HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

INSTITUTE FOR
NETWORKED SOLUTIONS

Study on Concurrency Checking in .NET

Studienarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Herbstsemester 2016

Autor(en):	Rafael Krucker, Markus Schaden
Betreuer:	Prof. Dr. Luc Bläser
Projektpartner:	Institut für vernetzte Systeme (INS)

Inhalt

Aufgabenstellung

Auftraggeber und Betreuer

Diese Studienarbeit findet als internes Projekt des Instituts für vernetzte Systeme (INS) statt.

Betreuer HSR:

- Prof. Dr. Luc Bläser, Institut für vernetzte Systeme, lblaeser@hsr.ch

Autoren

- Rafael Krucker, rkrucker@hsr.ch
- Markus Schaden, mschaden@hsr.ch

Ausgangslage

Das HSR Concurrency Lab des INS Institut für vernetzte Systeme arbeitet seit längerem an statischen Checker Tools für die Erkennung von Nebenläufigkeits- und Parallelitätsfehlern in Computer-Programmen. So hat das Lab insbesondere verschiedene Lösungen zur Erkennung von Data Races und Deadlocks für C# unter Visual Studio unter Benutzung von Roslyn entwickelt.

Neu sollen weitere Einsatzmöglichkeiten der statischen Analyse für mögliche Nebenläufigkeits- und Parallelitätsproblemen in C# untersucht werden. Einerseits sollen sog. Parallele Code Smells erkannt werden[1]. Andererseits lässt sich eine solche Analyse auch ideal für Continuous Integration einsetzen.

Ziele und Aufgabenstellung

Das Ziel dieser Studienarbeit ist es, eine Studie hinsichtlich der Erkennung von parallelen Code Smells in C# durchzuführen. Dazu soll ein Prototyp entwickelt werden.

Technologisch sollen dafür Microsoft Roslyn für .NET C# eingesetzt werden.

Die Arbeit hat folgende spezifische Ziele:

- Auswahl und Priorisierung von parallelen Code Smells aus der Liste[1] in Absprache mit dem Betreuer.
- Architektur, Entwurf und Implementation eines Roslyn-basierten Checkers, der die ausgewählten Code Smells detektiert.
- Integration des Checkers als Visual Studio Plugin und optional als Continuous Integration Tool (für TFS oder SonarQube etc.)
- Experimentelle Evaluation anhand Open Source Projekten

[1] (Bläser, 2016)

Zur Durchführung

Mit dem HSR-Betreuer finden wöchentliche Besprechungen statt. Zusätzliche Besprechungen sind nach Bedarf durch die Studierenden zu veranlassen. Besprechungen mit dem Auftraggeber werden nach Bedarf durchgeführt.

Alle Besprechungen sind von den Studenten mit einer Traktandenliste vorzubereiten und die Ergebnisse in einem Protokoll zu dokumentieren, das dem Betreuer und dem Auftraggeber per E-Mail zugestellt wird.

Für die Durchführung der Arbeit ist ein Projektplan zu erstellen. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten. An Meilensteinen gemäss Projektplan sind einzelne Arbeitsresultate in vorläufigen Versionen abzugeben. Über die abgegebenen Arbeitsresultate erhalten die Studierenden ein vorläufiges Feedback. Eine definitive Beurteilung erfolgt auf Grund der am Abgabetermin abgelieferten Dokumentation.

Dokumentation

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien der Abteilung Informatik zu verfassen (siehe https://www.hsr.ch/Allgemeine-Infos-Diplom-Bach.4418.0.html?&L=0). Die zu erstellenden Dokumente sind im Projektplan festzuhalten. Alle Dokumente sind nachzuführen, d.h. sie sollten den Stand der Arbeit bei der Abgabe in konsistenter Form dokumentieren. Die Dokumentation ist vollständig auf CD/DVD/USB in 3 Exemplaren abzugeben. Auf Wunsch ist für den Auftraggeber und den Betreuer eine gedruckte Version zu erstellen.

Termine

Siehe auch Terminplan auf https://www.hsr.ch/Termine-Diplom-Bachelor-und.5142.0.html?&L=0

19.09.16	Beginn der Studienarbeit, Ausgabe der Aufgabenstellung durch die Betreuer.
20.12.16	Die Studierenden geben den Abstract zur Kontrolle an Ihren Betreuer/Examinator frei. Die Studierenden erhalten vorgängig vom Studiengangsekretariat die Aufforderung mit den Zugangsdaten zur Online-Erfassung des Abstracts.
23.12.16	Der Betreuer/Examinator gibt das Dokument mit dem korrekten und vollständigen Abstract zur Weiterverarbeitung an das Studiengangsekretariat frei.
23.12.16	Abgabe des Berichtes an den Betreuer bis 17.00 Uhr.

Beurteilung

Eine erfolgreiche Studienarbeit erhält 8 ECTS-Punkten (1 ECTS Punkt entspricht einer Arbeitsleistung von ca. 25 bis 30 Stunden). Für die Modulbeschreibung der Studienarbeit siehe https://unterricht.hsr.ch/staticWeb/allModules/19456_M_SAI.html

Gesichtspunkt	Gewicht
1. Organisation, Durchführung	1/5
2. Berichte (Abstract, Management Summary, techn. u. persönliche Berichte) sowie Gliederung, Darstellung, Sprache der gesamten Dokumentation	1/5
3. Inhalt *)	3/5

*) Die Unterteilung und Gewichtung von 3. Inhalt wird im Laufe dieser Arbeit festgelegt

Abstract

Diese Arbeit soll eruieren, inwiefern das Erkennen von Code Smells im Bereich Parallele Programmierung das Schreiben von Software weniger fehleranfällig machen kann. Als Grundlage wurde dafür ein Paper[2] verwendet, dass 10 solche Code Smells identifiziert und beschreibt.

Das Ziel war das Schreiben einer Library, welche für die statische Code Analyse in C# zuständig ist. Als Grundlage dafür wird der Roslyn Compiler von Microsoft verwendet, welcher eine mächtige API für C# Code-Analysen bereitstellt. Dabei sollte die Library möglichst frei anwendbar sein. So wurden Integrationen für Microsofts Visual Studio sowie SonarQube erstellt, so dass die Analyse von der Kommandozeile bis hin zum Build Sever einsetzbar ist.

Hauptaugenmerk wurde dabei auf Behandlung in die Breite gelegt. In einer Analyse von bekannten Open Source Projekten sowie einigen unbekannten, eher unvollständigen Programmen wurde dann untersucht, ob diese Code Smells in der Praxis vorkommen und auch zuverlässig erkannt werden können. Dabei zeigen sich grosse Unterschiede bei den Häufigkeiten des Auftretens der einzelnen Smells. Es treten bei den verschiedenen Projekten oft die gleichen Muster im Code auf, so dass durch entsprechende Warnungen Programmierer früh auf solche Probleme sensibilisiert werden können. Vergleicht man die Häufigkeit mit den Auswirkungen der Smells auf die Qualität, offenbart sich, welche Smells besonders verfolgt werden sollen.

Die Untersuchung hat gezeigt, dass Smells im Bereich der parallelen Programmierung erkannt werden können und in der Realität auch auftreten. Dadurch kann gesagt werden, dass der Einsatz von Tools, die Smells erkennen, das Programmieren nebenläufiger Anwendungen sicherer und angenehmer macht. Die Untersuchung von realen Projekten hat weiterhin gezeigt, dass nicht alle Smells gleich häufig auftreten. Daher wurde eine Empfehlung gemacht, auf welche Smells sich die Tools konzentrieren sollten.

[2] (Bläser, 2016)

Management Summary

Durch die fortlaufende Digitalisierung und der in dessen Folge immer komplexer werdenden Applikationen, wird der Ruf nach parallel arbeitenden Abläufen immer lauter. Dies birgt aber die Gefahr, dass sich mehrere Abläufe in die Quere kommen und Fehler verursachen. Dies kommt dadurch zu Stande, dass nicht mehr vorhersagbar ist, welche Wege ein Programm genau nehmen wird und an welchen Verzweigungen die übrigen Programmteile aktuell sind. Diesem Problem sind auch erfahrene Softwareentwickler nicht immer gewappnet, weshalb es auch heutzutage noch zu Abstürzen oder Einfrierungen bei bekannten Softwareprodukten kommt. Meist ist dies ein Fehler im Ablauf, welcher in einem spezifischen Zusammenhang übersehen worden ist. Einerseits kann dies zu einem Deadlock führen, was bedeutet, dass die Applikation verklemmt und normalerweise nicht mehr verwendet werden kann. Andererseits, und weitaus schlimmer, können aber sogenannte Race Conditions oder auch Data Races die Folge sein. Diese bleiben meist unentdeckt, da sie nicht immer einen Programmstop zur Folge haben. Durch diese können aber falsche Werte verrechnet werden, was schlussendlich geschäftskritische Auswirkungen haben kann.

Um solchen Gefahren frühzeitig mit gutem Design entgegen zu wirken, wurde eine Applikation geschrieben, welche Code Smells erkennt, die auf schlechtes Design hinweisen oder später solche Fehler hervorrufen können. Es werden dabei keine Race Conditions, Data Races oder Deadlocks aktiv erkannt, sondern deren Auftreten nur indirekt über die Erkennung von Code Smells vermutet, welche zu solchen führen können.

Damit ein solch unschönes Design direkt und einfach während der Entwicklung behoben werden kann, lässt sich das Tool in die Entwicklungsumgebung einbauen, sodass der Entwickler zeitnah sieht, ob er sich an die Richtlinien hält.

Durch die aufwendige Analyse und das Testing des Tools kann gesagt werden, dass sich dessen Einsatz lohnt. Es werden diverse Code Smells gefunden, welche möglicherweise später zu Fehlern führen könnten. Die schnelle Laufzeit von wenigen Sekunden ist definitiv wichtig, damit kein langes Warten notwendig ist um zu sehen, ob Richtlinien verletzt worden sind. Durch die einfache Integration in ein Continuous Integration System kann die Code Qualität sehr leicht erhöht werden.

Für die Zukunft können leicht neue Code Smells Erkennungen eingebaut werden. Dank des modularen Aufbaus können auch andere Entwickler ohne grossen Aufwand neue Checker programmieren oder die bestehenden erweitern.

Technischer Bericht der Arbeit

Der technische Bericht soll eine Übersicht über die geleistete Arbeit geben. Dazu werden in einer Einleitung zuerst die Problematik und das Ziel der Arbeit beschrieben. Es folgt eine Übersicht und Klassifizierung der untersuchten Code Smells, woraufhin auf die Implementation des Checkers eingegangen wird. Nach einer kurzen Beschreibung der eingerichteten Infrastruktur werden Ergebnisse der Analyse einiger Open Source Projekte durch den Checker aufgelistet. Den Abschluss bildet eine Gesamtübersicht und Interpretation dieser Resultate, wonach das Schlussfazit der Arbeit folgt.

Einleitung und Übersicht

Das Ziel dieser Arbeit ist, herauszufinden ob Code Smells in der parallelen Programmierung erkannt werden können und dies dem Programmierer einen Mehrwert bringt. Es gibt bereits eine Fülle von Tools, die dies für bekannte Smells, wie sie etwa Fowler[3] beschreibt, machen und auch jede grössere IDE setzt diese Tools ein. Im Bereich der parallelen Programmierung gibt es jedoch noch keine solchen Tools, wohl auch wegen der fehlenden Taxierung von Smells.

Die Machbarkeitsanalyse beschränkt sich auf einen Prototyp für die .Net Umgebung, die Analyse setzt dabei auf die Roslyn API auf. Der Prototyp trägt den Namen ParaSmeller.

Problematik

Eine gängige Definition eines Code Smells ist «Strukturen im Code die auf einen Verstoss gegen fundamentale Design Prinzipien hinweisen und die Code Qualität negativ beeinflussen»[4]. In der Parallelen Programmierung haben Smells oft mit fehlender oder mangelnder Synchronisation zu tun. Eine Taxierung, auf die sich diese Arbeit stützt, findet sich in (Bläser, 2016). Ein Beispiel für einen Smell in der Parallelen Programmierung könnte etwa wie folgt aussehen:

```
class BankAccount {
   public int Balance { get; private set; }      unsynchronisiert

   public void Deposit(int amount) {
      lock (this) {
          Balance += amount;                     synchronisiert
      }
   }

   public bool Withdraw(int amount) {
      if (amount > Balance) return false;
      Balance -= amount;                         unsynchronisiert
      return true;
   }
}
```

Abbildung 1 Ungenügend synchronisiertes Code Beispiel

[3] (Fowler, 1999)
[4] (Suryanarayana, 2014)

In diesem Fall ist der Zugriff auf das Feld Balance der Klasse BankAccount nur teilweise synchronisiert. Es ist anzunehmen, dass es im Umfeld mit mehreren Threads zu Race Conditions kommt.

Eine genauere Beschreibung aller Smells findet sich im Kapitel Code Smells.

Code Smell Erkennung geschieht meist durch statische Code Analyse. Speziell im Bereich der parallelen Programmierung stösst man hier schnell auf Limitierungen, da bei einigen Smells der Code Fluss genau bekannt sein muss, um ein abschliessendes Urteil über dessen Qualität fällen zu können. Des Weiteren kann gezeigt werden, dass es auch bei Benutzung von dynamischer Analyse Probleme gibt, die nicht gelöst werden können, wie zum Beispiel das Halteproblem. Dies lässt sich eins zu eins in die Welt der Parallelen Programmierung übertragen, in der eine folgende Konsequenz ist, dass nie bei beliebigen Programmen mit abschliessender Sicherheit gesagt werden kann, ob wirklich ein Deadlock auftritt. ParaSmeller versucht deshalb durch statische Analyse ein möglichst grosses Feld häufiger Fälle abzudecken.

ParaSmeller

ParaSmeller soll ein möglichst schlankes und universell einsetzbares Werkzeug sein, welches dem Entwickler hilft, bezüglich der parallelen Programmierung die Codequalität hoch- und die Fehlerquote tiefzuhalten. Dazu kann eine .Net Compilation in jeder beliebigen .Net Sprache als Input dienen, als Output wird eine möglichst generische Liste von Warnungen herauszugeben, die alle Angaben enthält, die weitere Werkzeuge in der Toolchain wie etwa IDEs benötigen, um die Warnungen entsprechend zu verarbeiten.

Eine Analyse mit ParaSmeller läuft in zwei Schritten ab. Erstens wird aus dem Code eine leicht zu analysierende Repräsentationsstruktur aufgebaut, die auf dem Syntaxbaum aufsetzt und ihn ergänzt. Im zweiten Schritt operieren verschiedene Analyzer auf der gleichen Struktur, und melden ihre gefundenen Warnungen. ParaSmeller sammelt alle und bündelt sie zu einem Output.

Ein weiteres Ziel ist die Konfigurierbarkeit von ParaSmeller, es ist einfach, einzelne Smellchecks an- und auszuschalten, zudem bedarf es ausser der Logikprogrammierung keinen Aufwand, weitere Smells und ihre Analyzer hinzuzufügen.

