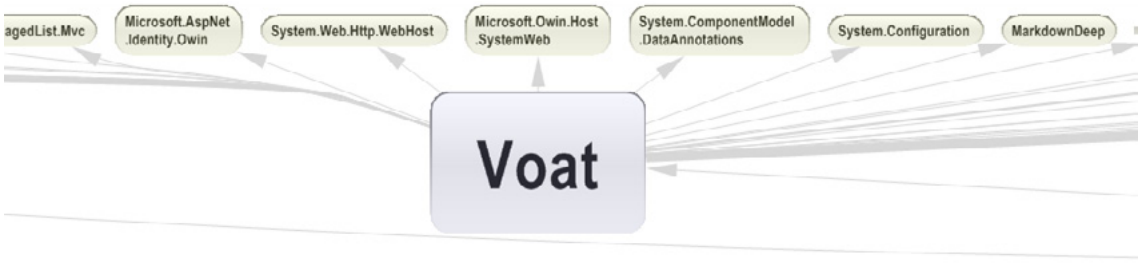
Dependency Matrix



NDepend provides a handy tool to identify the dependencies your code has, both on internal assemblies in your code, as well as dependencies on third party assemblies. It was pretty difficult to illustrate how handy this was with Voat, as the code is relatively small, however, the cool thing about this is you can drill down right to a method level by selecting the little pluses next to each one.

Clicking on any one of the squares, also drills down into a dependency graph, which leads nicely into:

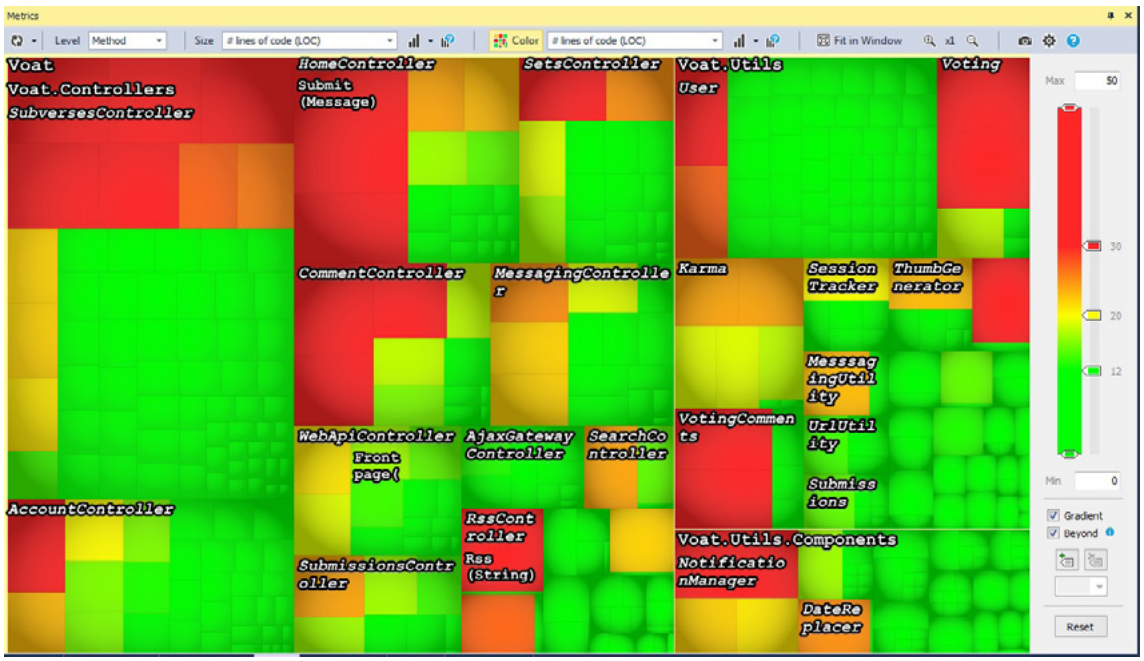
Dependency Graph



Basically the Matrix, in graph form. It is quite hard to fit it all on one page, however, this can be key in determining whether or not your code is highly coupled, or not. You can set what the size of each of the bubbles represent, and the thickness of the edges represent the number of dependencies it has.

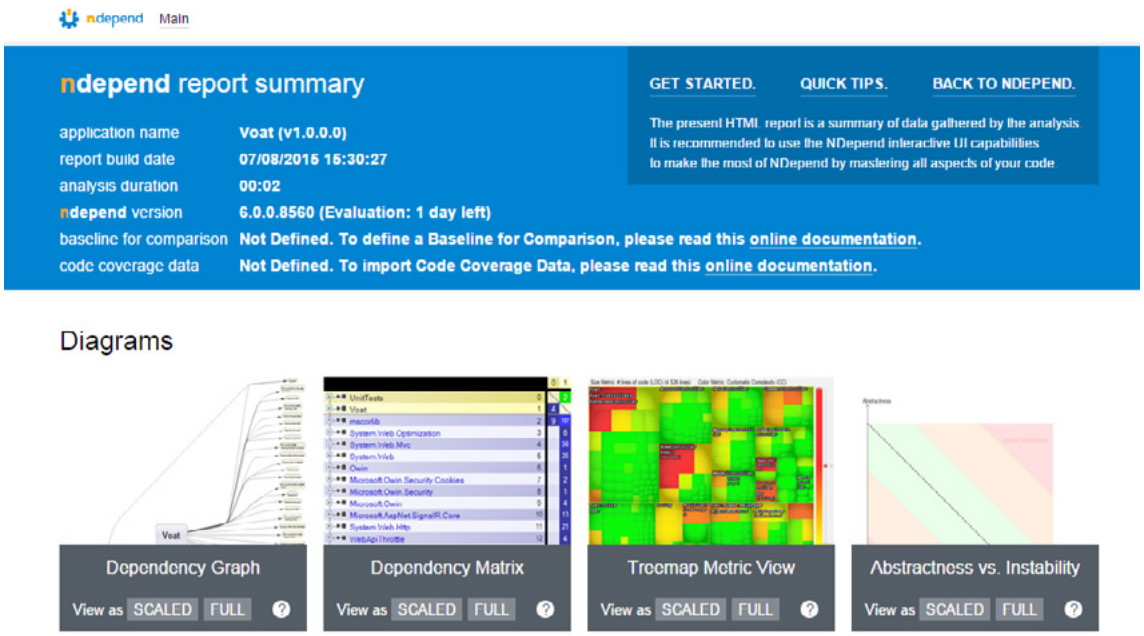
Metrics

Prior to v6, the metrics pane was a little drab, all grey scale, and somewhat confusing. However, now the pane is full of colour to identify the components in your system that may be causing trouble. In this following image, the more red a square is, the more lines of code it contains. You can set the “max” value to whatever you like, and in this case, it is any method over 50 lines.



Hovering over certain components gives you some quick access information about the method you're currently on. Once again, playing with this will increase your understanding, and therefore, the usefulness of the tool.

The Report



The report bundles all the sections above into a static representation. It extracts the rules you've saved in your NDepend Project, and turns them into HTML. This is useful if you wanted to publicly (within your dev team, that is) show how healthy the code is after each build/weekly.

Critical Rule warning: Methods too complex - critical

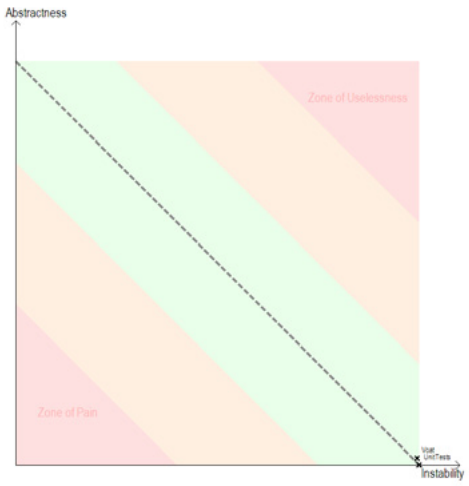
View Rule Description

30 methods matched

Display 25 records

methods	Cyclomatic Complexity (CC)	IL Cyclomatic Complexity (ILCC)	IL Nesting Depth	Full Name
Submit(Message)	36	57	19	Voat.Controllers.HomeController.Submit (Message)
SubverseIndex(Nullable<Int32>,String)	24	39	11	Voat.Controllers.SubversesController .SubverseIndex(Nullable<Int32>,String)
SendCommentNotification(Comment)	16	38	8	Voat.Utils.Components .NotificationManager .SendCommentNotification(Comment)
GenerateObject(Type,Dictionary<Type ,Object>)	15	17	12	Voat.Areas.HelpPage.ObjectGenerator .GenerateObject(Type,Dictionary<Type ,Object>)
SubmitComment(Comment)	13	29	9	Voat.Controllers.CommentController .SubmitComment(Comment)

You can also view how each of your assemblies fare in the Abstractness/ Instability graph that is also generated. Turns out in Voat, the code is relatively instable, therefore prone to change:



Phwoar, I think that's that for the briefest of flyovers on the NDepend GUI. I thoroughly recommend that should you get a trial, sit with another developer and figure out just how far NDepend can go in terms of rules you can write up, and the results you can get out of it - you can get a lot done in 14 Days.

