ndepend

Case Study

Company:
Siemens Healthcare GmbH

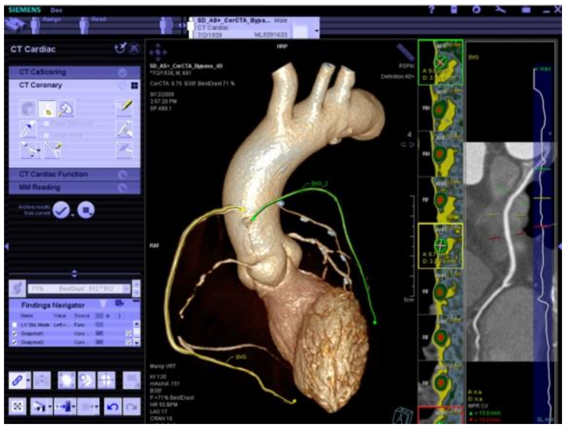
SIEMENS

Siemens Healthcare GmbH offers imaging equipment, information technology, management consulting and services to hospitals, clinics, medical laboratories, and other healthcare businesses. The company's products and solutions address medical needs in a wide variety of medical applications, including angiography, computed tomography, fluoroscopy, magnetic resonance imaging, mammography, nuclear medicine, oncology, patient monitoring, radiography, surgery, ultrasound, urology, and ventilation and anesthesia. Siemens Healthcare has 49 000 employees and operates on all five continents.

Challenges and Objectives

For its next generation of imaging applications, Siemens Healthcare GmbH decided to follow a platform approach to facilitate code reuse and to reduce development costs. But behind the attractive aspects of a platform, plenty of challenges presented themselves:

- **How can we guarantee the stability of APIs?**
- **Can we even make a breaking change?**
- **How do we guarantee quality over builds?**



API Stability

After more than half a decade of development on 5 different sites and involving around 200 developers, the code base has now about 5 million .NET and 500 thousand native Lines of Code (LOC). Most of the technologies provided by .NET platforms like Workflow, WCF, WPF and Direct-Show are being used, but the teams have been facing a continuous challenge: the impact of hundreds of changes and evolutions made each day. Each change, each new line of code, can potentially impact the code behavior at runtime and hence break the code correctness.

This is why at Siemens Healthcare, we invest in automatic testing to detect regression bugs early and to enforce that further refactoring won't break existing code behavior.

In this continuous challenge against **code erosion**, automatic tests are a great way to check code correctness. But automatic tests cannot detect less formalized flaws than bugs.

However, such flaws impact future code maintenance, and having code that works is not enough. New features are constantly requested by our customers. Code is evolving. When dealing with such a large scale code base, each minor change can potentially lead to a future high maintenance cost.

In 2007, the need for a tool like NDepend became clear after a particularly tedious situation we faced. One team developing a basic layer in the system did some API changes without correctly checking the impact. "No one is using this" or "just one line changed". After a few weeks of development, the result was a real tsunami: the whole system was not usable and not deliverable for almost a month and in this case a rollback was not conceivable, we had to make it run again!

Code Base Quality

From a proper quality point of view, how the code is actually structured certainly decides the future cost of maintainability. But preserving a proper code structure is one part of our challenge against code erosion. Transforming business logic into code necessarily appends some sort of fabricated complexity due to implementation details.

Such fabricated complexity typically comes from too complex classes, too-deep inheritance tree, wrong object-oriented programming, methods with too many parameters, complicated state mutation at runtime, poorly named code elements, a lack of abstraction usage to decouple some implementations, poor usage of encapsulation etc. The impact of one occurrence of such code smell is negligible on the whole system quality. But what we wanted to make sure that these wrong practices were the exception and not the rule. We needed a way to finely control the quality of the code developed.

Solutions

API Stability

To address the challenge of controlling the API evolution a first step was to define a communication process to bring transparency for our intern customer. Each developer was asked to document and distribute his breaking changes. Such a procedure might look counter-productive. Indeed, for a small team it is overkill but for more than 100 developers dispatched all over the world it is a way to communicate the changes, and it is a mandatory step to continuously deliver a running system.

NDepend made it easy to deal with dependencies and avoid chaos in our large code base. The tool provides a language named Code Query Language (**CQLinq**). With **CQLinq,** we can query our code base as one would query a relational database with SQL.

It is as fast and easy as:

```
from t in Types where t.IsUsing ("MyType") select t

from t in JustMyCode.Types
where t.DeriveFrom("MyBaseType") &&
      t.Implement("IInterface")
select t
```

And because each client of the platform delivers an NDepend repository to the platform team to be able to check the impact of a change, it becomes really simple to obtain an impact over the different product lines.

Despite these improvements, we were always exposed to unexpected API changes. To be able to reduce the API change leak, we developed a tool based on the NDepend API and the NDepend ability to compare 2 snapshots of our code base.