Roslyn API

Der Roslyn Compiler bietet durch eine API diverse Einstiegspunkte, um die Code Analyse durchzuführen. Im einfachsten Fall kann man sich für bestimmte Nodes anmelden, beispielsweise Methodendefinitionen, und kann dann diese entsprechend analysieren. Dabei kann man aber nicht ohne weiteres Nodes herbeiziehen, welche ausserhalb dieses Nodes liegen. So ist zum Beispiel die Nachverfolgung eines Methodenaufrufs in eine andere Klasse nicht möglich.
Um dies doch zu ermöglichen, gibt es aber die Möglichkeit, einen Code Checker am Ende des Kompiliervorgangs zu registrieren. Dieser kann im Gegensatz auf sämtliche Klassen zugreifen und so klassenübergreifende Analysen durchführen. Der grosse Unterschied liegt dabei in der Performance. Node basierte Checker werden standardmässig bei jeder Eingabe ausgeführt und sollten daher auch sehr schnell arbeiten, damit der Entwickler zeitnah eine mögliche Warnung sieht. Die «Solution Wide Checkers» sind dagegen standardmässig sogar deaktiviert. Diese müssen in den Optionen zuerst aktiviert werden[5]. Dadurch, dass Visual Studio selber

[5] (Stackoverflow, 2016)

solche Checker bereits besitzt, welche dann auch ausgeführt werden, wird der Buildprozess deutlich länger. Doch am Ende erhält man eine Übersicht über Smells, welche erkannt worden sind.
Um eine einheitliche Grundstruktur für die Analyse verwenden zu können, wurde entschieden, dass der Checker sich einmalig nach dem Kompiliervorgang registriert und alle Smellanalysen auf dieser Grundstruktur ausgeführt werden.

Code Smells

In diesem Kapitel werden die behandelten Code Smells beschrieben. Des Weiteren findet sich eine Klassifizierung bezüglich der erwarteten Auftrittshäufigkeit in der Realität sowie die geschätzte Schwierigkeit, eine Analyse für solch eine Smell zu implementieren. Diese Einschätzungen werden im Kapitel Ergebnisse mit dem tatsächlichen Auftreten verglichen. Eine detaillierte Beschreibung, wie die einzelnen Smells vom Prototyp erkannt werden, ist im Anhang zu finden.

Anmerkungen

An vielen Stellen sind Properties, Methoden und Konstruktoren von Klassen ähnlich beziehungsweise fast gleich zu behandeln. Um eine dauernde Wiederholung dieser Ausdrücke zu vermeiden, wird in den folgenden Beschreibungen jeweils der Überbegriff Member benutzt. Falls eine Unterscheidung erforderlich ist, wird wieder der jeweilige passende Begriff benutzt.

Halb-synchronisierte Klasse

Beim Smell Halb-synchronisierte Klasse werden Members sowohl synchronisiert und unsynchronisiert innerhalb von anderen Members aufgerufen. Dies ist gefährlich, da dies zu Race Conditions führen kann, nicht immer ist sichergestellt, dass beim Zugriff auch der neuste Wert zurückgegeben wird. Dies kann in nebenläufigen Programmen zu Fehlern führen, doch muss dies nicht immer falsche Resultate liefern. Das Design ist fragwürdig und sollte entsprechend verbessert werden.

Fire and Forget

Bei diesem Smell geht es darum, dass ein Task gestartet wird, dessen Ende aber nie abgewartet wird. Dies ist insofern problematisch, dass normalerweise bei solchen auf Exception Handling komplett verzichtet wird. So bleiben Exceptions unentdeckt und können Fehler nach sich ziehen. Zusätzlich ist nicht sichergestellt, dass Tasks kontrolliert beendet werden. Falls die Applikation gestoppt wird, können solche mitten in einer Berechnung abgebrochen werden.

Einfaches Monitor Wait oder Signal

Bei Einfaches Monitor Wait oder Signal wird die Nutzung des Monitor Locks betrachtet. Bei der Prüfung einer Bedingung ohne Schleife kann es zu Fehlern kommen. Dies kommt dadurch zu Stande, dass sich die Wartebedingung während des Aufweckens bereits wieder verändert haben könnte und nun nicht mehr zutrifft. Daher muss diese zwingend in einer Schleife geprüft werden, um sicherzustellen, dass diese immer noch zutrifft.

Die andere Problematik ist die Nutzung von Signal, in C# Pulse genannt. Diese Funktion erlaubt das Wecken eines einzelnen oder das Wecken aller Threads, die zurzeit auf ein bestimmtes Lock warten. Das Aufwecken eines einzelnen kann aber zum Stopp des Programms führen, wenn mehrere Wartebedingungen zum Einsatz kommen. So könnte zwar ein einzelner Thread

weitermachen, doch wird ein Thread aufgeweckt, für welchen die Wartebedingungen nicht zutrifft, woraufhin auch dieser wieder wartet. Nun ist kein Thread mehr aktiv und es wird kein Fortschritt mehr erzielt.

Explizite Threads

Bei diesem Code Smell geht es um die Verwendung von expliziten Threads. Dies muss nicht schlecht sein, doch falls damit kurze Operationen ausgeführt werden sollen, sollten hierfür Tasks verwendet werden. Diese skalieren viel besser, sodass hunderte Tasks mit einer kleinen Anzahl an Threads ausgeführt werden können. Diese werden in einem Thread Pool ausgeführt, welcher Threads wiederverwendet. So muss nicht jedes Mal ein neuer Thread erstellt werden. Auch ist es so möglich, kleine Tasks zu gruppieren und so Abläufe zu definieren.

Falls Threads Warteabhängigkeiten untereinander haben, können diese nicht als Tasks verwendet werden, da es sonst zu Deadlocks kommen kann. Mehr dazu aber im nächsten Kapitel, da dies ebenfalls ein Code Smell ist.

Warteabhängigen unter Tasks

Wie in der obigen Beschreibung erwähnt, sind Warteabhängigkeiten unter Tasks gefährlich, da dies zum Blockieren des Programmes führen kann. Dies geschieht, falls ein Task auf etwas wartet, was ein anderer Task erfüllen muss. Das Problem ist, dass nur eine begrenzte Anzahl an Tasks ausgeführt werden kann, was durch die geringe Anzahl verfügbarer Threads bedingt ist. Falls nun alle laufenden Tasks auf ein Resultat warten, welches ein zukünftiger Task bereitstellen könnte, friert das Programm ein, da der Pool bereits komplett ausgelastet ist. Damit dies nicht geschieht, sollte man von solchen Abhängigkeiten absehen, ausgenommen ist das Warten auf das Ende von Sub -Tasks.

Geschachtelte synchronisierte Methodenaufrufe

Der Code Smell Geschachtelte synchronisierte Methodenaufrufe zeichnet sich dadurch aus, dass das Ausführen einer Methode mehrere geschachtelte Locks zur Folge hat. Falls sich keine Lockingreihenfolge überlegt wurde, kann es zu einem Deadlock kommen. Aus diesem Grund ist es wichtig, bei verschachtelten Locks eine lineare Sperrordnung zu verwenden. Diese schliesst zyklische Warteabhängigkeiten aus und damit auch Deadlocks.

Über-Asynchronität

Durch die zunehmende Verbreitung der asynchronen Programmierung wird teilweise sogar die kleinste Methode noch asynchron ausgeführt. Doch bringt dies nicht immer Performance Steigerung, sondern birgt auch Gefahren. Falls die komplette Logik nur noch asynchron durchgeführt wird, verschlechtert dies nicht nur die Lesbarkeit, sondern erhöht auch die Anzahl Context Switches. Dadurch wird das Programm insgesamt langsamer und Data Races oder Race Conditions können auftreten.

Daher sollte man nur die Hauptlogik asynchron programmieren und den Rest synchron. Dies ist meist ausreichend, falls keine UI Zugriffe notwendig sind.

Atomic, Volatile und Yield

Die Verwendung von primitiven Synchronisierungsmechanismen für die Lock-Freie Programmierung birgt viele Fehlermöglichkeiten und sollte im Normalfall nicht verwendet werden. Diese werden für Low-Level Algorithmen oder Datenstrukturen benützt, welche bereits

vom .Net Framework den Entwicklern zur Verfügung gestellt werden. Daher ist deren Nutzung meist nur in Performance kritischen Applikationen und mit dem nötigen Knowhow in Ordnung. Allenfalls kann es schnell zu Fehlern kommen, falls das Memory Model nicht wirklich verstanden worden ist.

Finalizer

Den meisten Entwicklern ist gar nicht bewusst, dass Finalizer in einem anderen Thread als der Rest der Applikationslogik laufen. So werden dort meist Synchronisationsmechanismen vergessen, welche aber für den korrekten Zugriff auf gemeinsame Ressourcen notwendig wären. So sollte bei der Verwendung von Finalizers genau hingeschaut werden, ob und auf welche Objekte zugegriffen wird.

Versuchsweiser Ressourcenbezug

Der wiederholte Versuch, mit Hilfe eines Timeouts, eine Ressource zu erhalten, kann einen Code Smell darstellen. Falls dies innerhalb einer Schleife ohne Abbruchkriterium geschieht, kann Starvation auftreten. Es ist nicht sichergestellt, dass diese Aktion jemals ausgeführt wird und nicht ein anderer Thread jeweils zuvorkommt. Daher sollte auf solche Logik verzichtet und traditionelle Lockmechanismen verwendet werden.

Klassifizierung und Erwartung

Smell	Identifikation	Erwar. Schwierigkeit	Erwar. Häufigkeit
Halb-synchronisierte Klasse	HSC	Easy	Häufig
Fire and Forget	FaF	Easy	Häufig
Einfaches Monitor Wait oder Signal	MWS	Easy - Hard	Selten
Explizite Threads	ET	Easy	Häufig
Warteabhängigen unter Tasks	WCT	Hard	Mittel
Geschachtelte synchronisierte Methodenaufrufe	NSMC	Hard	Häufig
Über-Asynchronität	OA	Challenge	Mittel
Atomic, Volatile und Yield	PS	Easy	Mittel
Finalizer	FS	Easy	Selten
Versuchsweiser Ressourcenbezug	TRR	Challenge	Selten

Tabelle 1 Klassifizierung und Erwartungen der 10 Smells

Zu Beginn der Studienarbeit wurden die 10 Smells jeweils der erwarteten Schwierigkeit, sie zu implementieren, zugeteilt. Dabei werden die Kategorien Easy, Challenge und Hard benutzt. Dies stellte aber nur eine ungefähre Einschätzung dar. Je nachdem, wie genau ein Smell identifiziert werden soll, ist die Schwierigkeit unterschiedlich. So können viele Smells mit einfachen Mitteln gefunden werden, doch befinden sich dann eine grosse Anzahl an False Positives ebenfalls darunter. Zusätzlich gibt es Smells, die nicht als in Stein gemeisselt betrachtet werden können, sondern eher in den Bereich der Stilfragen gehören. Ein Beispiel hierfür ist der Smell Über-Asynchronität, zudem gibt es dort zu setzende Parameter wie etwa die notwendige Verschachtelungstiefe, bei der der Checker anschlagen soll.

Weiterhin wurde eine Schätzung abgegeben, wie häufig die Smells jeweils in der Realität auftauchen. Die Einteilung gliedert sich dabei in selten, mittel oder häufig.

Library

Dieses Kapitel beschäftigt sich mit der Implementation. Zuerst wird die Struktur der erstellten Library erklärt, wonach aufgezeigt wird, wie die Analyse abläuft. Eine genauere Beschreibung der Architektur findet sich im Anhang.

Um ein möglichst einfaches Arbeiten mit den individuellen Smell Analyzern zu ermöglichen, wurde entschieden, eine Library zu schreiben, welche bereits viele Dienste bereitstellt, die man oft benötigt. Die Library besteht dabei aus zwei Teilen.

Code Repräsentation

Der erste Teil der Library nimmt Source Code entgegen und bringt ihn in eine Form, welche einfacher analysiert werden kann. Dazu werden folgende Abstraktionen gebildet: Solution – Class – Member – Body – Invocation, wobei Bodies jeweils Codeblöcke wie etwa ein Block in einem if-Statement oder einer while Schleife darstellen. Diese können natürlich beliebig verschachtelt sein. Invocations repräsentieren dabei den Aufruf eines Members im Code, also zum Beispiel einen Methodenaufruf oder den Zugriff auf ein Property.

Das Ziel dabei war, es möglich zu machen, vom Aufruf eines Members direkt auf dessen Implementation zu kommen. Durch die Unterscheidung von synchronisierten und unsynchronisierten Codeblöcken ist es zudem einfach zu untersuchen, ob ein Aufruf synchronisiert ist oder nicht und ob dessen Implementation wiederum Synchronisationen beinhaltet, was bei der Analyse von Codesmells oft gebraucht wird. Für jede Abstraktionsstufe wurden Factories bereitgestellt, die aus einem Syntaxbaum alle weiteren Strukturen auslesen.

Will man einen Überblick über ein komplettes Projekt haben, kann man die Analyse erst nach der kompletten Kompilierung des Projektes starten, da erst danach alle für die Verknüpfung nötigen Informationen bereit sind.

Für alle anderen Abstraktionsstufen kann man sich aber bereits nach dem Erstellen des syntaktischen Modells einklinken, so sind alle Factories einzeln ansprechbar und es muss nicht stets die ganze Solution aufgebaut werden.

Das Matching von Aufrufen auf Implementationen wird über das semantische Modell vorgenommen. Hierbei herrscht jedoch durch Interfaces oder Vererbungshierarchien nicht immer eine Eindeutigkeit, sodass die Implementation als eine Liste von potentiellen Implementationen modelliert wurde, welche von den Smell-Analyzern alle verfolgt werden. Dies ist bei statischen Code Analysen nicht anders möglich.

Eine detailliertere Einsicht auf das Modell und eine tiefergehende Erklärung finden sich im Anhang.

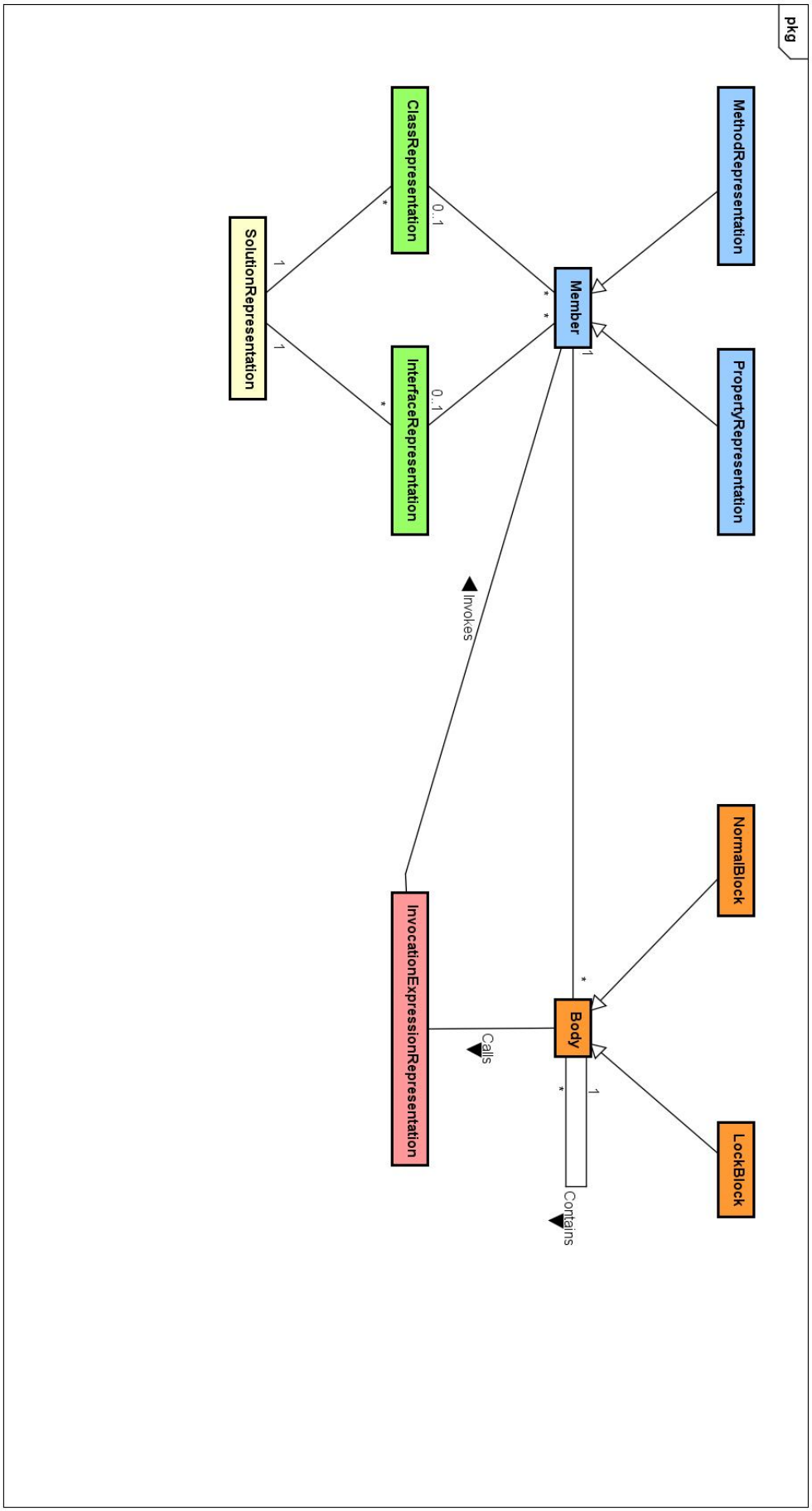


Abbildung 2 Domain Modell

Analyse

Der zweite Teil der Library besteht aus dem Bereitstellen einer Infrastruktur für das Einklinken von verschiedenen Smell-Analyzern. Falls ein neuer Smell rapportiert werden soll, so muss nur die Logik für dessen Erkennung in einem Reporter, der eine gegebene Klasse erweitern muss, implementiert werden. Dabei hat der Reporter die Möglichkeit, für verschiedene Abstraktionsstufen Erkennungslogiken zu registrieren, etwa über Klassen hinweg oder nur innerhalb eines bestimmten Membertyps.