NDepend API

The second component I'll look at is the API that NDepend provides, and as an extension, the NDepend.PowerTools code snippets that are provided to show you the capabilities of the API. Through my use of the API and Visual NDepend, I believe the GUI is built atop the API itself, and therefore, the API contains everything you need to implement the power of the NDepend Analysis into your .NET project.

Note: this API stuff is mostly based on my experience with v5, but I have double

checked some notes on the documentation to determine the differences with v6, but I apologise if I whine about features fixed in v6.

Installation

The **NDepend.API.dll** can be found in the **Lib** folder of wherever you unzipped your NDepend files. There are a couple things to note, however, when you choose to include the API into your project

- You need to set the dll to Copy False. Although you're only including the API dll, it needs to talk to the rest of the dll's in the Lib folder.
- In order for NDepend to talk to every other lib file, you need to edit the Assembly Resolver with the code found in the NDepend.PowerTools namespace (**AssemblyResolver.cs**)
- You must have all the files that you downloaded be present, that is, you must keep the VisualNDepend.exe and the Visual Studio plugin present when using the API in the same file structure you downloaded it in. I couldn't see a reason for this.
- If you're using a WebAPI/MVC application, you may have to add in a probing statement like in this Stack Overflow answer

One of my dissatisfactions with NDepend is the API installation. It seems very voodoo magic-y to get it to behave, and the documentation is a little vague on a lot of how the install works. Even now, I don't exactly know what I did, but it works, and it is definitely something that would be neat to see fixed, or become more streamlined in a future version.

Usage

I use the NDepend API extensively in a work project I have been working on, however, below is a Method from an API test I wrote as I was learning the in's and outs'

```
private static void Run()
{
    Console.WriteLine("Deginning NDepend Analysis [ ANALYSIS ONLY ]");
    var sw = new Stopwatch();
    sw.Start();
    var projectManager = new NDependServicesProvider().ProjectManager;
    var project = projectManager.LoadProject(ProjectPath.ToAbsoluteFilePath());
    project.Coverage.AddCoverageFile(CoverageResultPath.ToAbsoluteFilePath());
    project.RunAnalysis(log => Console.Write(string.Empty),
        output -> Console.Write("*"));
    Console.WriteLine("Analysis complete! Time Taken: " + sw.Elapsed);
    sw.Stop();
    if(Environment.UserInteractive) Console.ReadKey();
}
```

All this method does is receive the project from a designated ProjectPath, and attach a Test Coverage output from dotCover (NDepend supports dotCover test coverage files to contribute to the metrics you gather from your code). and then prints out the analysis as it happens, before printing the time taken, then closing. Do note, however, you can ***only run the analysis through the API on a Build Machine License.***

This is a fraction of what the API can achieve, as a majority of what is achievable in the VisualNDepend executable is also usable through using the API - refer to the NDepend.PowerTools project for more examples.

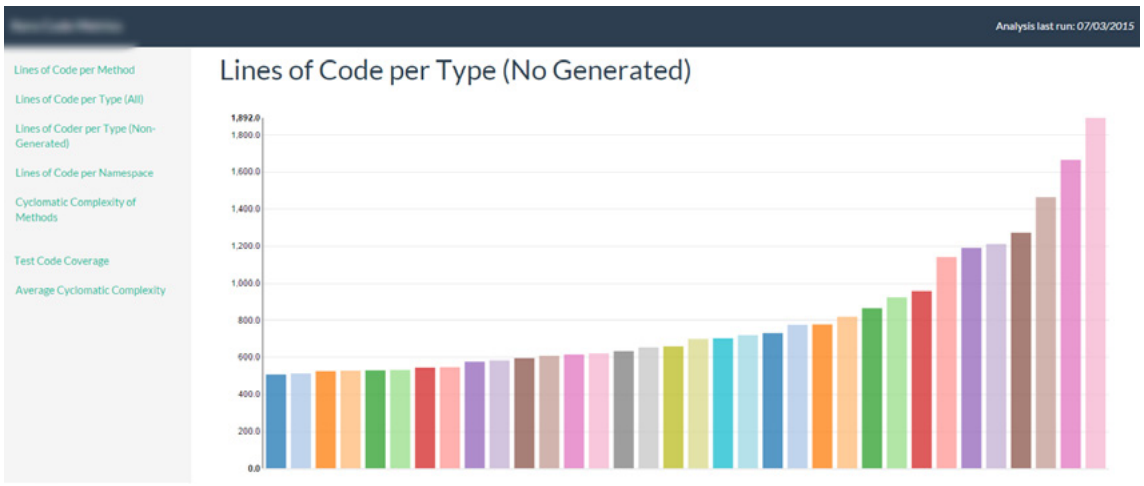
A pitfall to watch out for is the CQLinq reserved namespace, and the fact you cannot write raw CQLinq into your project, or at least, in the method I attempted to. To get around this, I test out queries in an NDepend Project file, save the ones I like, then when I run the queries through the API, they are included in the order in which they appear in the GUI. It's a simple workaround, and I'm not entirely sure that is how it is meant to be run, however, once again, the lack of extensive documentation is another con to the otherwise powerful API.

Once again, I've barely scratched the surface with the API, and definitely warrants a view. The trial version unlocks all the features of NDepend, so if you're curious, have a play around, and you can soon discover how it works for you.

NDepend in our Developer Ecosystem

My summer, and now, winter project was incorporating NDepend metrics into a dashboard for developers to see. Originally, we wanted to add it into its own TeamCity build process, however, these days, the architecture looks a little different these days.

What we do is have a scheduled task run on a Friday night pull all the changes from the previous week, before building, running dotCover, then compiling it all together through NDepend. The most recent results are stored in a database for ease of access (the code base is relatively huge and ends up ~150MB of memory each analysis loaded if we did it via NDepend). A WebAPI project loads up that database, and displays the information in handy graphs.



Hovering over a bar will show the related class/method, and its value, be it Lines of Code or Cyclomatic Complexity. We also store averages as trends over time to analyse how our code base changes over time.

Issues

It wouldn't be a project without its fair share of issues.

- This is the second version of the code. I scrapped the original as it was prone to Memory Leaks and lack of Testing (turns out dotMemory crashes on dumping 7GB).

- The initial design didn't handle new rules very well. I'm so used to academic coding, that enterprise software is a bit odd, but certain design decisions make sense to me now, and I continually implement them into my work and my university work.

- I didn't, and still don't, fully grasp NDepend. I'm still a student who hasn't finished their degree, and the full limits of my static analysis knowledge amount to perhaps 5 hours of university time, and whatever I have learnt along the way with NDepend. The current project is definitely an MVP, and I hope to extend upon it in the very near future.

Conclusion

I'll summarise with a **tl;dr**
I'm young an I'm naive: I don't get paid to write code.
Code Metrics can be important: Our mental models suck (sometimes), but we should really understand what our code does

NDepend is Pretty Good at it: It's a steep learning curve, but the possibilities are endless
You can use NDepend anywhere: Just can't use it to microwave your dinner (unless you have massive projects #laptopdev)
NDepend is a Swiss Army Knife of capabilities: If you think you know NDepend, you'll be surprised.

Jack Robinson

Jack is a twenty-something student in his final year of a degree in Software Engineering at Victoria University of Wellington. Currently an Intern Developer at Xero, he enjoys writing clean code, playing a board game or two with his friends, or just sitting down and watching a good film. You can read about not just his musings on computer science, but also reviews on films and more at his website jackrobinson.co.nz

Iterate towards Better Code Reviews

Prasad Narravula

When code review meetings get derailed, they become painful and unproductive. We will cover in this post how to make them focused and effective sessions by eliminating DONTs and doing more DOs. Though the post focuses on the .NET environment, the principles apply to any object-oriented environment.