On the contrary to a source code repository, NDepend is able to make the difference between a change in a comment or in the code itself in the characterization of the code element. This capability was used to develop a tool to definitively detect any breaking changes at a syntactic level and to integrate it as a barrier in the continuous integration build chain. The tool is able to detect about 15 breaking change types: from removing a public class, changing a value of an enumeration, or making a class sealed, and changing class hierarchy. Doing it like this, the developer can work in an efficient way and any breaking changes can still be detected before they reach the consumer and be treated accordingly

Code Base Quality

A second challenge of a big software project is to maintain a low ratio of code smells in the implementation; NDepend proposes some predefined **CQLinq** rules. Each rule can be easily customized to our particular needs. Dozens of code metrics are supported and detecting complex methods can be as easy as writing:

```
warnif count > 0 from m in JustMyCode.Methods
where           // Code Metrics' definitions
  m.NbLinesOfCode > 30
  m.CyclomaticComplexity > 20
  m.ILNestingDepth > 5 ||
  m.NbParameters > 5 ||
  m.NbVariables > 8 ||
  m.NbOverloads > 6

select new { m, m.NbLinesOfCode,
m.CyclomaticComplexity, m.ILNestingDepth,
m.NbParameters, m.NbVariables, m.NbOverloads
 }
```

And now with NDepend v6, we are also able to define our own metric and to visualize it in the metric view. This feature is very useful for the architects when they want to make a decision of a refactoring or test improvements. Making a decision just based on one metric does not make sense as you just see one facet of the problem. But being able to combine metrics (facets) and to weight and visualize them, the matches on the metric view provides an incredible opportunity for analysis and decision support capability to the architects.

Benefits

The benefits of adopting NDepend in our development efforts are clear. The work of our architects is made more concrete and a lot of time is saved. Meantime, the overall code quality has increased, making our code base a better place to develop.

Improved Communication

CQLinq gives our architects and key developers a natural way to formalize expectations in term of code quality, code structure and code evolution. CQLinq rules are named and can contain comments. A CQLinq rule itself represents a concrete artifact to express implicitly to the team a good practice that we want to enforce. When a developer violates a CQLinq rule, he gets immediately informed about the rule they have to follow or the code smell to be avoided.

By dispatching their knowledge through the means of NDepend's capabilities, the guidance of our architects is now more visible and made more concrete to the team. Feedback direction used to be from the architect team to the developers, to inform them about large scale decisions. Now, we often observe that developers come back to us to discuss particular choices that were made.

The whole project benefits from this new synergy. Developers feel more implicated in the respect and definition of our code rules, and architects can make better decisions. This is thanks to being able to gather more accurate information from the code itself.

Save Up to 1 Day per Sprint per Architect

NDepend also helps our architects save time. Now we can program all sorts of rules and constraints to be verified often and automatically through **CQLinq**. The process of code reviewing is relieved from numerous code smells that are fixed at development time. Moreover, reviewing code changes is now made easy thanks to NDepend's diff features.

Time is also saved because of the very interactive nature of **CQLinq** querying. A **CQLinq** query can be refined on a whim and the result is displayed live. Now we get immediate answers when we wonder if some refactored code is well covered by tests, if a library is properly thread-safe, or how many projects will be broken if we do a public API change.

Finally, time is also saved thanks to the ability of NDepend to cope with very large code bases. With more than 3000 Visual Studio projects (including test projects), our code is obviously partitioned in numerous Visual Studio solutions. In this context, answering simple questions like who's dependent on what out of the Visual Studio shell, can quickly become a burden. NDepend lets us visualize high-level facts that involve several Visual Studio solutions.

Lower Maintenance Cost

As mentioned earlier, we experienced that a small mistake in a popular library can lead to weeks of hard work to recover. Trying to formalize and to enforce hundreds of constraints is a way to control what kind of code we want in the future. While we anticipate the evolution of our product and struggle for higher quality code, our continuous goal is code that is cheaper to maintain.

It is hard to define what maintainable code is, but it is easy to express what should be avoided: code not properly layered, not properly encapsulated, components boundaries that are not well defined, code that violates basic code metrics thresholds, all these are flaws that we definitely want to get rid of. NDepend helps us to find and fix existing flaws and prevent future ones. For the last 7 years, we have been investing in a great deal of effort to keep our code maintainable. It is now clear that we already avoided several costly large scale refactoring dramas.

Siemens AG Corporate Technology
Sébastien Andreo
Guenther-Scharowsky-Str. 1
D-91058 Erlangen Germany

Siemens Healthcare Headquarters
Siemens Healthcare GmbH
Henkestraße 127
91052 Erlangen
Germany

Phone: +49 9131 84-0
siemens.com/healthcare