Die Library nimmt dann die Erstellung der Analyse-Struktur vor und legt sie dem Reporter vor. Des Weiteren wird nach der Entdeckung von Smells durch einen Reporter auch das Rapportieren der Diagnosen in einer einheitlichen Form durch die Library geregelt. Zudem ist konfigurierbar, welche Smelltypen rapportiert werden sollen, default-mässig werden alle definierten Smells gemeldet.

Ein detailliertes Diagramm sowie eine vertiefte Erklärung dazu findet sich im Anhang.

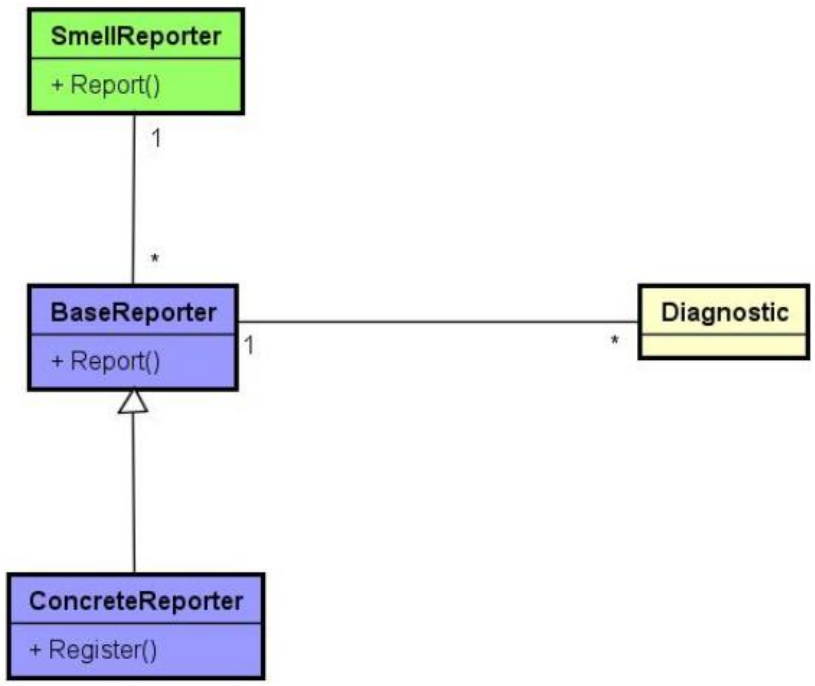


Abbildung 3 Reporting Framework

Code Helpers

Ebenfalls werden häufig Suchabfragen auf einen Syntaxbaum abgesetzt, die zum Beispiel die Form «Alle Elemente X die die Eigenschaft Y haben» besitzen. Auch für diese Fälle werden in der Library Grundstrukturen angeboten, die man oft verwendet. Zudem werden solche Filter auch direkt auf den oben beschriebenen Abstraktionen angeboten, falls es Sinn macht. So braucht man zum Beispiel oft alle synchronisierten Methoden auf einer Klasse oder alle unsynchronisierten Aufrufe, welche ein Property Accessor vornimmt.

Die freien Filter werden als Extensionmethoden auf SyntaxNode-Typen implementiert, um einfaches manövrieren in Syntaxbäumen zu ermöglichen.

Falls die Filter direkt mit den Abstraktionen zu tun haben, werden sie direkt als Klassenmember implementiert oder ebenfalls als Extensionmethoden, aber auf den Abstraktionsklassen.

Bei den Codefixes müssen zudem neue Syntaxelemente erstellt und in Syntaxbäume eingehängt werden. Für die wiederkehrenden Fälle wurden dazu Builder implementiert, die einem das Erstellen dieser Strukturen abnehmen. Auch für kosmetische Transformationen wie korrekte Einrückungen und so weiter wurden Hilfsstrukturen implementiert, um unnötigen Boilerplate Code zu reduzieren.

Implementationsdetails

Dieses Kapitel beschäftigt sich mit einigen Aspekten, die bei der Implementation besondere Beachtung geschenkt wurde. So wird insbesondere auf anfängliche Performanceprobleme eingegangen, woraufhin beschrieben wird, wie die Library getestet wurde und wie man sie in Visual Studio integrieren kann. Technische Details dazu finden sich in im Anhang.

Performance

Durch die Analyse einiger grossen Projekte zeigte sich, dass speziell das Matching von Methodenaufrufen zu ihren Zielen sich als einer der grössten Schwierigkeiten bezüglich der Performance gestaltet. Dies ist dadurch bedingt, dass sämtliche Aufrufe analysiert werden, welches schnell mal mehrere 10'000 sind, und für jeden mehrere Vergleiche notwendig sind. Deshalb wurde mit folgenden Heuristiken versucht, den Vorgang schneller zu gestalten: Zuerst wird ein Matching auf der lokale Klasse durchgeführt. Falls dort keine identische Methode gefunden wird, wird die Hierarchie in Betracht gezogen. Wird dort eine abstrakte Klasse oder ein Interface gefunden, welches diese Methode spezifiziert, werden alle implementierenden Klassen als mögliches Ziel hinzugefügt.

Dieser ganze Aufwand macht sich am Ende aber bezahlt. Es ist so sehr einfach möglich, von einem Aufruf auf alle möglichen Ziele zuzugreifen und weitere Analysen durchzuführen. Um den Prozess schneller zu gestalten, werden alle Klassen und Interfaces in Maps gespeichert, welche den direkten Zugriff durch vollständigen Klassennamen bieten. Dies allein machte die Ausführung 50 – 100x schneller. Diese Massnahmen verringerten die Analysezeit von NHibernate auf einem handelsüblichen Laptop von anfänglich 3 Stunden auf 30 Sekunden.

Visual Studio Integration

Die Integrierung des Checkers in Visual Studio ist sehr einfach gehalten, es wurde eine schlanke Implementation eines DiagnosticAnalyzers erstellt, der sich leicht als Nuget-Package in Visual Studio installieren lässt. Im Analyzer wird lediglich eine CompilationAction registriert, die den Compilation - Context an die Analyse-Library weiterreicht und die zurückgegeben Diagnosen in Visual Studio – eigene Diagnostics umwandelt, welche dann dem Context gemeldet werden.

Durch Spezialisierungen dieser Implementation ist es möglich, sich nur für ausgewählte Smelltypen zu registrieren.

Ein detailliertes Diagramm sowie eine vertiefte Erklärung dazu findet sich im Anhang.

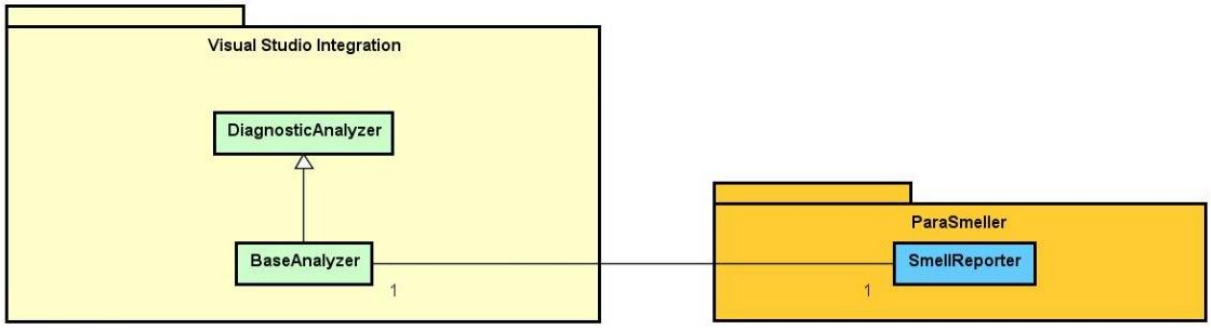


Abbildung 4 Visual Studio Integration

Testing

Um sicherzustellen das die Analyzer auch korrekt funktionieren, ist es selbstverständlich notwendig, diese testen. Glücklicherweise gibt es bereits ein Framework, nämlich das CodeAnalyzerTestingFramework von Microsoft, welches dies einfach erlaubt. Dazu schreibt man fehlerhaften Code als Input und definiert, an welchen Stellen welcher Code Smell detektiert werden soll. Durch den Aufbau der Library und dadurch, dass im echten Analyzer alle Checks durchführt werden, wurde entschieden, dies beim Testing anders zu lösen. Es wurden für jeden Smell einen eigenen Checker und eine Testklasse erstellt. So kann jeder Checker isoliert getestet werden und bei Fehlern wird das Debugging erleichtert.

Ein detailliertes Diagramm sowie eine vertiefte Erklärung zum Testing dazu findet sich im Anhang.

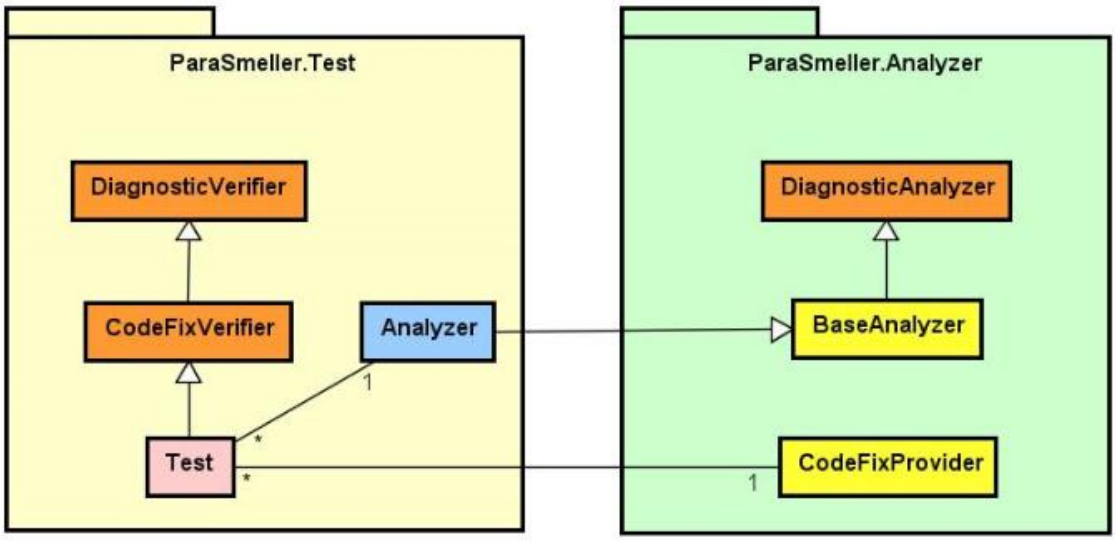


Abbildung 5 Testing Aufbau

Continuous Integration

In diesem Kapitel wird beschrieben, wie die Infrastruktur für die Implementation des Checkers aufgebaut ist.

Um die Qualität des Checkers sicherzustellen, wurde eine Continuous Integration Infrastruktur aufgebaut, welche bei jedem Check-In alle Tests des ParaSmellers ausführt. So sieht man zeitnah, ob ein Commit zu Fehlern geführt hat. Zusätzlich wird über Nacht immer die neuste Version des ParaSmellers auf SonarQube deployt.

Übersicht

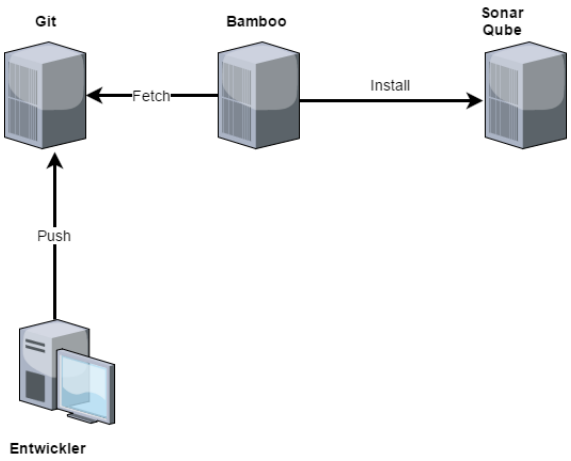


Abbildung 6 Übersicht Continuous Integration

Build & Testing

Es gibt aktuell 3 Buildprojekte. Zwei für den ParaSmeller und eines für das Testprojekt.

ParaSmeller:

> Build & Test: Builded den Checker und führt alle Tests aus
>
> Create Nuget Package: Builded den Checker (Release) und speichert das nupkg

Testprojekt

> Build and Analysis: Builded das Testprojekt mit diversen Code Smells und führt eine Analyse auf SonarQube aus. So kann man sehen, ob die SonarQube Integration sauber arbeitet.

Deployment

Bamboo bietet zusätzlich die Möglichkeit, Deployments auszuführen. Dies erlaubt es einfach, den Checker in SonarQube zu deployen.

Dabei wird das nupkg, welches beim «Create Nuget Package» gespeichert wurde, zuerst in ein SonarQube Plugin umgewandelt. Detaillierte Informationen dazu finden sich im Anhang. Danach wird SonarQube gestoppt, das alte Plugin gelöscht und das neue hinzugefügt. Anschliessend wird Sonar wieder gestartet.

Dies findet entweder manuell statt, oder jeweils über Nacht mit der neusten Version des Checkers.

Ergebnisse

In diesem Kapitel geht es um die Ergebnisse, die durch die Untersuchung von einigen Open Source Projekte mit dem Checker gewonnen wurden. Zuerst wird kurz jedes einzelne untersuchte Projekt und die Resultate der Analyse angerissen. Am Schluss folgt ein generelles Fazit der Ergebnisse, bei dem versucht wird, auf die Relevanz der jeweiligen Smells einzugehen und die Qualität ihrer Erkennung zu besprechen. Die detaillierten Bewertungen der einzelnen Projekte finden sich im Anhang.

Log4Net

Log4Net ist eine Logging Library, welche viele Möglichkeiten bietet, Logs zu schreiben. So kann das Ziel eine Datei, ein entferntes Zielsystem oder direkt die Konsole sein.