> *A Process Cannot be a Substitute for a Skill, but can Enable continuous improvement*

Software design is a team effort. As the code is developed, a developer makes many design decisions on a daily basis such as adding methods, creating associations between classes, use of switch statements and so on. Whether these are good ones is a different problem. When a team comes to embrace this reality, it finds a need for the different type of technical leaders. For example, Architectus Oryzus is such a leader that enables team design activities while acting as a guide when needed. Martin Fowler writes that a guide is a more experienced and skillful team member, who teaches other team members to fend better for themselves yet is always there for the tricky stuff.

Depending on skill levels and understanding of the business domain, developers make bad decisions. You will miss teachable moments if you try to avoid them by taking control over the design. The reviews present these teachable moments. Pair Programming is another such practice that presents the opportunities for mentoring. Both the reviews and the pairing improve collective code ownership.

Code Consistency, taken care by individual programmers, helps the team to focus on the design and the functionality. Automate the consistency related guidelines as much as possible.

You will see how code reviews become focused with the little effort. You will find practical guidance on continuous improvement as essential skills need learning

and practice.

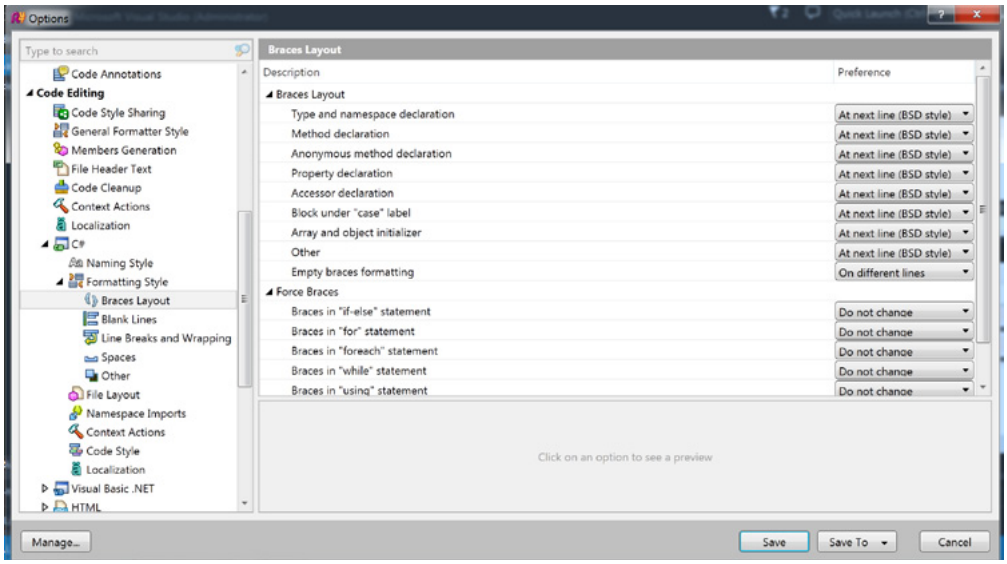
Take Care of Typos

Tools such as Respeller, a R# plugin, can help in finding the typos as they happen. It checks for misspelled words in comments, strings, and identifiers-classes, methods, variables and so on.

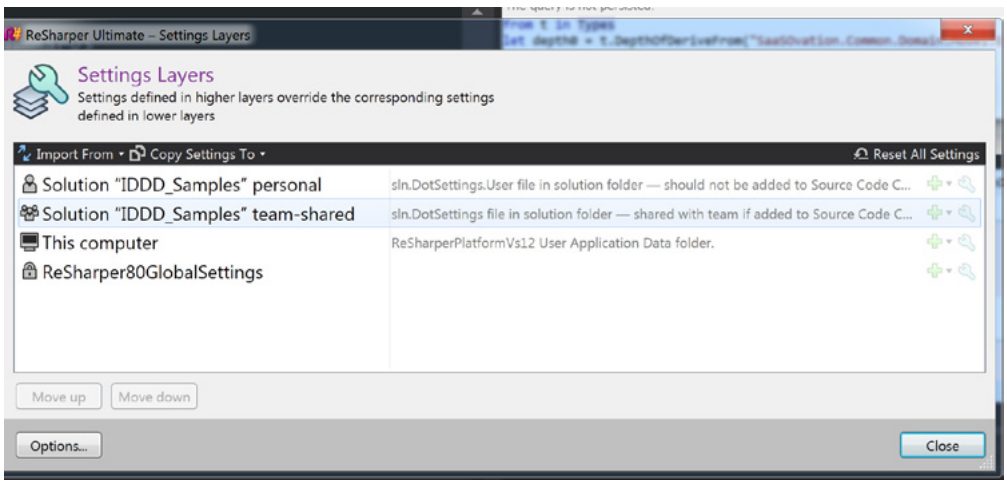
If needed, you can ask for a quick review from one or two people. Early reviews of public contracts prevent misspellings from reaching the world.

Fix Code Formatting Issues Early On

Formatting issues such as indentation, blank lines, and spaces irritate the team. R# formatting feature works well in cleaning up the formatting.



Team level R# settings help in maintaining consistency.

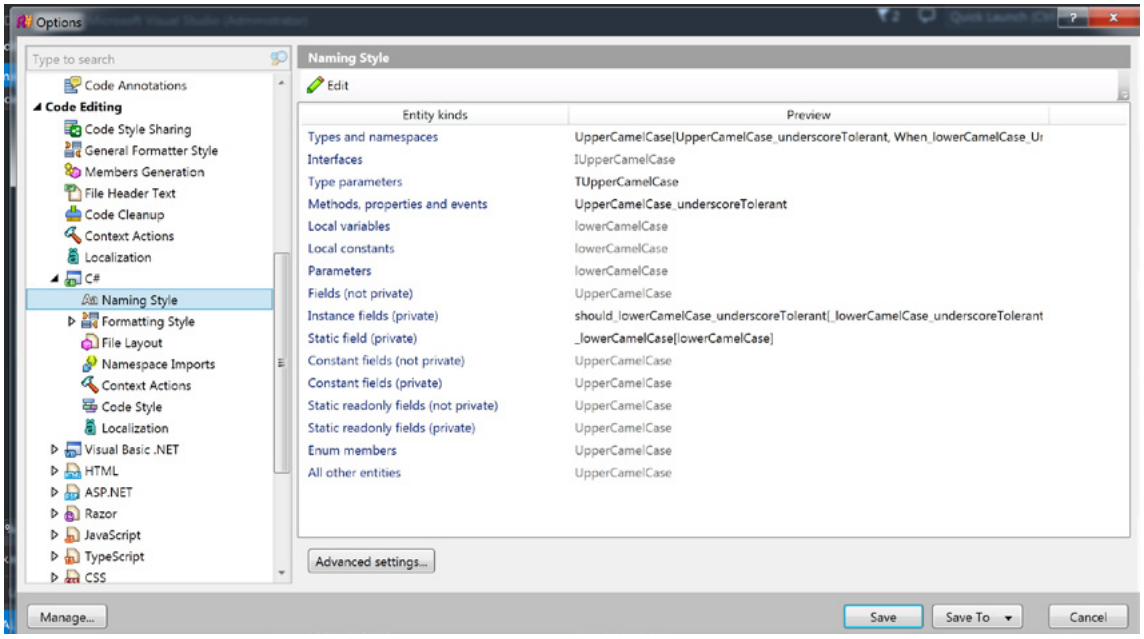


Document Naming Conventions and Coding Styles

It is important to compile the coding guidelines at the beginning of the project. Framework defined guidelines are a good starting point. Even if an enterprise-wide document is already available, the team still should go through it. The collaborative effort promotes collective code ownership. Code consistency depends on the buy-in from the entire team; otherwise it becomes the focus of the code reviews making them inefficient.

Achieve consistency

There are tools you could use to create shorter feedback loops. R#, with extensive rule sets and auto correcting capabilities, tops the list.