Grösse	**10'000 LOC**
Quelle	https://github.com/apache/log4net

Übersicht Code Smells

Abbildung 7 Übersicht Code Smells Log4Net

Code Smell	Anzahl	Beschreibung
HSC001	18	Halbsynchronisierte Klasse
HSC002	5	Halbsynchronisierte Klasse
ET001	17	Explicit Threads
OA001	4	Überasynchronität
OA002	2	Überasynchronität
PS001	4	Primitive Synchronisierungsmechanismen
TRR001	2	Timeouts

Tabelle 2 Gefundene Code Smell von Log4Net

Fazit

Log4Net ist weit verbreitet und wird oft eingesetzt, dem entsprechend sind die meisten Warnungen auch nicht gravierend, beziehungsweise ist deren Verfolgung eine Stilfrage. Es tauchen bei der Analyse jedoch einige wenige Stellen auf, die echtes Gefahrenpotential bergen, welche man mit dem Wissen darum zudem auch leicht beheben könnte.

NHibernate

NHibernate ist ein Open Source ORMapper für das .NET Framework, welcher sehr weit verbreitet ist und auch einen grossen Codeumfang hat.

Grösse	**400'000 LOC**
Quelle	https://github.com/nhibernate/nhibernate-core

Übersicht Code Smells

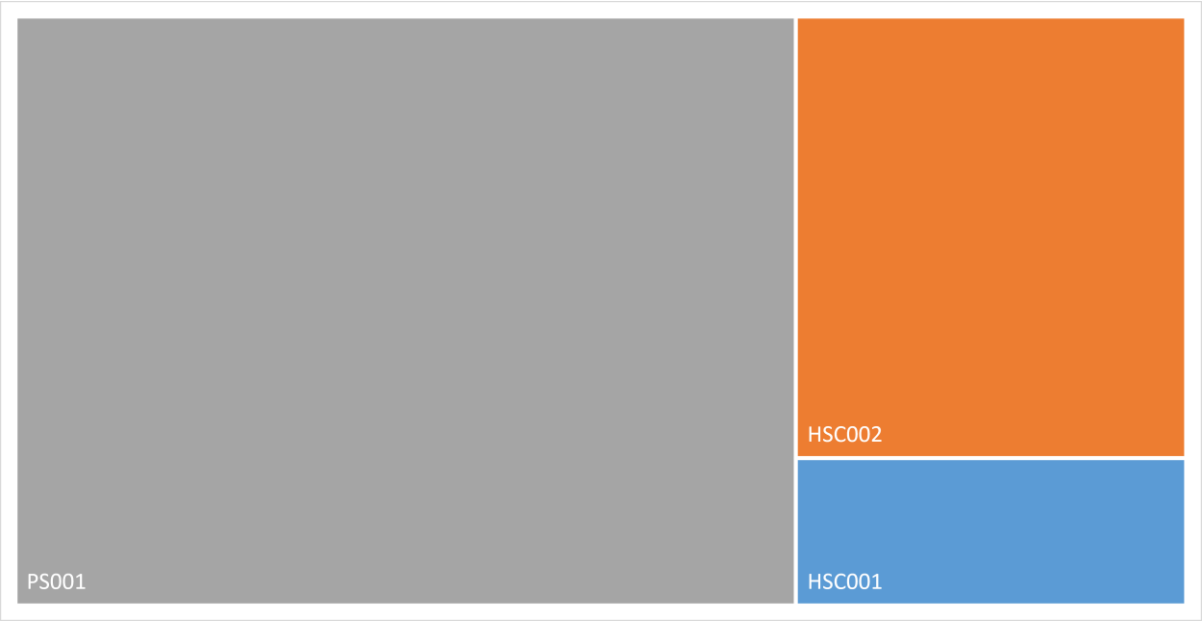


Abbildung 8 Übersicht Code Smells NHibernate

Code Smell	Anzahl	Beschreibung
HSC001	1	Halbsynchronisierte Klasse
HSC002	3	Halbsynchronisierte Klasse
PS001	8	Primitive Synchronisierungsmechanismen

Tabelle 3 Gefundene Code Smell von NHibernate

Fazit

Bei einem weit verbreiteten Projekt wie NHibernate ist zu erwarten, dass nicht sehr viele gravierenden Fehler oder potentielle Fehlerquellen im Code lauern. Abgesehen von den Warnungen vor Low-Level Synchronisation, welche hier nicht angebracht sind, wurden keine Falls Positives gefunden. Die Diagnose von nur 12 Warnungen bei solch einer Codemenge erscheint jedoch ziemlich klein. Durch verfeinerte Analysen dürfte noch mehr im Code zu entdecken sein.

OpenRA

OpenRA ist eine Real Time Strategy Game Engine, die komplett in C# geschrieben ist und 200 Contributors hat.

Grösse	**320'000 LOC**
Quelle	https://github.com/OpenRA/OpenRA

Übersicht Code Smells

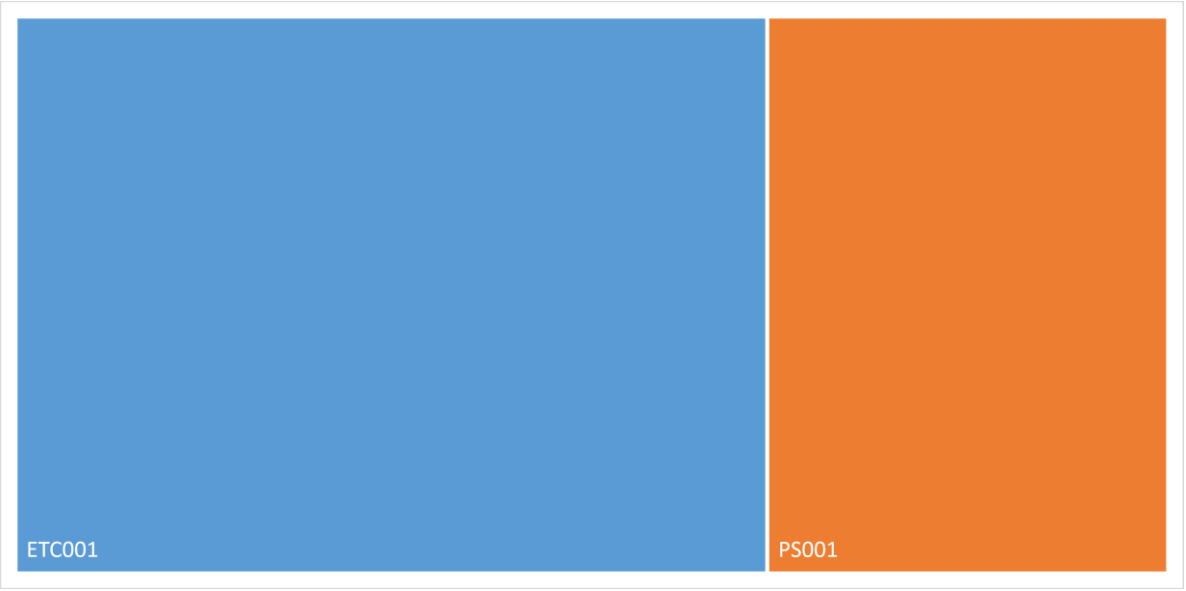


Abbildung 9 Übersicht Code Smells OpenRA

Code Smell	Anzahl	Beschreibung
ET001	15	Explicit Threads
PS001	8	Primitive Synchronisierungsmechanismen

Tabelle 4 Gefundene Code Smell von OpenRA

Fazit

Wie bei einer Game Engine, bei der Performance und Parallelität wichtige Anforderungen sind, zu erwarten war, meldet der Checker das Vorhandensein von einigen primitiven Synchronisierungsmechanismen sowie viele Verwendungen der Thread-Klasse. Diese sind in dieser Domäne natürlich legitim, doch es böte sich die Ablösung von expliziten Threads an. Es werden keine potentiellen Fehler im Bereich der parallelen Programmierung gemeldet, was bei einer weit verbreiteten Game Engine auch glaubwürdig erscheint, da diese wohl schnell zu Symptomen in der Anwendung führen würden.

Roslyn

Roslyn ist der C# / VB Compiler von Microsoft. Er ist zentraler Bestandteil für die Entwicklung von Programmen in C# oder Visual Basic und wurde zudem für die Entwicklung der in dieser Arbeit vorgestellten Analyzer gebraucht.

Grösse	**1'500'000 LOC**
Quelle	https://github.com/dotnet/roslyn

Übersicht Code Smells

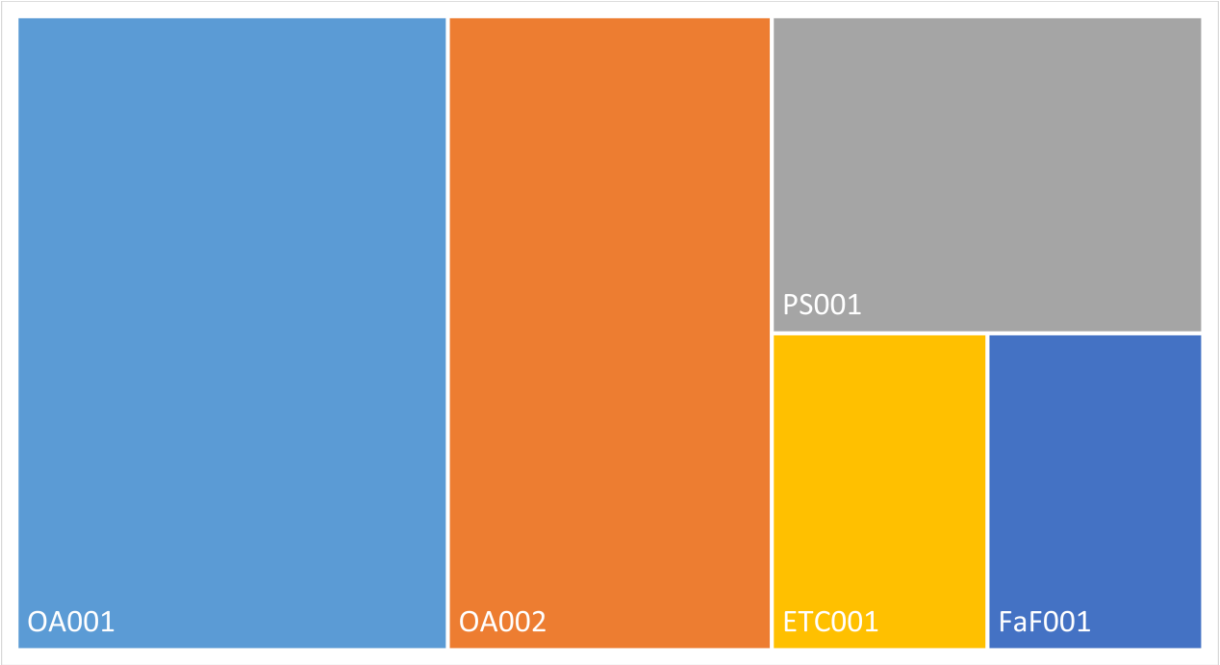


Abbildung 10 Übersicht Code Smells Roslyn

Code Smell	Anzahl	Beschreibung
ET001	1	Explicit Threads
OA001	4	Überasynchronität
OA002	3	Überasynchronität
PS001	2	Primitive Sychronisierungsmechanismen
FaF001	1	Fire and Forget

Tabelle 5 Gefundene Code Smell von Roslyn

Fazit

Der Checker findet bei Roslyn keine schlimmen Smells, welche gefährlich für dessen Ausführung sein könnten. Dies ist sicherlich dadurch bedingt, dass an diesem Projekt viele erfahrene Entwickler arbeiten und das meiste sequentiell aufgebaut ist. Die schiere Projektgrösse lassen aber vermuten, dass es doch einige kritische Stellen geben könnte, die man durch eine detailliertere Analyse erkennen würde.

Dining Philosophers

Ein einfaches Beispiel des bekannten «Dining Philosophers» Problem, welches bei der Ausführung in einem Deadlock endet.

Grösse	**100 LOC**
Quelle	https://code.msdn.microsoft.com/windowsdesktop/Dining-Philosophers-in-C-9ec1dcef

Übersicht Code Smells

Abbildung 11 Übersicht Code Smells Dining Philosophers

Code Smell	Anzahl	Beschreibung
WCT001	2	Waiting Conditions mit Tasks

Tabelle 6 Gefundene Code Smell von Dining Philosophers

Fazit

Leider konnte die Sperrordnung nicht gefunden werden, was aber dadurch gegeben ist, dass diese durch Parameter in einer Methode überreicht wird und es so dem Checker nicht möglich ist vorher zu sagen, ob es zu einem Deadlock kommen kann oder nicht. Immerhin wird aber durch die Erkennung von Synchronisierungsmechanismen in Threads auf die Möglichkeit hingewiesen, dass es zu einem Deadlock kommen könnte.

SignalR

SignalR ist eine Library für ASP.Net, welche es sehr einfach machen soll, einer Webapplikation Realtime Funktionalität hinzuzufügen.

Grösse	**27'000 LOC**
Quelle	https://github.com/SignalR/SignalR

Übersicht Code Smells

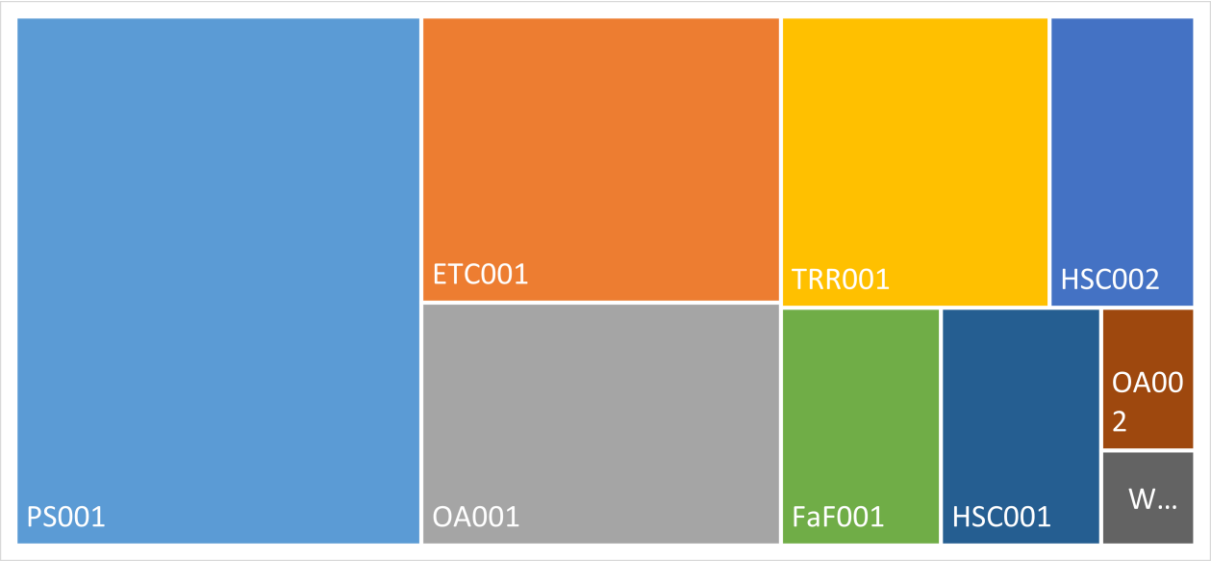


Abbildung 12 Übersicht Code Smells SignalR

Code Smell	Anzahl	Beschreibung
HSC001	17	Halbsynchronisierte Klasse
HSC002	19	Halbsynchronisierte Klasse
ET001	46	Explicit Threads
OA001	39	Überasynchronität
OA002	6	Überasynchronität
PS001	96	Primitive Sychronisierungsmechanismen
TRR001	35	Timeouts
FaF001	17	Fire and Forget
WCT001	4	Waiting Conditions in Tasks

Tabelle 7 Gefundene Code Smell von SignalR

Fazit

Das Framework ist im grossen Ganzen gut aufgestellt. Es wird speziell darauf geachtet, dass keine Warnungen vom Compiler angezeigt werden, auch werden sonst gute Programmierpraktiken eingehalten.