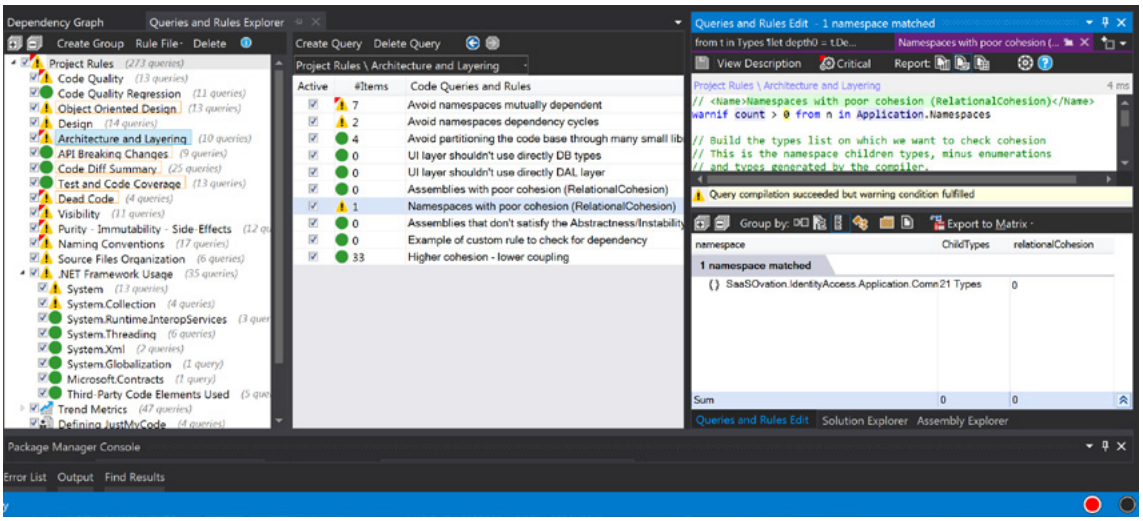
You can customize default naming styles and share them across the team using the custom profiles. Stylecop users can benefit from R# auto fixes using the Resharper.Stylecop plugin.

Document General Coding Guidelines

Teams compile general coding guidelines, DOs and DONTs, such as preferring exceptions over return codes, implementing IDisposable if a class contains disposable fields. Generally, it is a best practice to make expectations explicit to avoid rework. You can refer heuristics in Bob Martin's Clean Code book. People at csharpguidelines.com compiled a set of coding guidelines including some best practices, and the R# settings targeting these coding styles.

Achieve consistency

As with the styles/naming conventions, the team should strive to write code following the heuristics. Teams waste enormous amount of time in enforcing the guidelines through the reviews. A tool is helpful if it reminds you about these guidelines as you write the code. NDepend comes with a set of code rules. You can write new ones or customize them. As everyone in the team may not know all the guidelines, you could start with what the team can follow and then you could drive continuous improvement over few iterations. Iteration/Sprint goals are best suited to learn and practice these rules before enabling them. It helps to start with NDepend's green status circle (in the IDE status bar). Over few iterations, you can add rules progressively as the team gets more comfortable.



This immediate feedback reduces the knowledge gap and reminds people when they take any shortcuts. If the developers use everything they know, that itself is a big improvement.

Code Commenting

It is hard to maintain the quality of the comments if you enforce it through the review sessions. Ad-hoc comments lack consistency whereas a Samaritan effort is suboptimal. It is a challenging task to keep the comments alive throughout the project cycle.

The team should name the things to improve the code readability. You should add comments where necessary explaining why. Redundant, superfluous comments are such a waste of time. A Team that focuses on the readability of the tests and the code gets better ROI. One side-benefit of TDD is that the tests become a reliable documentation. Intention revealing tests along with good

documentation help your code users outside your team.

If the code is the design, then tests are the best documentation you could give. GhostDoc plugin automates the routine tasks of inline XML commenting.
It is necessary to make the strategic decision early on, and the team should strive to achieve consistency throughout the project. There is no free lunch when it comes to commenting. The team should keep this effort in mind during estimating sessions. Consistency is the key.

Avoid Premature Optimizations

If there is one thing you should keep out of the reviews, that should be premature optimizations.

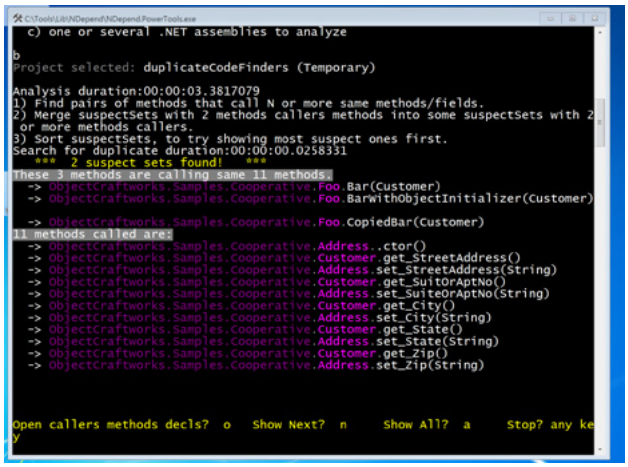
> *Programmers waste enormous amounts of time thinking about, or worrying about, the speed of non-critical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*
>
> - *Donald Knuth*

Don't let the team fall into guesswork. Get the story done before carrying out optimization efforts. Most often, the optimizations efforts are started by defects. But SLAs should drive the performance efforts. SLAs are constraints on the stories. Find them out early in the project and make them public and visible in the team area. Use performance stats and profiling to find the critical 3% effort.

To avoid uniformly slow code, the team can compile the platform-specific performance idioms and add them to the guidelines document. Using StringBuilder for complex string operations is one such item, for example. Iterative development makes it easier to spot any such code behavior before going too far.

Now comes the meaty stuff that requires effort to learn and practice to gain expertise.

Take Care of Code Duplicates Early On

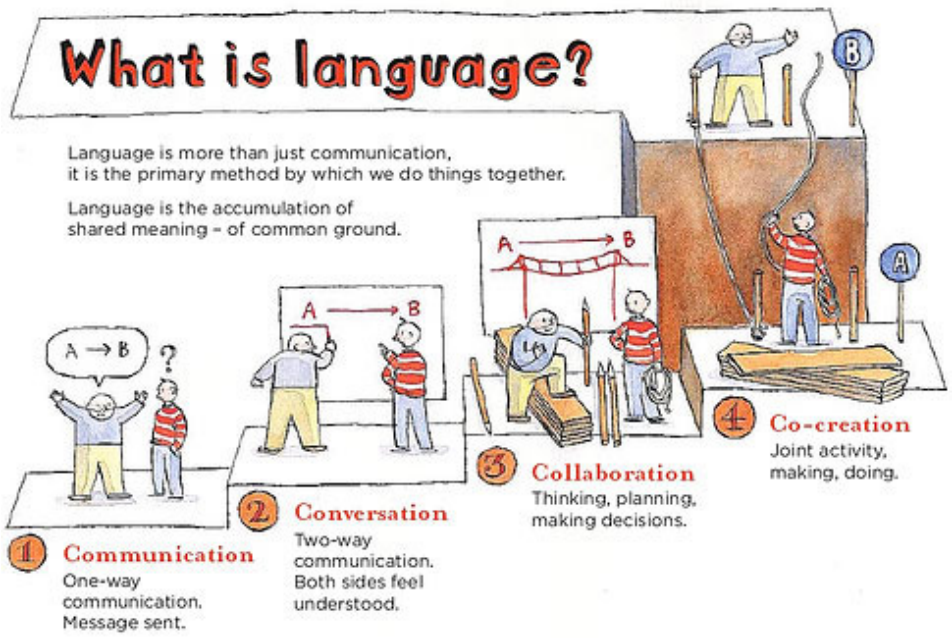


Code reviews are too late to find the duplicates.

Develop Shared Vocabulary

Arbitrary language creates conflicts making the meetings painful to attend next time. The team should guard against language ambiguities such as semantic diffusion, and flaccid words. It should put effort to come to common understanding of the key vocabulary. For example, the word **"Refactoring"** is being used to describe various things, weakening its intended use. Martin Fowler calls this Refactoring Malapropism.

The team that speaks shared vocabulary is set for the success. The power of shared vocabulary can do wonders in other areas too. Dave Gray created this wonderful graphic.



For better code reviews, the team needs to develop the vocabulary to talk about the issues and their available remedies. Luckily, you need not reinvent the language, as, since XP, there are several sources. Code smells, design smells and the refactorings that fix these issues are a good starting point. SOLID principles can help in refactoring and designing activities. Once you gain the expertise in the design patterns usage, you will have a rich vocabulary.

The team should gain fluency. A culture that supports the expressions in such rich vocabulary is essential to agile maturity. The culture should be mindful of the mixed skill level. The leadership should enable the environment for experiential learning. With such common understanding, the team organically opts for the pair programming.

Here are few examples of expressions:

When you see minor changes to several classes, you could say

> *Looking at the changes you just made, we have this behavior all over the place. It is a shotgun surgery. Let's inline this class.*

When a class is too much dependent on other classes:

> *With this change, class Foo has just become a feature envy. Let's extract this part and then move it to that class.*

When someone overzealously applied OCP principle:

> *This instantiation does not need a separate factory now. We can take the first bullet, and wait for the actual need. One less indirection is always good.*

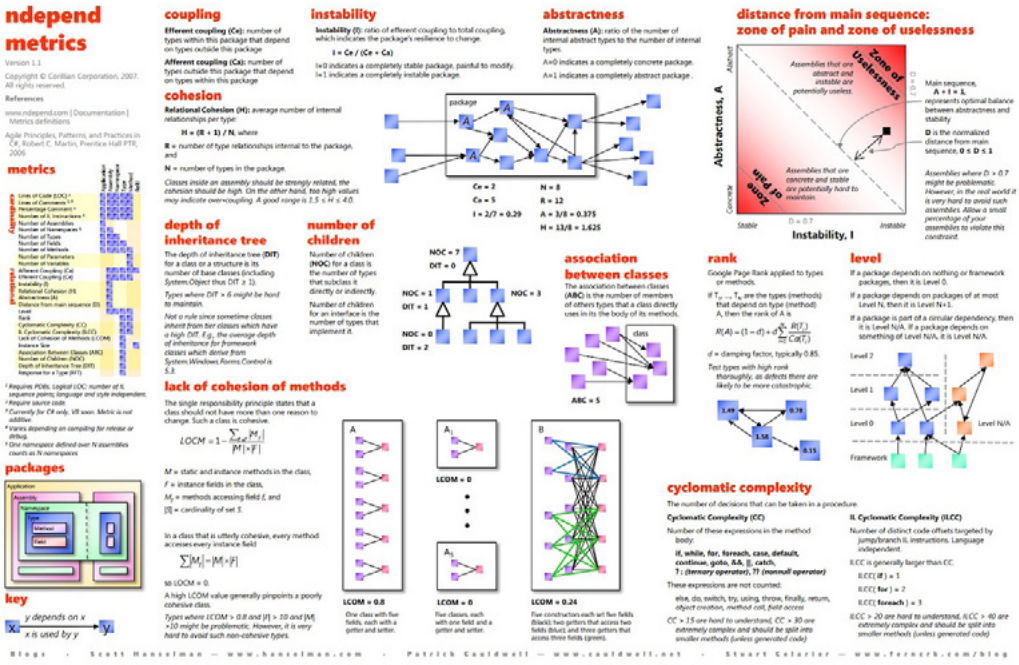
As you see this is much more effective than what you hear typically - *I like this*, or *I don't like that*. Once a team gets comfortable with the Refactoring Catalog and the awareness of code smells, its code reviews become much shorter and effective.

Manage Code with Metrics

As information measurements, code metrics give us a useful view of the codebase. Code Metrics based vocabulary makes the code quality a team activity.

NDepend is a handy tool to calculate metrics. With this tool, you can create shorter feedback loops by turning expectations into code rules. Its status circle, which is in the IDE status bar, will turn red as soon as a violation is detected. You can also turn these code rules into critical rules to fail the build whenever the metrics cross the thresholds. For example, you can write a code rule to fail the build when LOC of any method crosses a threshold value let's say seven.

Here is a metrics placemat for your reference.



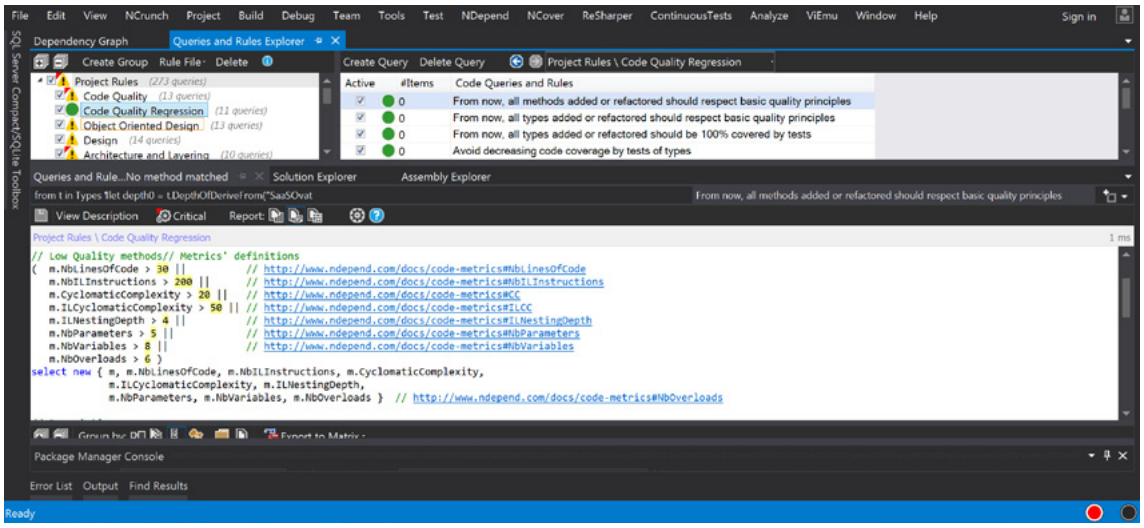
Don't Let the Bad Code Pile Up

The refactoring in a brownfield project can overwhelm, no matter what you do, there will be code smells. If you allow them to happen, the code further deteriorates. As Broken Window Theory states, bad design piles up. Any time we take a shortcut, we lose an opportunity to hone our skills.

One approach that works well is **"from now onwards"**. The idea is simple; you would create a baseline to track the quality of the code for all the present and future code changes to make sure you are not making it worse. Continuous improvement becomes fun and brings motivation from the job satisfaction. It creates a *positive reinforcing loop*.

You can use NDepend to create the baseline. With metrics, code rules make *from now on goals* measurable. For example, the goal such as the distance from the

main sequence, measuring stability and abstractness, of module X, should not exceed standard deviation of 1, is easy to measure and track.



Wrap-up

Items we covered that make code reviews focused and effective:

- Embrace the design as a team activity and create the culture of continuous learning.
- Document coding guidelines, naming styles, formatting, and performance idioms.
- Don’t wait for the code reviews; Take care of typos, formatting, naming conventions, general coding guidelines, and code duplicates, as the code is being developed.
- Use tools to achieve consistency and to let the machines do the dirty work.
- Avoid premature optimizations. SLAs, profiling, and performance tests should drive optimization efforts. Make SLAs public and add them as constraints on the stories.
- Set expectations early on the project and aim for shorter feedback loops. For certain activities, the code reviews may be too late.
- Develop a shared vocabulary for the design activities. Code smells, OO metrics, the refactoring catalog, and the design patterns are the great language sources.

Prasad Narravula

Prasad Narravula is a programmer, architect, consultant, and problem-solving leader. He helps teams with agile development essentials- feedback loops to fail fast, enabling (engineering) practices, iterative and incremental design, starting at the right place, discovery, and learning. When time permits, he writes at ObjectCraftworks.com.

Relax, Everyone's Code Rots

Erik Dietrich

I earn my living, or part of it, anyway, doing something very awkward. I get called in to assess and analyze codebases for health and maintainability. As you can no doubt imagine, this tends not to make me particularly popular with the folks who have constructed and who maintain this code. "Who is this guy, and what does he know, anyway?" is a question that they ask, particularly when confronted with the parts of the assessment that paint the code in a less than flattering light. And, frankly, they're right to ask it.

But in reality, it's not so much about who I am and what I know as it is about properties of code bases. Are code elements, like types and methods, larger and more complex than those of the average code base? Is there a high degree of coupling and a low degree of cohesion? Are the portions of the code with the highest fan-in exercised by automated tests or are they extremely risky to change? Are there volatile parts of the code base that are touched with every commit? And, for all of these considerations and more, are they trending better or worse?

It's this last question that is, perhaps, most important. And it helps me answer the question, "who are you and what do you know?" I'm a guy who has run these types of analyses on a lot of codebases and I can see how yours stacks up and where it's going. And where it's going isn't awesome — it's rotting. But I'll come back to that point later.

Communication Complexity Grows Non-Linearly

Imagine that you're working alone on a project of some sort. You're certainly going to be bounded by your own productivity, but the communication overhead to whatever you're doing is essentially nil (unless you're counting leaving yourself notes and reminders, which I won't). Now, let's say it becomes necessary to substantially improve the throughput on this project, so an additional person is added to the mix. Communication is now more of a consideration, but it's also quite simple. There's one channel for it and that's it.

But what happens as the team grows? Once you add a third person, the number of lines of communication goes from 1 to 3: AB, BC, AC. If you add a fourth person, you get another non-linear increase in the number of lines of communication: AB, AC, AD, BC, BD, CD, for a total of 6. If you go to 5, 6, and 7 people, the lines of communication increase to 10, 15, and 21, respectively. Mathematically, this growth makes sense. Each new person coming in adds one line of communication for each already existing person, which is why the lines of communication grow by 2, 3, 4, 5, etc. If you prefer a more mathematically rigorous way to understand this, it's the idea in discrete mathematics known as combinations.