Die Methoden sind aber oft lang und teilweise recht komplex. Die vielen Async-Verschachtelungen und Context-Switches machen es zusätzlich schwierig, das Programm genau zu verstehen. Zudem wurden in der Analyse einige gefährliche Stellen gefunden, welche eine lauernde Quelle für auftretende Symptome wie Data Races sind. Durch die Rückmeldungen der Analyse könnte man das Projekt an einigen Stellen sicherer und wartbarer machen.

ParaSmeller

Auch der Checker selbst wurde analysiert, er ist im Vergleich zu den untersuchten Projekten sehr klein.

Grösse	**1’200 LOC**
Quelle	https://github.com/currencyCon/ParaSmeller

Übersicht Code Smell

Abbildung 13 Übersicht Code Smells ParaSmeller

Code Smell	Anzahl	Beschreibung
PS001	2	Primitive Synchronisierungsmechanismen

Tabelle 8 Gefundene Code Smell von ParaSmeller

Fazit

Wie zu hoffen war, findet sich bei der Library selber keine schwerwiegenden Fehler und die gefundenen Warnungen sind im gegeben Kontext als irrelevant zu betrachten.

Gesamtübersicht

Durch die vorherigen Analysen kann nun eine Auswertung über die gefundenen und nicht gefundenen Code Smells gemacht werden.

Zu Beginn wird die Häufigkeit der einzelnen Smells aufgezeigt, anschliessend folgt eine Erklärung, warum nicht alle Smells gefunden worden sind.

Verteilung

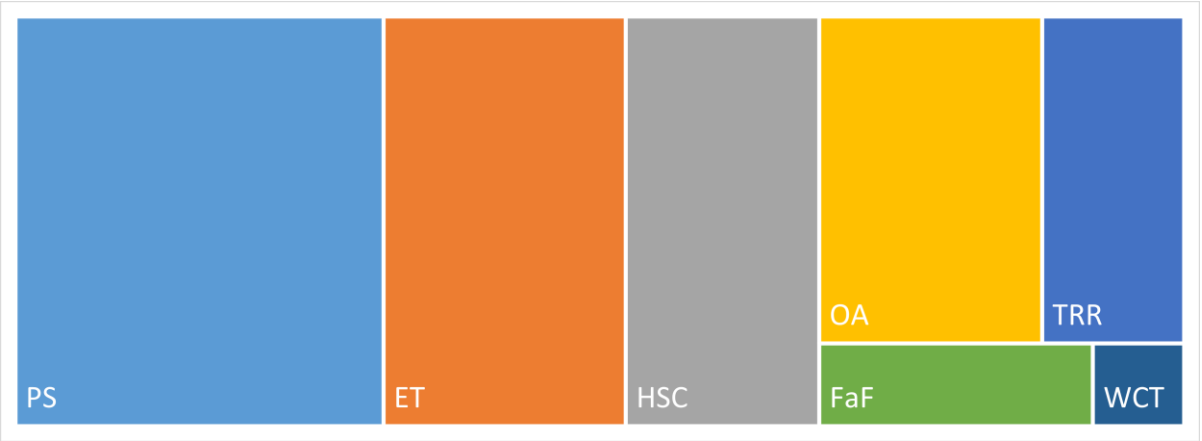


Abbildung 14 Gesamtübersicht Code Smells

Code Smell	Anzahl	Beschreibung
PS	120	Primitive Synchronisierungsmechanismen
ET	79	Explicit Threads
HSC	63	Halbsynchronisierte Klasse
OA	58	Überasynchronität
TRR	37	Timeouts
FaF	18	Fire and Forget
WCT	6	Waiting Conditions in Tasks

Tabelle 9 Gesamtübersicht aller gefundener Code Smells

In der Verteilung wurden die Smells zusammengefasst, falls sie vom gleichen Typ sind. So wurden HSC001 und HSC002 als HSC abgebildet.

Die am häufigsten gefundenen Smells sind «Primitive Synchronisationsmechanismen» und die Verwendung von «Explicit Threads». Dies ist gut nachvollziehbar, da speziell SignalR, welche hohe Anforderungen an Performance hat, viele solcher verwendet. Dadurch, dass Tasks erst seit dem .Net Framework 4.0 verfügbar sind, wurden viele dieser Projekte mit Threads programmiert und müssten nun umgestellt werden. Andererseits gibt es auch einige Aufrufe von Thread.Sleep, welche von dieser Anzahl noch abgezählt werden müssten.

Anschliessend folgt in der Häufigkeit der «Halbsynchronisierte Klasse» Smell, dessen Auftreten deutlich kritischer ist. Dass dieser so oft vorkommt ist erstaunlich. Speziell bei Erweiterungen kann dieser Smell gefährlich werden, da dann zusätzliche Zugriffe geschehen, welche Data Races verursachen können.

Auch Timeouts, Fire and Forget und Waiting Conditions in Tasks treten vermehrt auf, Smells, die ebenfalls unbedingt Aufmerksamkeit seitens des Entwicklers verlangen.

Korrelation zwischen Codezeilen und Anzahl Smells

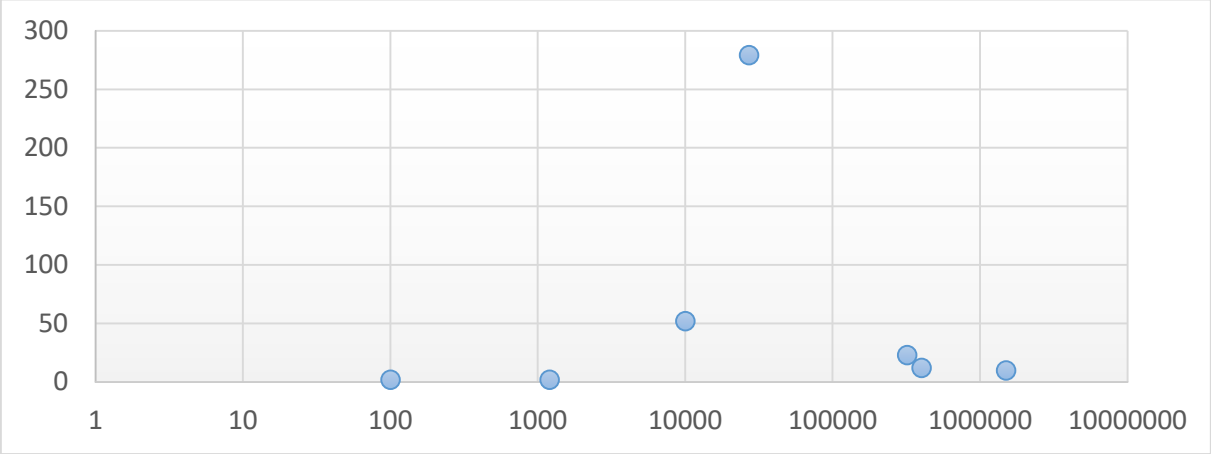


Abbildung 15 Korrelation zwischen Codezeilen und Anzahl Smells

Diese Grafik sollte den Zusammenhang zwischen Anzahl Zeilen Code (X-Achse) und der Anzahl gefundener Smells (Y-Achse) darstellen. Es ist schnell ersichtlich, dass dieser Zusammenhang nicht linear ist, da ausser einem Ausreisser, der von SignalR herrührt, eine verhältnismässig konstant kleine Anzahl Code Smells gefunden wurde. So halten sich diese im Bereich von 2 bis 50 auf. Einen viel grösseren Einfluss als die Grösse scheint also die Qualität der untersuchten Codebasis zu haben.

Nicht gefundene Smells

Die nachfolgenden Smells wurden in keinem der analysierten Projekten gefunden.

Code Smell	Beschreibung
FS001	Finalizer
MWS001	Wait nicht in einem While oder do While Block
MWS002	Verwendung von Pulse anstatt PulseAll
NSMC001	Geschachteltes Locking der gleichen Klasse
NSMC002	Überprüfung lineare Sperrordnung

Tabelle 10 Übersicht nicht gefundener Smells

FS001

Dieser Smell wurde zu Beginn noch oft gefunden, doch durch die fortlaufende Verbesserung des ParaSmellers konnten viele False Positivs eliminiert werden. So müssen nun mehrere Kriterien wie etwa die Deklaration des Feldes als static und readonly, zutreffen damit dieser Smell überhaupt gefunden werden kann. In den untersuchten Projekten wurden also keine gefährlichen Zugriffe auf ein Member innerhalb eines Destruktors gemacht.

MWS001 / MWS002

Diese Checks hätten Warnung angezeigt, falls der Monitor Lock nicht korrekt verwendet worden wäre. Dies Einerseits falls Monitor.Wait nicht innerhalb einer While Schleife benützt würde, andererseits falls Pulse anstatt PulseAll verwendet worden wäre. Doch wurden in diesen Projekten Monitor Locks sehr selten benötigt und daher konnte auch kein Auftreten dieser Smells gefunden werden,

NSMC001 / NSMC002

Dieser Reporter ist sehr spezifisch programmiert und wird sehr wahrscheinlich nicht in freier Wildbahn gefunden. Der erste Check versucht zu erkennen, ob innerhalb eines Lockblockes eine Methode eines anderen Objekts derselben Klasse aufgerufen wird. Wird in dieser Methode ebenfalls ein Lock verwendet, kann es zu einem Deadlock kommen. Zum einen treten solche Verschachtelungen bei einer guten Architektur nicht auf, zum anderen erkennt der Checker wohl auch noch viele Spezialfälle nicht.

Der zweite Check ist ebenfalls sehr spezifisch und nicht unbedingt geläufig. Dieser prüft, ob die Locks in linearen Sperrordnung bezogen werden. Dies ist aber nur möglich, falls dies bereits zur Compile Zeit feststeht. Werden die Locks dynamisch aus einer Liste gelesen, wie im Beispiel der Dining Philosophers, ist es nicht möglich, herauszufinden, ob es zu einem Deadlock kommen kann. Somit ist Einsatz des Checkers in jetziger Form in diesem Bereich auch nur eingeschränkt möglich.

Fazit

Trotz dessen, dass nicht alle Code Smells gefunden worden sind, sind doch kritische Stellen in den untersuchten Projekten gefunden worden. Nicht alle sind gleich gefährlich und einige Code Smells lassen sich eher als Stilfragen betrachten. Nichts desto trotz ist ParaSmeller ein gutes Werkzeug für die Entwicklung grosser Projekten mit asynchronen Aspekten. Die schnelle Analyse zeigt Warnungen zeitnah an und erlaubt so, mögliche Fehler gleich zu verhindern. Sicherlich bedarf es noch Erweiterungen der Checker selber, damit weitere Sonderfälle erkannt werden und aber auch noch weniger False Positives angezeigt werden. Auch die Verteilung innerhalb bestehender Projekte ist sehr unterschiedlich und ist nicht von der Anzahl Zeilen Code abhängig. Einerseits hängt es damit zusammen, um was es für ein Projekt handelt. Falls das meiste synchron abläuft, ist es offensichtlich, dass nicht gleich viele Smells gefunden werden. Bei Performance kritischen Libraries werden viele Warnungen im Bereich «Primitive Synchronisierungsmechanismen» gefunden werden, da diese dort auch erforderlich sind. So ist es nötig, für jeden Fall abzuwägen, wie schwer eine Warnung ist und ob sie behoben werden soll. Auch in ParaSmeller selber gibt es zwei Warnungen, die von primitiven Synchronisierungsmechanismen herrühren, die im Kontext aber kein Problem darstellen. Nachfolgend eine Übersicht der Auftrittshäufigkeiten der Smells und den Stand ihrer Erkennung im Prototypen.

Smell	%	Weiterfolgen?
HSC – Halb-synchronisierte Klasse	16%	Dieser Smell wird bereits relativ gut erkannt, trotz dessen sind noch Optimierungen denkbar. Beispielsweise könnte die Analyse durch Vererbungshierarchie hindurch gemacht werden.
FaF – Fire and Forget	5%	Wird bereits gut erkannt und kann so belassen werden.
MWS – Einfaches Monitor Wait oder Signal	0%	Trotz dessen, dass dieser Smell nicht gefunden wurde, wäre es empfehlenswert, diesen zu optimieren, um möglichst viele Fälle zu erkennen, da dieser Smell zu ernsthaften Symptomen führen kann
ET – Explizite Threads	20%	Threads werden bereits markiert. Hier könnte man die White List erweitern, sodass weniger False positives auftreten.
WCT – Warteabhängigen unter Tasks	2%	Ist sehr schwer zu detektieren, es ist aber auch sehr wertvoll, wenn solche gefunden werden. Sollte daher unbedingt weiterverfolgt und die Suchlogik verbessert werden.
NSMC – Geschachtelte synchronisierte Methodenaufrufe	0%	Das Erkennen von solchen Gerüsten wäre wünschenswärt und würde erkannt, falls es zur Compile Zeit geschieht. Hier müsste man für alle Fälle den dynamischen Flow analysieren, was über statische Code Analyse hinausgeht.
OA – Über-Asynchronität	15%	Dieser Smell ist schwierig zu gewichten, da er eher subjektiv ist. Es ist eine Stilfrage, bis zu welcher tiefe «asyncs» verwendet werden sollen. Daher ist das Weiterverfolgen dieses Smells schwierig.
PS – Atomic, Volatile und Yield	31%	Die Erkennung dieses Smells geschieht bereits sehr gut und kann so übernommen werden.
FS – Finalizer	0%	Sicherlich ein wichtiger Smell, welcher weiterverfolgt werden soll, da auch dieser Smell ernsthafte Symptome mit sich bringen kann.
TRR – Versuchsweiser Ressourcenbezug	10%	Die meisten gefundenen Stellen sind nicht direkt gefährlich. Hier müsste die Logik weiter ausgebaut werden, um False Positives zu vermeiden.

Tabelle 11 Auswertung der Smells

Schlussfolgerungen

Wie die Analyse der Open Source Projekte gezeigt hat, ist es durchaus möglich, Code Smells im Bereich der parallelen Programmierung zu erkennen. In den einzelnen Projekten konnten so einige kritische Stellen identifiziert werden, welche man mit dem Wissen darum einfach korrigieren könnte. Mit Hilfe der Ergebnisse konnte eine Klassifizierung der Smells gemacht werden, welche Aufzeigen soll, welche Smells in der Praxis tatsächlich auftreten. Zusätzlich wird auch zwischen Smells unterschieden, welche wirkliche Probleme in einer Implementation darstellen und solchen, welche eher in den Bereich der Stilfrage gehören. Diese Liste stellt eine Empfehlung der Autoren dar, welche Smells ein Checker in wie weit beachten soll, der produktiv eingesetzt wird.