As the team grows, one person at a time, the amount of communication overhead beings to explode. By the time you have 20 people on the team, there are 190 one on one interactions (to say nothing of situations that call for multiple people). This means that, from a practical perspective, there is a limit on team size beyond which there are diminishing and, eventually, negative returns. The team will eventually do nothing but manage all of these communication channels.

What does this have to do with code? Well, a code base grows in about the same way. It's just easier for people, particularly non-technical folks, like managers, to wrap their heads around team lines of communication.

Code Breaks Down the Way Disorganized Collaboration Breaks Down

In modern languages, code in a codebase is assembled into some form of logical units or modules. These might be functions, classes, whatever. When there are few of them, life is pretty good and the code is easy to reason about. As the number of these things grows, so too does the complexity, and not linearly.

Without any kind of deliberate intervention, codebases suffer the same fate as teams with 20 or 30 or 40 human beings on them all trying to collaborate. Eventually they reach a point where adding to them introduces more problems than it fixes.

How do you prevent this? Well, it's not easy, and it requires intentionality. This is where I'll return to the theme of your code rotting. Yes, your code is rotting, but so is almost everyone else's as well. It's not an unusual circumstance, and it doesn't mean that you've done anything horribly wrong. It just means that you haven't yet figured out how to prevent it from rotting.

So, what does it take, in the end? Well, it's simple... to describe. Put on your managerial hat and ask yourself what would do with a team of 20 or 30 people that was slowed to a crawl by communication overhead. I bet you'd break them into sub-teams with much less communication overhead and have limited, strategic communication between those teams. Maybe this would be reminiscent of how companies organize themselves?

To do this with a codebase requires the same approach, in concept. You minimize the size and complexity of the code components, the way you would with teams. You eliminate unnecessary dependencies in favor of cohesive units. You make sure you have solid backup plans around any high-risk communication bottlenecks and you try to eliminate those whenever possible. And you evaluate the whole thing on a consistent basis to ensure that you're getting better (or at least not getting worse).

Take-Aways

So in the end, there are two lessons to take away when it comes to your code base. The first is that having a codebase that is rotting with tech debt, while problematic, is not unusual, nor is it a personal failing of yours or your teams. The second is that you need to understand how to manage complexity within your code. The first part is easy. The second part is why code assessments, analysis tools, and coursework on clean code exists in the first place. Because writing clean code takes a lot of work.

Erik Dietrich

Erik Dietrich, founder of DaedTech LLC, is a programmer, architect, development coach, writer, Pluralsight author, and technologist. You can read his writing and find out more about him at http://www.daedtech.com/ and you can follow him on Twitter @daedtech.

Integrating NDepend with TeamCity 9

Tomasz Jaskula

NDepend 6 comes along with really exciting new features, for example, Visual Studio 2015 integration, analysis enhancements, asynchronous support, etc. The feature I'm interested in is the TeamCity integration, which should be very easy to do according to NDepend's team.

If you use C# as your main programming language and you have already worked on a quite big project, you know how painful is to enforce coding rules and to query about dependencies if you don't have right tool. NDepend is a static analysis tool that helps you dig into your code in a very easy and efficient way to carry-out code rule checks, code quality checks, comparing two versions of a code base, browsing for differences and much more. This is something that you should have in your tool belt. If you're working on a medium to large project, you are supposed to have a build server and a sort of Continuous Integration process. I'm using TeamCity 9 for my projects. It would be nice to take advantage of NDepend's static code base analysis inside your CI build. It was always possible to integrate NDepend into your build process in TeamCity but it was quite tedious. NDepend 6 comes with this new shiny TeamCity plugin (for TeamCity 8 and 9) and I decided to give it a shot. Let's try it out and check if this is easier to do than before.

NDepend TeamCity Plugin Installation

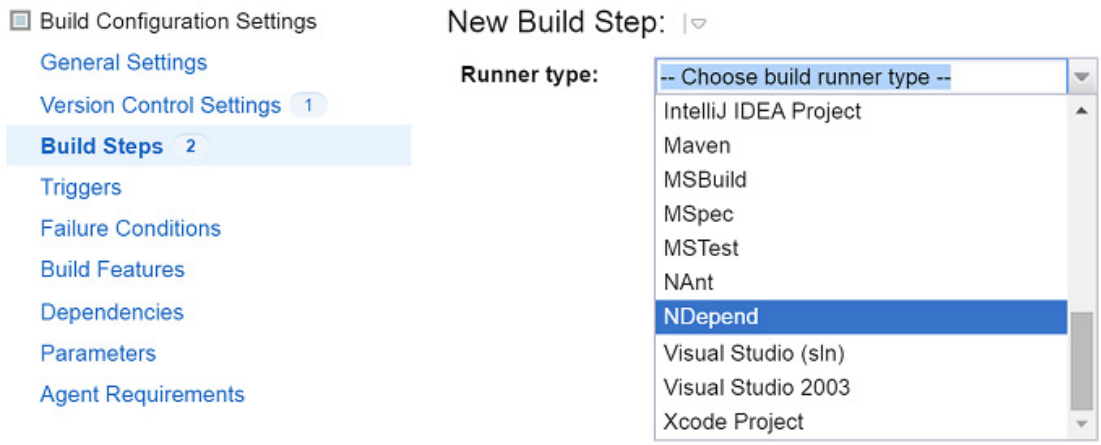
I have followed the installation steps described at NDepend documentation so there is no need to rewrite it here. However, I'll highlight some of the differences in the process that I found on my specific environment compared to the official documentation.

First of all, finding the TeamCity plugin folder was not straightforward as my environment variable **TEAMCITY_DATA_PATH** was not defined.

After checking the official TeamCity documentation we can read the following:

> ***Before TeamCity 9.1**, the TeamCity Windows installer configured the TeamCity data directory during installation by setting the **TEAMCITY_DATA_PATH** environment variable. The default path suggested for the directory is: %ALLUSERSPROFILE%\JetBrains\TeamCity.*
>
> ***Since TeamCity 9.1**, installer does not ask for the TeamCity data directory and it can be configured on the first TeamCity start.*

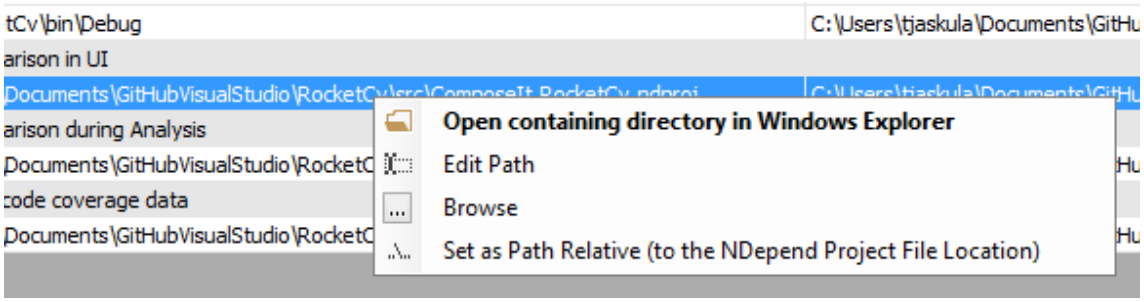
Obviously, I have installed the TeamCity server v9.1 and I don't have the **TEAMCITY_DATA_PATH** set. The best way to check where your data directory is located is to check it directly in the TeamCity web interface. Browse to **Administration -> Global Settings** and check the value of **Data directory:** Mine is located in *C:\TeamCity.BuildServer*. You can now copy/paste the NDepend plugin and follow the instructions in the NDepend documentation. In the end, you should be able to display NDepend step in the TeamCity server.