Smell	Erkennung sinnvoll	Bemerkungen
HSC – Halb-synchronisierte Klasse	Ja	Die Erkennung der halbsynchronisierten Klassen funktioniert sehr zuverlässig und sie sollten auch unbedingt vermieden werden. Zudem hat sich gezeigt, dass dieser Smell häufig auftritt.
FaF – Fire and Forget	Ja	Die Erkennung funktioniert bereits zuverlässig und diese Stellen sollten unbedingt noch genauer durch einen Entwickler betrachtet werden.
MWS – Einfaches Monitor Wait oder Signal	Nein	Dadurch, dass nie ein solcher Smell gefunden wurde, ist es schwierig abzuschätzen, ob die Verfolgung dieses Smells sinnvoll ist. In der Praxis scheint dieser Smell zumindest in der Grundform nicht häufig auftreten, daher hat er für einen Checker im produktiven Umfeld nicht die höchste Priorität.
ET – Explizite Threads	Ja	Dadurch, dass seit einigen Jahren Tasks anstatt Threads verwendet werden können, sollten diese auch eingesetzt werden. Zudem ist die Erkennung einfach zu Implementieren und deshalb empfehlenswert.
WCT – Warteabhängigen unter Tasks	Ja	Die Erkennung ist noch am Anfang, trotzdem lohnt sich der Einsatz, wie sich beim Philosophen Problem gezeigt hat.
NSMC – Geschachtelte synchronisierte Methodenaufrufe	Nein	Dadurch, dass nie ein solcher Smell gefunden wurde, ist es schwierig abzuschätzen ob dieser sinnvoll ist. Die Erkennung von allen Fällen ist sehr komplex bis unmöglich, dafür ist die Behebung dieses Smells jedoch sehr wichtig in einem Projekt. Sollte aufgrund der Komplexität und des seltenen Auftretens jedoch nicht in die erste Version eines produktiven Checkers einfliessen.
OA – Über-Asynchronität	Teilweise	Hier muss man sich überlegen, ob die Erkennung wirklich einen Mehrwert bringt. Es ist schlussendlich auch eine Stilfrage, in einem produktiven Checker sollte die Anzeige dieses Smells zumindest konfigurierbar sein.
PS – Atomic, Volatile und Yield	Teilweise	Die Erkennung dieses Smells kann helfen, typische Business Software auf der richtigen Abstraktionsstufe zu implementieren, zudem ist die Erkennung sehr einfach. Jedoch sollte auch die Anzeige dieses Smells konfigurierbar sein, damit man beim Arbeiten in entsprechenden Domänen nicht von Warnungen überschwemmt wird.

FS – Finalizer	Nein	In der Praxis scheint dieser Smell nicht häufig aufzutreten, damit hat die Erkennung dieses Smells in einem produktiven Checker nicht die grösste Priorität.
TRR – Versuchsweiser Ressourcenbezug	Teilweise	Die meisten gefundenen Stellen, waren False Positives. Daher ist die Erkennung nur teilweise sinnvoll und müsste noch optimiert werden.

Tabelle 12 Sinnvolle Erkennung der einzelnen Smells

Wie bereits erwähnt, wurde bei diesem Prototyp eher in die Breite als in die Tiefe gearbeitet, so dass alle 10 Smells abgedeckt werden konnten. Es ist daher nicht auszuschliessen, dass wenn einzelne Smellerkennungen tiefergehend implementiert werden würden, es zu einer Verschiebung der Auftrittshäufigkeiten kommen könnte. Zudem kann eine Analyse dieser Smells nie abschliessend sein, da sie beliebig viele Sonderfälle abhandeln könnte und man zudem schnell an das Halte-Problem gelangt. Für eine gute Aufwand-Ertrag Balance ist es daher nicht förderlich, jedes Auftreten eines Smells erkennen zu wollen.

Bereits durch diesen Prototypen konnten jedoch schon viele Smells erkannt werden. Dadurch kann gesagt werden, dass der Einsatz von Tools, die Smells erkennen, das Programmieren nebenläufiger Anwendungen sicherer und angenehmer macht. Das Evaluieren bestehender Projekte ist sehr einfach möglich und bietet klares Feedback für die Entwickler. Es ist weiterhin anzunehmen, dass es bei den untersuchten Projekten in der Entstehungsgeschichte weitere Probleme im parallelen Bereich gegeben hat, die durch Techniken wie Code Reviews, Testing oder Nutzerfeedback eliminiert werden konnten. Durch den Einsatz eines statischen Analysetools während der Entwicklung könnte ein Teil dieser Probleme bereits früher, und damit billiger, erkannt und behoben werden.

Damit kommt diese Arbeit zum Fazit, dass die Entwicklung und Anwendung von statischen Analysetools für Code Smells im Bereich der parallelen Programmierung verfolgt werden soll.

Anhang A - Implementationsdetails

A-1 Diagramme

In diesem Kapitel werden alle Diagramme, die im technischen Bereich in vereinfachter Form abgebildet wurden, in ihrer vollen Form dargestellt sowie detailliert erklärt.

Domain Modell

Bei allen Elementen wird die Implementation als Syntaxbaum gespeichert. Dies ermöglicht sehr einfach tiefere Einsicht in die Struktur. Zudem werden zusätzliche benötigte Informationen auf den jeweiligen Elementen gespeichert, so werden bei den Property-Repräsentationen etwa die allfälligen Getter und Setter gespeichert, bei den Methoden die Parameter. Wo immer möglich, wurden dazu Strukturen benutzt, die Roslyn bereits anbietet, so werden etwa die Namen der Elemente als SyntaxToken gespeichert, damit bleiben die Repräsentationen relativ schlank und benötigen keine zusätzliche selbst geschriebene Informationshalter.

Dadurch, dass Member sowohl in Interfaces als auch in Klassen deklariert werden können, ergibt sich die etwas unschöne Modellierung, dass ein Member beide Elternbeziehungen hat, jedoch nur eine davon gesetzt ist. Um dies zu umgehen, müssten Interfacerepräsentation und Klassenrepräsentation unter einer gemeinsamen Oberklasse verwaltet werden. Diese Idee wurde jedoch verworfen, da sonst auf Solutionebene der Umgang mit den Kinderelementen sehr mühsam geworden wäre.

Der grösste Mehrwert dieser Modellierung ist, dass die Methoden- und Property-Aufrufe in den Memberblöcken jeweils genau auf die jeweilige Implementation zeigen. Ist diese Information wegen Polymorphie in der Roslyn-API nicht eindeutig, so wird auf eine Liste von möglichen Implementation verwiesen. Dies ermöglicht in der Analyse eine einfache Iteration durch den Kontrollfluss, was für viele Fälle wie das Auflösen von Wartebedingungen unerlässlich ist.

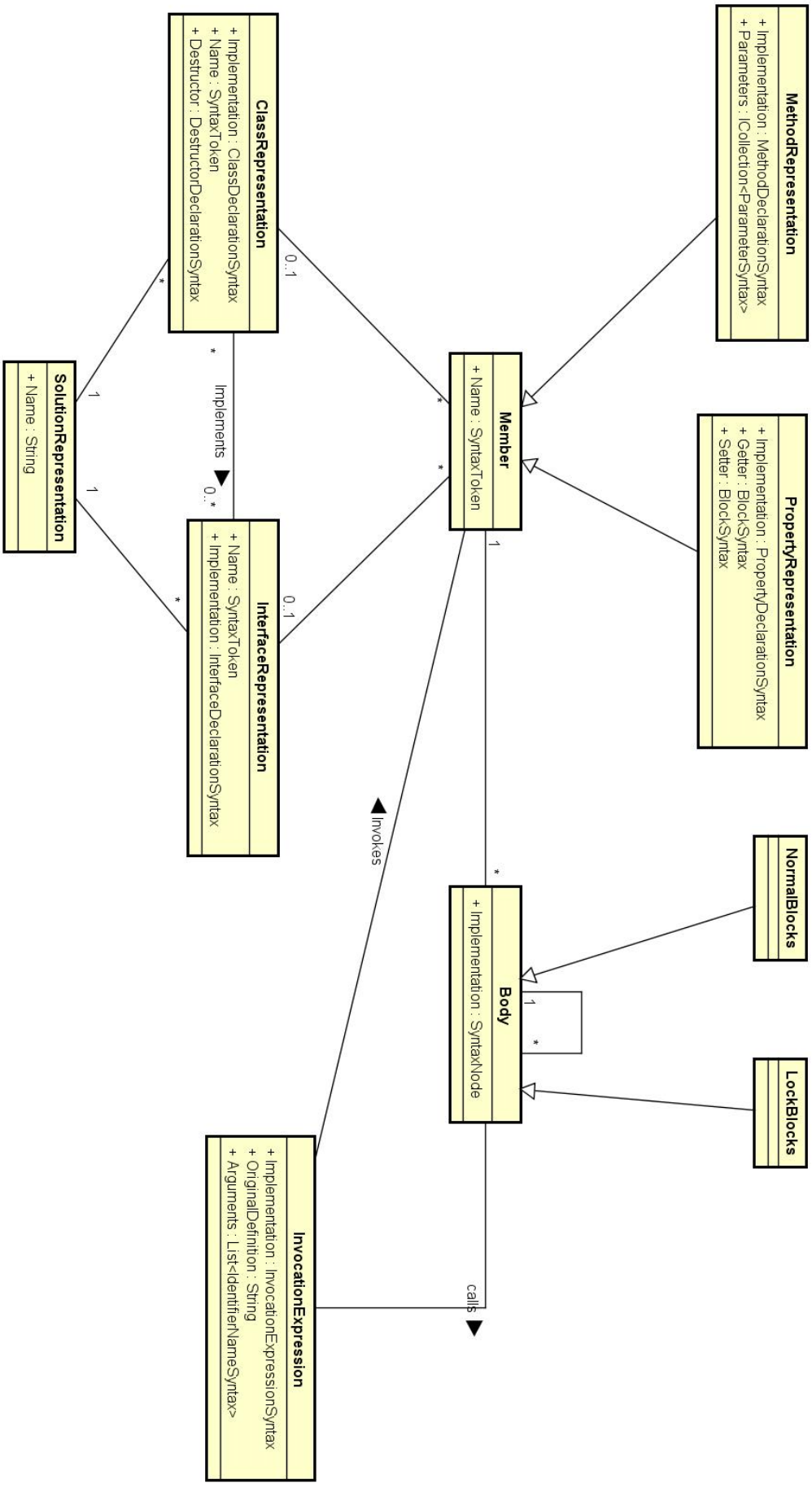


Abbildung 16 Domain Modell Detail Ansicht

Reporting Framework

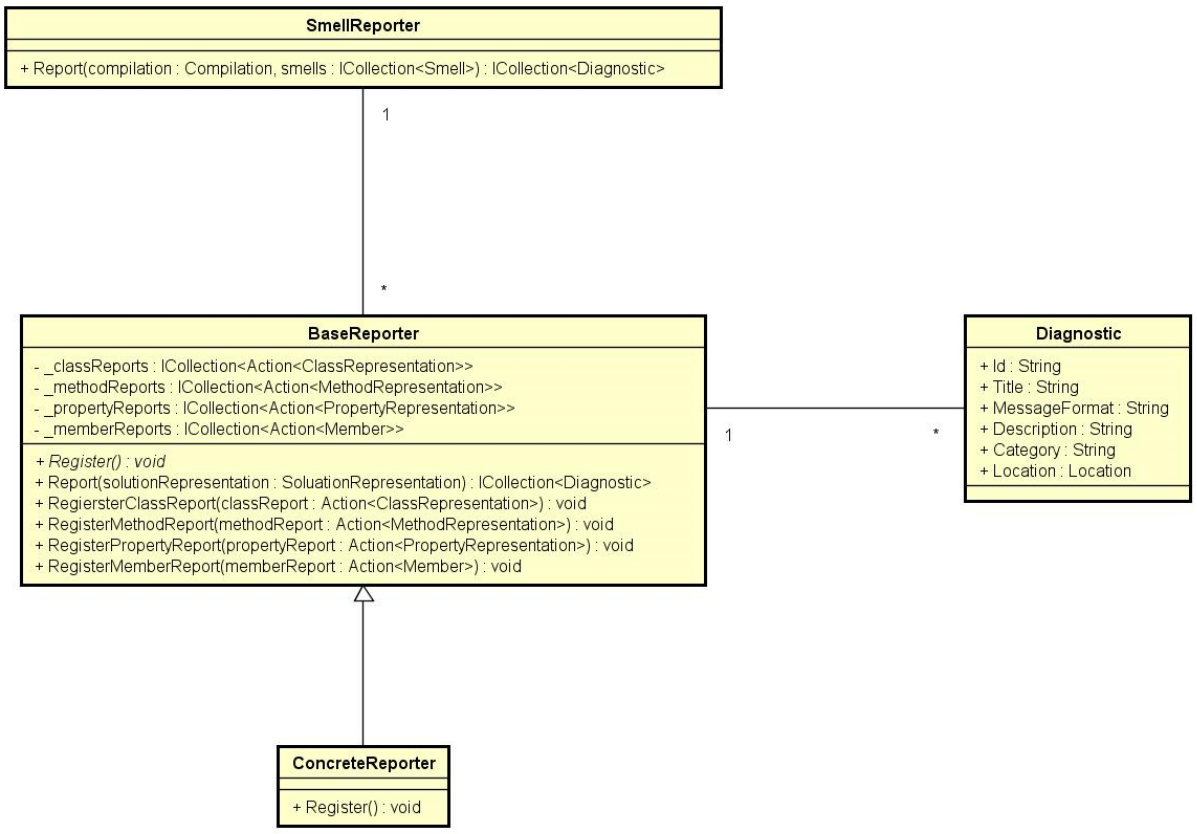


Abbildung 17 Reporting Framework Detail Ansicht

Die einzigen zwei Schnittstellen der Library nach aussen sind die Report-Funktion des SmellReporters und die Diagnoseklasse. Eine Diagnose ist ein einfacher Container für einen gefundenen Smell. Er beinhaltet die Fundstelle im Code, sowie weitere Metadaten wie eine ID, Titel und Schwerekategorie. So bietet eine Diagnose eine einfache Form an, um den gemeldeten Smell weiter zu verarbeiten, auf der Konsole wird eine gut lesbare Form ausgegeben, Tools wie Visual Studio können diese Informationen abgreifen und ohne grossen Aufwand in ihr eigenes Diagnoseformat überführen

Der SmellReporter nimmt eine Compilation in einer .Net Sprache sowie eine Liste zu beachtender Smells entgegen, in den Default Einstellungen untersucht er dabei auf alle implementierten Smells. Als Rückgabewert fungiert eine Liste von gefundenen Diagnosen, die natürlich auch leer sein kann. Im ersten Schritt bringt der SmellReporter die Compilation durch Aufruf von Factory-Funktionen in eine leicht zu untersuchende Form (siehe Abbildung 1). Im zweiten Schritt wird die Analyse gestartet.

Der SmellReporter hält dabei eine Liste von Reportern, welche alle auf einen bestimmten Smell spezialisiert sind und die Logik für dessen Erkennung beinhalten. Diese erben dabei alle von einem BaseReporter, die bereits alle Logik enthält, die für das Weiterleiten von Diagnosen an den SmellReporter von Nöten ist. Der BaseReporter bietet zudem Registrierfunktionen für Analysen auf verschiedenen Stufen wie etwa Klassen- oder Memberebenen an. Ein konkreter Reporter muss also nur Actions implementieren, die die gewünschte Abstraktionsstufe als Parameter erhalten und Logik enthalten, um daraus Diagnosen zu generieren. Indem der konkrete Reporter die Register-Methode des BaseReporter implementiert und dort alle Actions registriert teilt er mit, welche Untersuchungen er ausführen will. Durch Anwendung des Template-Method-Patterns in der Report-Funktion führt der BaseReporter dann alle Analysen

durch und meldet die gefundenen Diagnosen dem SmellReporter, der die Diagnosen aller konkreter Reporter sammelt und schlussendlich nach aussen weitergibt.

Dies ermöglicht sehr einfach die Implementation für Erkennungen weiterer Smells, es ist lediglich eine neue Klasse zu schreiben, die von BaseReporter erbt und die Logik für die Erkennung enthält, welche dann in der Register-Methode registriert werden muss.

Visual Studio Integration

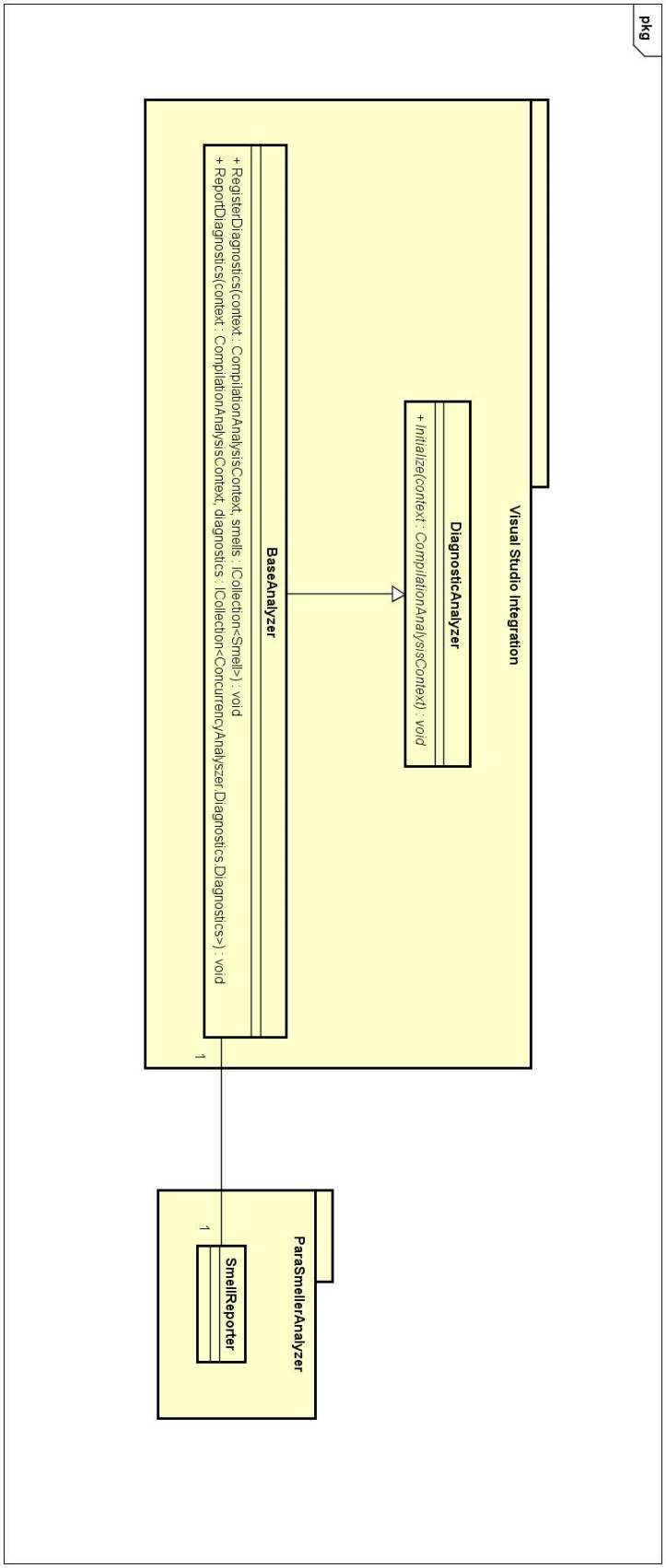


Abbildung 18 Visual Studio Integration Detail Ansicht

Die Integration in Visual Studio gestaltet sich sehr einfach, da es bereits eine gute Infrastruktur gibt, um eigene Analyzer zu schreiben. Dabei erweitert man einfach die Klasse DiagnosticAnalyzer. In der zu implementierenden Funktion Initialize erhält man dabei den Kontext der Kompilation, von wo aus man Zugriff auf den Syntaxbaum hat und auf dem man durch eine einfache Schnittstelle Diagnosen melden kann.

In der Integration von ParaSmeller erbt dazu ein schlanker BaseAnalyzer von DiagnosticAnalyzer, und leitet die ganze Arbeit durch Delegation an den Kern von ParaSmeller weiter, wobei als Schnittstelle der SmellReporter dient (siehe Abbildung 2). Alleine die von ParaSmeller zurückgegeben Diagnosestrukturen werden noch in die Visual Studio eigene Diagnoseform umgewandelt und auf dem Kontext gemeldet. Um die Analyse einfacher konfigurierbar zu machen, wird dabei noch eine Funktion implementiert, die es erlaubt, nur eine Auswahl von zu untersuchenden Smells auszuwählen.

Dazu ruft die Initialize-Methode des Visual Studio Analyzer-Framework zuerst die Methode RegisterDiagnostics des BaseAnalyzers auf, worin die zu untersuchenden Smells festgelegt werden, die daraufhin wiederum die Methode ReportDiagnostics aufruft, welche schliesslich die Analyse durchführt und die gefundenen Diagnosen dem Kontext meldet.

In der Verwendung des BaseAnalyzer wird standardmässig auf alle Smells untersucht, doch durch die Erweiterung der BaseAnalyzer Klasse könnte man verschiedene Analyzer-Profile anlegen, die jeweils auf eine andere Smell-Auswahl untersuchen.

Testing

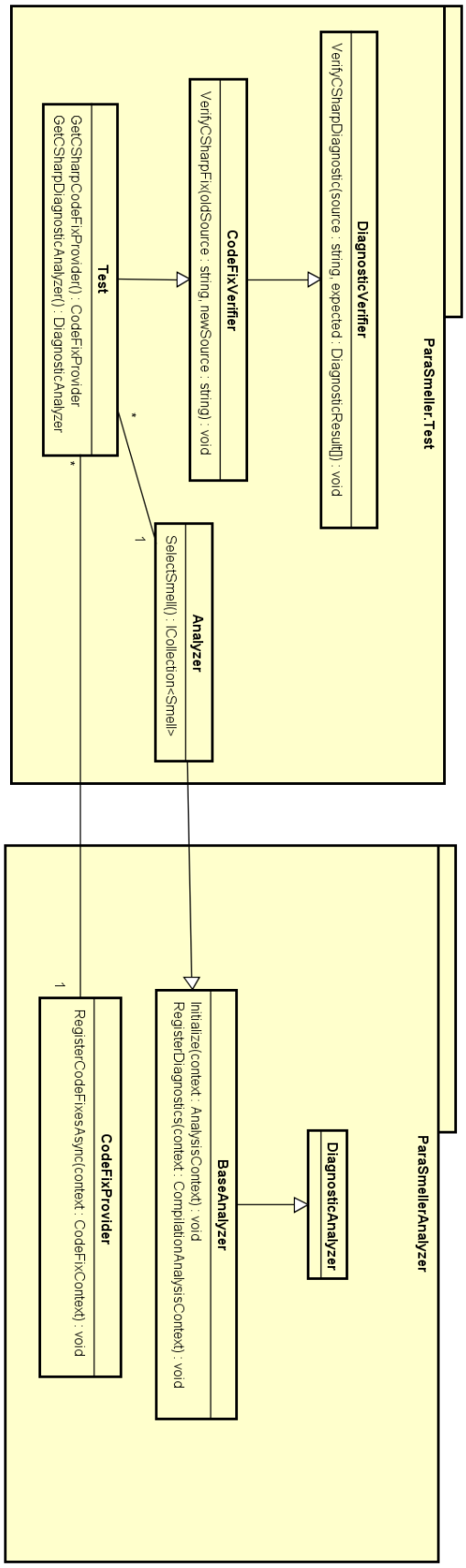


Abbildung 19 Testing Detail Ansicht

Um das Rad für die Tests nicht neu zu erfinden, wurde auf das Testing-Framework für Analyzer von Visual Studio zurückgegriffen. Dadurch, dass die Integration in Visual Studio sehr schlank und nicht fehleranfällig ist (siehe Abbildung 3), kann trotzdem sichergestellt werden, dass alle Tests direkt die Funktionalität von ParaSmeller testen.

Für das Testen von Analyzern hat Visual Studio vorgesehen, dass man die Klasse CodeFixVerifier erweitert. In den Testmethoden kann man dann Source-Code in String-Form sowie eine Liste von zu findenden Diagnosen übergeben und das Framework meldet, ob auch alle Diagnosen korrekt gefunden werden. Dazu muss auch noch die Methode GetDiagnosticAnalyzer implementiert werden, die den jeweils zu verwendenden Analyzer zurückgibt.

Um sicher zu stellen, dass alle Smells unabhängig voneinander diagnostiziert werden, wurden für alle Smells eigene Testklassen geschrieben, die in der GetDiagnosticAnalyzer jeweils eine Spezialisierung des Unter Kapitels Visual Studio Integration beschriebenen BaseAnalyzer zurückgeben, welche nur auf den bestimmten Smell Typ untersucht.

Auch Code Fixes lassen sich testen, indem man zwei Code-Stücke für den Vorher-Nachher-Vergleich angibt sowie die Methode GetCodeFixProvider überschreibt, welche dann eine Implementation zurückgibt, die den entsprechenden Code Fix durchführt. Diese Arbeit wird direkt an entsprechende Implementation delegiert.

Doch durch die Limitierung, dass standardmässig einfach an der ersten Stelle, an welcher eine Warnung gefunden wurde, der CodeFix Provider ausgeführt wird, musste das Framework ein wenig angepasst werden. Dies war nötig, da zum Beispiel bei dem «Halfsynchronized Class Checker» die Warnungen immer paarweise angezeigt werden und beide Fälle getestet werden müssen. So wurde die Erweiterung implementiert, dass spezifiziert werden konnte, bei welchem Code Smell der Code Fix Provider angewendet werden sollte. Ansonsten kann gut der bestehende Code verwendet werden und so speziell bei Refactorings die Sicherheit durch viele Unit Tests massiv erhöht werden.

Gegen Ende, als die komplexeren Verbindungen zwischen Klassen und Interfaces analysiert und Beziehungen zwischen Funktionsaufrufe erstellt wurden, stiess man aber an die Limitierung des Testing Frameworks. So kann man nicht direkt eine Solution analysieren, sondern wie vorher auch musste die «Test-Klasse» in Form eines Strings angegeben werden. Dies ist unpraktisch und sollte das Projekt in Zukunft weiterwachsen, wird wohl das Schreiben eines eigenen Testing Framework unausweichlich.

A-2 Code Smells

Halb-synchronisierte Klasse

Klasse	HalfSynchronizedReporter
Erkennung	**HSC001** Ein Member benutzt ein anderes Member und synchronisiert den Zugriff darauf nicht. Eine drittes Member greift ebenfalls auf jenes Member zu, synchronisiert den Zugriff aber. Das heisst, der Zugriff im ersten Member sollte wohl ebenfalls synchronisiert werden. **HSC002** Eine Member synchronisiert den Zugriff auf ein Property. Das Property selber ist aber standardmässig mit Settern und Gettern ausgestattet oder synchronisiert den Zugriff auf die Backingvariable nicht. Es sollte gegebenenfalls eine Backingvariable eingeführt werden und der Zugriff synchronisiert werden.
Refactoring	**HSC001**: Der Memberzugriff wird in einen lock-Block eingeschlossen. **HSC002**: Falls keine Backingvariable eingesetzt wird, wird eine erschaffen. Die Accessors werden nun so umgeschrieben, dass der Zugriff auf die Backingvariable synchronisiert wird.

Detailerkennung

Allgemein ist zu sagen, dass bei diesem Smell nur die Synchronisation durch einen Monitor-Lock beachtet wird. Eine naheliegende Verbesserung des Checkers wäre es, auch andere Synchronisierungsmechanismen in Betracht zu ziehen. Auch werden die Lockobjekte bis jetzt nur oberflächlich verglichen, sie gelten als gleich, wenn sie beide das this-Objekt sind oder wenn sie das gleiche Feld der Klasse ohne Aliase referenzieren. Hier könnte durch eine genauere Untersuchung der eingesetzten Lockobjekte eine bessere Analyse geliefert werden.

HSC001

Falls das Member bereits vollständig synchronisiert ist, muss nichts unternommen werden. Auch falls das Member auf keine Properties zugreift, besteht kein Problem. Sonst werden in einem ersten Schritt alle nicht synchronisierte Properties (im Folgenden als A bezeichnet) der Klasse gesammelt, in welcher das Member definiert ist.

Dann werden alle Member der Klasse gesammelt, welche zumindest zum Teil synchronisiert sind. Daraus werden dann die in Lock-Statements eingesetzten Properties der eigenen Klasse gefiltert (im Folgenden als B bezeichnet).

Im dritten Schritt werden alle in der untersuchten Member verwendeten Properties gesammelt (im Folgenden als C bezeichnet).

Schlussendlich wird die Schnittmenge von A, B und C gebildet. Ist diese nicht leer, heisst dies, dass der Zugriff auf ein Property in einigen Fällen synchronisiert wird, im untersuchten Member aber nicht.

In diesem Falle wird eine Diagnose herausgegeben.

HSC002

Falls das zu untersuchende Property bereits synchronisiert auf eine Backingvariable zugreift, muss nichts unternommen werden.

Ansonsten wird untersucht, ob das Property in einem synchronisierten Kontext verwendet wird. Falls dies der Fall ist, sollte besser das Property selber synchronisiert werden.

Dazu werden alle Properties gesammelt, welche in einem synchronisierten Block innerhalb der Klasse verwendet werden. Gibt es ein Matching, ist die Wahrscheinlichkeit gross, dass ebenfalls der Zugriff auf das Property synchronisiert werden sollte.

In diesem Falle wird eine Diagnose herausgegeben.

Detail Refactoring

HSC001

Im einfachsten Falle wird einfach der ganze Memberbody mit einem Lockstatement umgeben und man ist fertig. Dabei wird zum jetzigen Stand jenes Lockobjekt genommen, dass am meisten in der Klasse verwendet wird. Dabei werden private Felder in Betracht gezogen, komplexe Referenzen von ausserhalb im jetzigen Stand nicht. Das Default-Lockobjekt stellt dabei immer das this-Objekt dar.

Es gibt jedoch eine sehr sinnvolle Verbesserung des Refactorings. In den meisten Fällen dürfte das Locken des gesamten Bodys viel zu grobgranular sein. Daher sollte nur der Zugriff auf das Property selbst synchronisiert werden. Falls dies jedoch mehrmals in der Member auftritt, und jeweils dasselbe Lockobjekt verwendet wird, wäre es sinnvoller, über den entsprechenden Bereich eine Synchronisation zu legen. Hier ist eine ansprechende Balance zwischen fein- und grobgranularem Locking zu finden.

HSC002

Falls das Property nur über Standard Getter / Setter verfügt oder ein Expression Body benutzt, ist der Fall ziemlich einfach, es muss lediglich eine private Backingvariable vom gleichen Typ in der Klasse eingeführt werden. Die Getter / Setter greifen nun synchronisiert auf diese zu.

Komplizierter wird es, wenn bereits eine Backingvariable verwendet wird. Hier stellen sich dann dieselben Überlegungen zu grob versus feingranularem Locking wie bei HSC001. Auch muss die entsprechende Backingvariable identifiziert werden.

Noch komplexer ist der Fall, wenn keine Backingvariable eingesetzt wird, sondern der Rückgabewert aus anderen Methoden und Properties zusammengesetzt wird. In diesem Falle ist noch eine gute Lösung zu finden.

Fire and Forget

Klasse	FireAndForgetReporter
Erkennung	**FaF001** Ein Task wird gestartet, aber es wird nicht auf dessen Ende gewartet. Wahrscheinlich ist dies sowieso ein Design Fehler, speziell könnte der Main Thread beendet sein, bevor der Task fertig ist.

Detailerkennung

FaF001

Der einfachste Fall stellt der Fall dar, in dem nur eine ein Thread Aufruf in einem Statement passiert in der Form von

Task.Run(()=>DoStuff());.

Hier kann sofort eine Warnung herausgegeben werden, natürlich ausser das Ende wird auch gleich abgewartet, etwa in der Form

Task.Run(() => DoStuff()).Wait();.