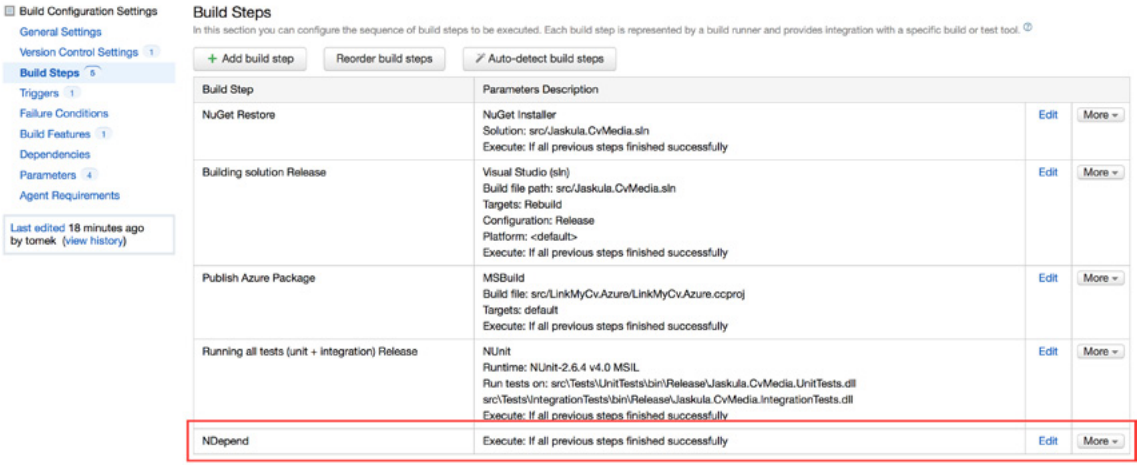
NDepend TeamCity Plugin Configuration

If you follow the NDepend documentation, setting up the NDepend build step is rather straightforward. Don't forget to change paths inside your NDepend project from absolute to relative. Remember that your build server may have different paths than those defined on your development machine. That's why relative paths are very handy. You can achieve it very easily, by going to your NDepend

project properties, clicking on the **Paths Referenced** tab and then right clicking on the listed paths and selecting **Set as Path Relative** option from the menu.



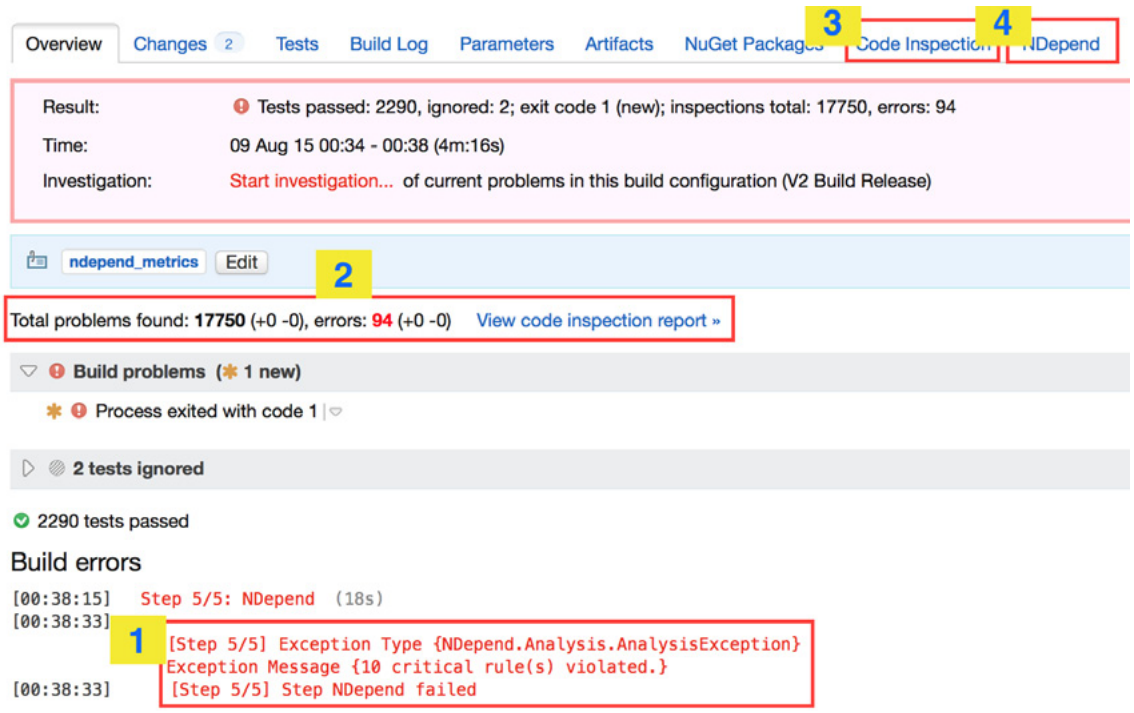
Once the change is done and committed, I've added the NDepend build step to my build process



Running the First Build

We're finally ready to run our first build with NDepend. The whole NDepend plugin configuration step inside TeamCity took me something like 10 minutes! It's really a piece of cake!

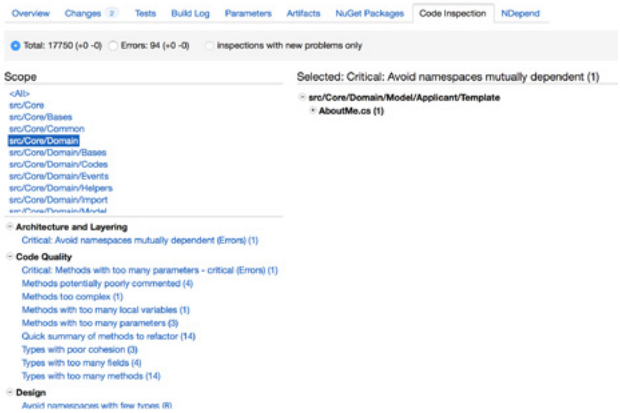
Let's check the output of the first run:



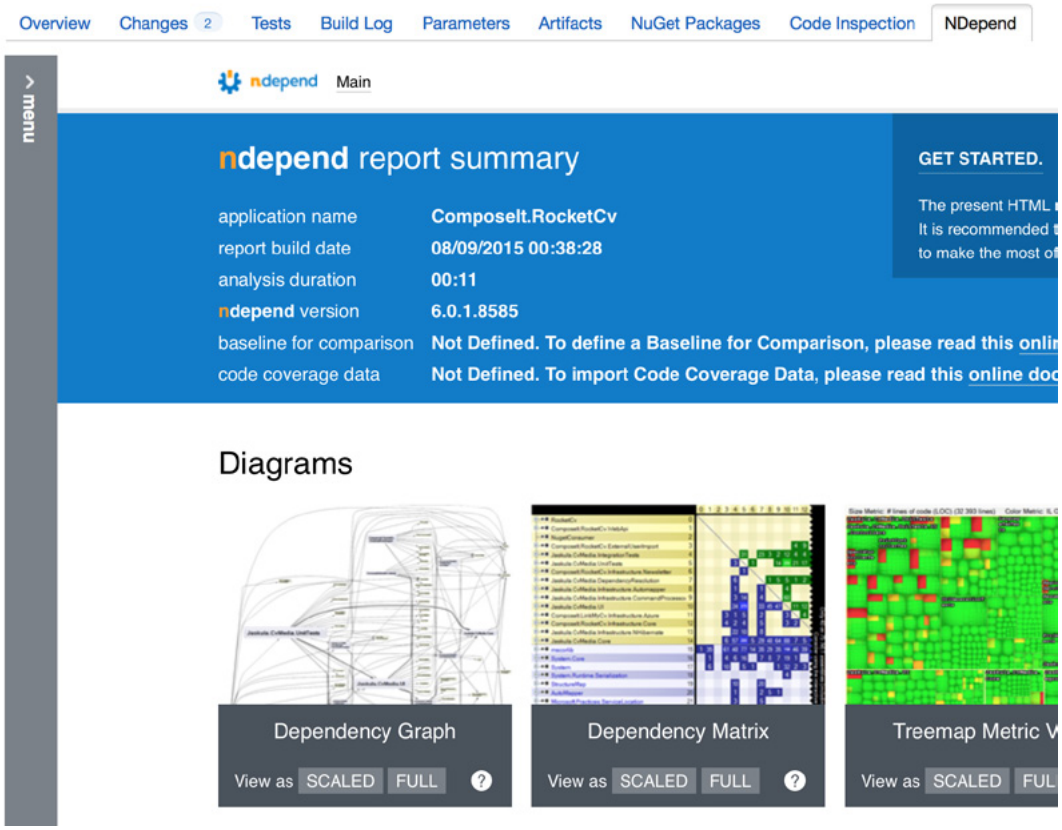
1 The build failed because of NDepend's exception. When the static analysis detects that critical rules are violated, the build fails, NDepend plugin maps rule violations to the TeamCity code inspections and errors. Rule violations are mapped as TeamCity code inspections and critical rule violations are mapped as errors.

2 Total number of problems detected during the NDepend analysis. These are visible in the TeamCity code inspections tab (step number 3).

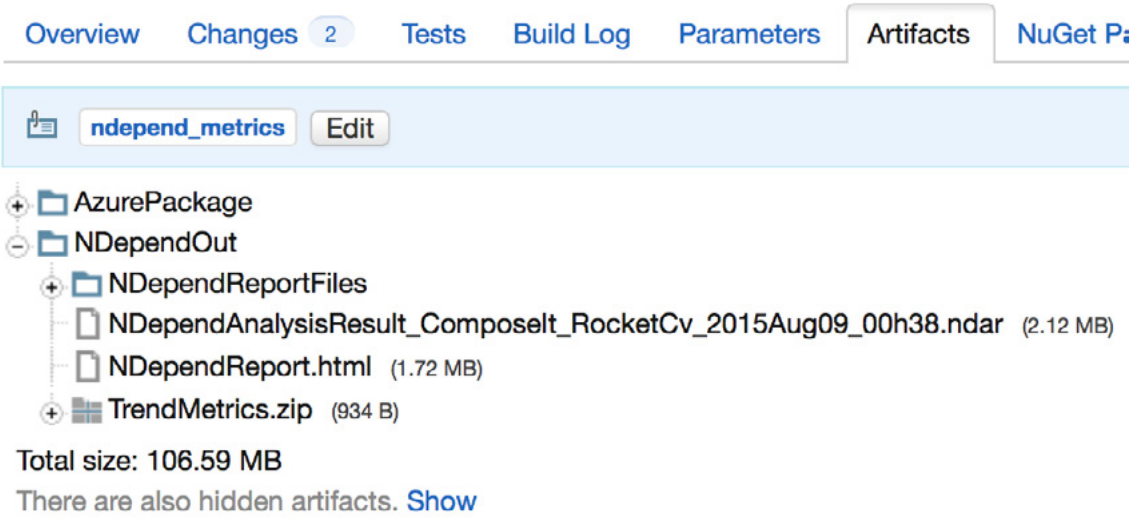
3 The list of code inspections as a result of NDepend's analysis. You can browse through all the problems detected by NDepend. Clicking on the details of the problem it brings you directly to an opened Visual Studio solution.



4 NDepend report is accessible in it's own tab! Awesome!



As a bonus, NDepend's output is available as a TeamCity artifact so you can download it if needed.

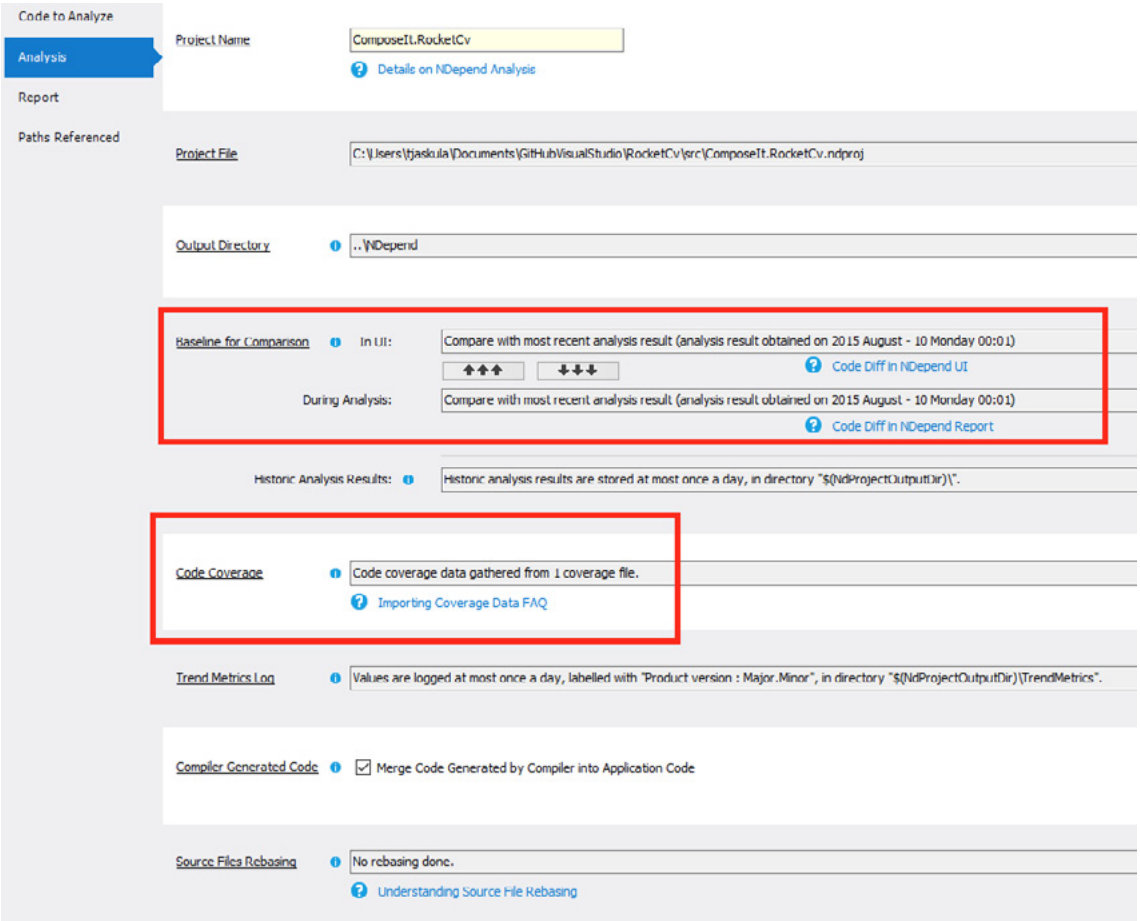


In the Build Configuration settings, you can also set build failure upon rules and critical rules violations values. This may be useful if you want the build fail because of the evolution of the NDepend metric change.

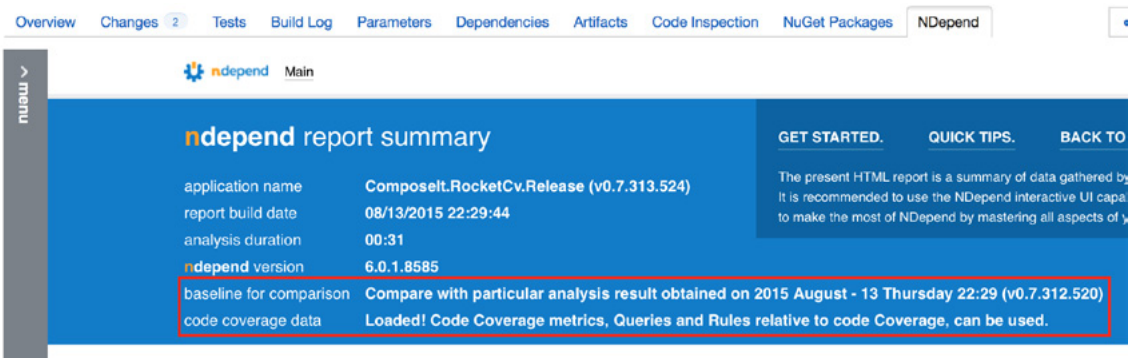
More NDepend Features for Free

Of course you don't need to work that hard to get other useful NDepend features for free. I'm talking about comparing differences between two different NDepend analyses and code coverages. For code coverage, all you need to do is to configure it in the NDepend project properties.

For code comparison, you have to choose a value in the NDepend build step on TeamCity for "baseline for comparison". I've selected "The last successful build" so the effect will be the comparison between the current build and the last successful one. You have to also set the right value in the NDepend project properties, as shown on the next page.



Once that's done, you run the build and check the build output. You should see the baseline for comparison and code coverage picked up by the build as in the picture below:



You should also see on the NDepend tab the code metrics of the current build (brown letters) and how it compares to the previous analysis (grey letters).

Summary

What to say? The NDepend plugin is just so easy to install compared to the manual process I had to do with the previous versions. This is a really nice feature to have on your CI server.

TeamCity NDepend plugin test: PASSED!

Tomasz Jaskula

Software craftsman, founder and organizer of Paris user groups for F# and Domain Driven Design. I'm mainly focused on creating software delivering true business value which aligns with the business's strategic initiatives and bears solutions with a clearly identifiable competitive advantage.

I have worked for many companies in the SIRH, e-commerce and financial fields and I have a great experience in solving their real problems for more than 13 years.

Currently working for a big French bank for the Forex financial field building reactive applications in F# and C#. In my free time, I run my startup project on applying machine learning with F# to recruitment field, speak at conferences and user groups, and write blogs and articles for a French magazine for coders called "Programmez !".

You can visit his site jaskula.fr