Schwieriger ist der Fall, wenn das Ganze in eine Variable gespeichert wird, etwa so:

var x = Task.Run(() => DoStuff());

Hier muss der Flow des Programms analysiert werden. Es könnte nämlich sein, dass Wait() irgendwann auf der Variable ausgeführt wird. Im einfachsten Fall findet dies noch im gleichen Body statt, doch theoretisch kann die Variable durch beliebig viele Methodenaufrufe in beliebig viele Klassen weitergereicht werden.

Dank der AnalyzerLibrary können diese Aufrufketten jedoch einfach verfolgt werden. Findet in der ganzen Kette kein Warten statt, wird eine Warnung ausgegeben, ansonsten nicht, so können relativ verlässlich False Positives vermieden werden.

Einfaches Monitor Wait oder Signal

Klasse	**MonitorOrWaitSignalReporter**
Erkennung	**MWS001** In einem Monitor wird Wait verwendet, welches sich aber nicht in einer Schleife befindet **MWS002** Pulse wird anstelle von PulseAll verwendet
Refactoring	**MWS001** Ändern von if nach while **MWS002** Ändern von Pulse nach PulseAll

Detailerkennung

MWS001

Die Erkennung erfolgt relativ simpel. Bei jedem Monitor.Wait wird geprüft, ob der nächst höhere Block entweder ein while oder ein do while ist. Ist dies nicht der Fall, wird eine Warnung angezeigt. Es wird immer nur direkt der nächst höhere Block geprüft. Komplexe Verschachtelungen werden daher als Warnung markiert. Es ist nicht entscheidbar, welches genau die Wartebedingung für den Wait Aufruf ist.

MWS002

Hier ist es ähnlich wie beim MWS001. Es ist leider nicht wirklich entscheidbar, ob ein Pulse oder ein PulseAll verwendet werden muss. Einzig wenn nur ein Wait verwendet wird, geht der Checker davon aus, dass ein Pulse reicht, da hoffentlich alle wartenden Threads auf die gleiche Bedingung warten. Leider ist dies auch nicht wirklich sicher.

Detail Refactoring

MWS001

Falls sich das Wait in einem If Block befindet, kann dieses durch ein while Block ersetzt werden.

MWS002

Dieses Refactoring erlaubt es Pulse durch PulseAll zu ersetzen.

Explizite Threads

Klasse	ExplicitThreadsReporter
Erkennung	**ET001** Gebrauch der Thread[6] Klasse

Detailerkennung

Die Thread Klasse selbst ist verbose und sollte besser durch die Klasse Task ersetzt werden. Die Problematik ist aber, dass nicht jeder Thread durch einen Task ersetzt werden kann. Meist ist dies wegen Warteabhängigkeiten innerhalb der Threads. Im Gegensatz zu Threads werden bei Tasks nicht beliebig viele Threads eingesetzt, sondern durch Heuristiken gesteuert. Dabei kann es vorkommen, dass alle Threads auf eine weitere Aktion warten, diese aber nicht erfüllt werden kann, da es genau für den Thread, welche diese Aktion auslösen würde, keinen freien Slot mehr hat.

Die Erkennung von solchen Warteabhängigkeiten ist selbst ein Code Smell «Warteabhängigen unter Tasks» und ist nicht entscheidbar. Daher werden in diesem Checker nur Threads markiert, welche ähnlich wie beim Fire and Forget gestartet werden aber danach keine Zugriffe mehr haben.

ET001

Die Erkennung des ersten Falls ist sehr simpel. Es wird geprüft, ob irgendeine Aktion mit der Klasse Thread geschieht, zum Beispiel ob eine neue Instanz erstellt, eine gestartet oder in einer Schleife darüber iteriert wird. Sämtliche solche Aktionen werden als Warnung angezeigt. Es ist unmöglich zu entscheiden, ob hier der Einsatz von Thread wirklich nötig ist oder ob besser Task eingesetzt werden soll.

Die einzige Ausnahme dieses Checkers sind folgende Felder: «CurrentThread», «CurrentPrincipal» und «CurrentContext». In diesen können beispielsweise die Culture gesetzt werden, welches durchaus legitim ist.

[6] System.Threading.Thread

Warteabhängigen unter Tasks

Klasse	WaitingConditionsTasksReporter
Erkennung	**WCT001** Es wird untersucht, ob in als Tasks gestartete Aufgaben eine Wartebedingung auftritt. Ist dies der Fall, wird eine Warnung ausgegeben

Detailerkennung

WCT001

Die definitive Erkennung von Verklemmungen von Tasks ist sehr schwierig bis unmöglich. Es wird deshalb lediglich geprüft, ob innerhalb von Tasks ablaufenden Funktionen ein oder mehrere Synchronisierungsmechanismen wie Locks eingesetzt werden. Trifft dies zu, wird eine Warnung herausgegeben, denn falls in mehreren Tasks auf die gleiche Bedingung gewartet wird, könnte hier ein Deadlock auftreten.

In einem späteren Schritt könnte die Analyse verfeinert werden, indem untersucht wird, ob festgestellt werden kann, dass die Tasks auf die gleiche Bedingung warten. Aber auch dann wird meist eine Architekturanalyse durch den Entwickler von Nöten sein, um sicher gehen zu können, dass die Warnung legitim ist.

Geschachtelte synchronisierte Methodenaufrufe

Klasse	**NestedSynchronizedMethodClassReporter**
Erkennung	**NSMC001** Member beinhaltet ein Lock und ruft durch ein anderes Member wiederum einen synchronisierten Block auf, dessen Lockobjekt sich in der gleichen Vererbungshierarchie wie dasjenige des ersten Locks befindet. **NSMC002** Erkennt bei geschachtelten Locks innerhalb einer Klasse, ob es Methoden gibt, die die Locks in einer anderen Reihenfolge beziehen, was zu einem Deadlock führen kann.

Detailerkennung

NSMC001

Es wird jeweils durch eine Tiefensuche durch die ganze Aufrufkette eines Members iteriert, dies beliebig tief. Dabei merkt sich der Checker alle Lockobjekte, die er auf diesem Pfad auffindet und prüft bei jedem neu gefundenen, ob dieses sich in der gleichen Vererbungshierarchie wie ein bereits früher gefundenes befindet. In diesem Falle wird eine Warnung ausgegeben.

NSMC002

In jeder Klasse werden von jeder Methode die geschachtelte Lockblöcke gesucht und die Lockargumente je in einer Liste gespeichert. Diese Listen werden anschliessend miteinander verglichen. Dabei wird geprüft ob es allenfalls möglich ist, dass zwei LockObjekte so gelockt werden, dass es zu einem Deadlock kommt, wie zum Beispiel so:

Methode AAA	Methode BBB
lock(A)	lock(B)
lock(B)	lock(A)

Dieses könnte zu einem Deadlock führen. Die Limitation der Erkennung ist, dass Locks nur innerhalb einer Methode betrachtet werden, dafür aber beliebig tief. Ausserdem ist es nicht möglich Lock Objekte, welche erst zur Laufzeit erstellt werden, zu analysieren.

Über-Asynchronität

Klasse	OverAsynchronyReporter
Erkennung	**OA001** Falls eine Methode private und async ist, wird dies als Warnung angezeigt. **OA002** Tiefe Verschachtelung von async Methoden, welche einen Grenzwert überschreiten

Detailerkennung

Bei der Über-Asynchronität geht es im Allgemeinen um das Design und die Verwendung von async Methoden. Theoretisch könnten diese beliebig tief verschachtelt werden. Dies würde aber nicht zwingend bedeuten, dass diese schneller ausgeführt werden, sondern eher langsamer, bedingt durch die vielen Context Wechsel. Somit ist eine Best Practice dass man nur am Anfang die Methoden async definiert, danach der Ablauf aber synchron ist. So kommen auch die beiden Code Smells zur Stande. OA001 versucht sicherzustellen, dass nur public Methoden, welche eine grössere Funktionalität bereitstellt auch asynchron arbeitet und die privaten Methoden dann aber synchron geschehen. OA002 prüft die Tiefe von verschachtelten async Methoden, das Limit wurde auf 3 festgelegt.

OA001

Die Erkennung, ob eine Methode nicht public und async ist, kann leicht über das semantische Modell gemacht werden. Ist dies beides der Fall, wird eine Warnung angezeigt.

OA002

Die zweite Warnung wird angezeigt, wenn ein Entwickler zu viele verschachtelte asynchrone Methoden verwendet. Dies wird geprüft, indem sämtliche Methodenaufrufe aus einer async Methode nachverfolgt werden, bis entweder das Limit von 3 async Methoden erreicht wurde oder es keine weiteren mehr gibt.

Atomic, Volatile und Yield

Klasse	PrimitiveSynchronizationReporter
Erkennung	**PS001** Es wird auf den Gebrauch von primitiven Synchronisierungsmechanismen untersucht. Problemlösungen, welche solche Mechanismen benötigen, werden oft nicht in gemanagten Sprachen geschrieben oder könnten auf bereits bestehende Library Funktion zugreifen.

Detailerkennung

PS001

Die Erkennung gestaltet sich ziemlich einfach, es werden jeweils einfach das Auftreten von bestimmten Keywords gemeldet. Zum jetzigen Stand sind es folgende:

- Alle Verwendungen des volatile-Keywords
- Alle Verwendungen der Interlocked-Klasse
- Jegliche Aufrufe der Methode Yield der Thread Klasse
- Der Einsatz eines Spinlocks, dabei wird die Definition, das Beziehen und das Freigeben des Locks gemeldet
- Der Einsatz der Methode MemoryBarrier der Thread Klasse

Sobald ein oben beschriebenes Element im Source Code gefunden wird, wird eine Diagnose erstellt, die auf primitive Synchronisierungsmechanismen hinweist.

Finalizer

Klasse	FinalizerReporter
Erkennung	**FS001** Wenn im Destruktor auf statische Member zugegriffen wird, ist dies problematisch, da Destruktoren asynchron laufen.

Detailerkennung

FS001

Im ersten Schritt werden alle Zugriffe auf statische Member im Destruktor gesammelt. Nicht statisch definierte sind nicht problematisch, da Destruktoren am Ende des Lebenszyklus einer Instanz aufgerufen werden.

Im weiter Schritt muss untersucht werden, ob die Zugriffe synchronisiert werden. Zum jetzigen Stand wird dabei lediglich der Monitor-Lock Mechanismus in Betracht gezogen. Falls dies nicht der Fall ist, wird eine Warnung ausgegeben.

Sobald ein statisches Member im Destruktor verwendet wird, müssen auch alle anderen Zugriffe darauf in der Klasse und generell in public Membern synchronisiert werden, da der Destruktor ein stets potentiell asynchron laufender Vorgang darstellt. Falls diese Zugriffe nicht synchronisiert werden, wird ebenfalls eine Warnung ausgegeben.

Falls beide Arten von Zugriffen (im Destruktor und alle sonstige) synchronisiert werden, muss in einem weiteren Schritt sichergestellt werden, ob dasselbe Lockobjekt verwendet wird. Des Weiteren muss das Lockobjekt ebenfalls ein privates statisches Member sein, ansonsten nützt der ganze Synchronisierungsvorgang nichts, da instanzenübergreifend gearbeitet werden muss. Trifft einer dieser Fälle nicht zu, wird ebenfalls eine Warnung ausgegeben.

Versuchsweiser Ressourcenbezug

Klasse	TentativelyResourceReferenceReporter
Erkennung	**TRR001** Es wird untersucht, ob Synchronisationsprimitiven mit Hilfe eines Timeouts benutzt werden. Ist dies der Fall und befinden sich diese in einem zusätzlichen loop, wird eine Warnung ausgegeben.

Detailerkennung

TRR001

Die Erkennung gestaltet sich relativ einfach. Es werden sämtliche Aufrufe von Methoden analysiert und folgende Fälle näher untersucht:

- System.Threading.Monitor.Wait
- System.Threading.Monitor.TryEnter
- System.Threading.WaitHandle.WaitOne
- System.Threading.WaitHandle.WaitAll
- System.Threading.WaitHandle.WaitAny
- System.Threading.SpinLock.TryEnter
- System.Threading.Barrier.SignalAndWait

Anschliessend werden die Parameter untersucht. Dies ist flexibler, da so auch neue Overloadings erkannt werden. Befindet sich entweder ein Int32 oder ein System.TimeSpan in der Argumenten Liste, wird noch geprüft, ob sich die Methoden in einer Schleife oder Schleifenkopf befindet. Allenfalls wird eine Warnung angezeigt.

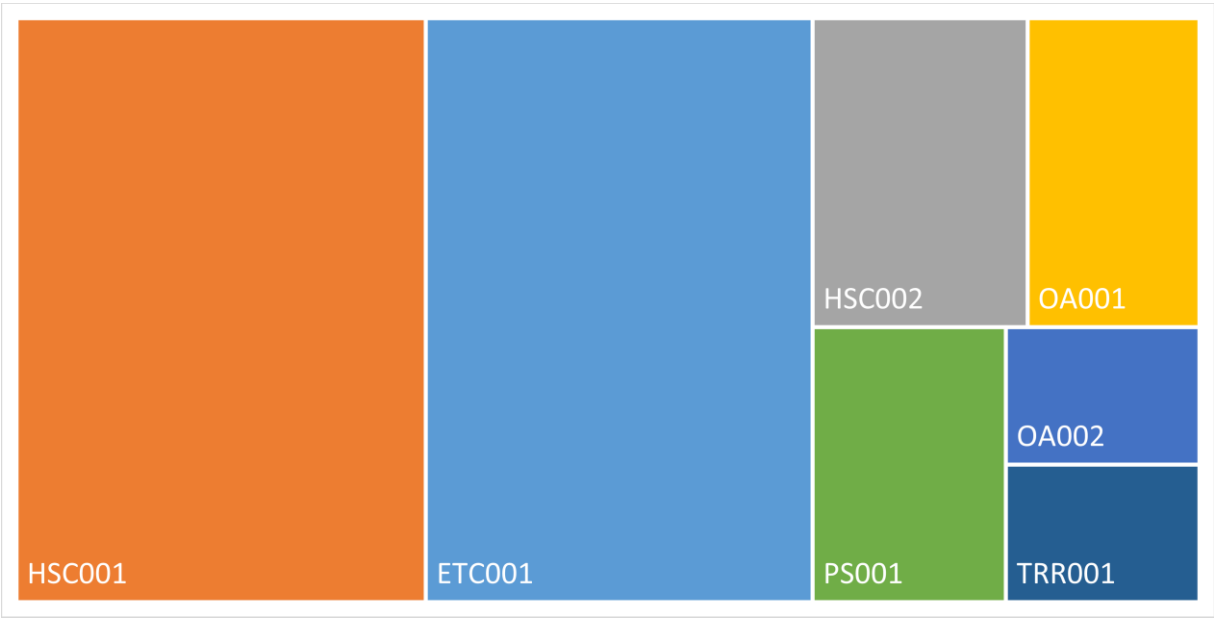
A-3 Ergebnisse

In diesem Kapitel werden detailliert alle Ergebnisse der Analyse von Open Source Projekten dargestellt. Dazu wird von jedem Projekt eine Übersicht von gefundenen Smells dargestellt, woraufhin die einzelnen Smells im Kontext des Projektes besprochen werden.

Log4Net

Grösse	**10'000 LOC**
Quelle	https://github.com/apache/log4net

Übersicht Code Smells



Code Smell	Anzahl	Beschreibung
HSC001	18	Halbsynchronisierte Klasse
HSC002	5	Halbsynchronisierte Klasse
ET001	17	Explicit Threads
OA001	4	Überasynchronität
OA002	2	Überasynchronität
PS001	4	Primitive Sychronisierungsmechanismen
TRR001	2	Timeouts

ET001

Es werden oft explizite Threads vorhanden. Hier wäre Potential vorhanden, auf Tasks zu wechseln.

TRR001

Die Timeout Verwendungen dienen nicht direkt der Synchronisation. In einem Fall wird beim Beenden gewartet, bis alle Messages in der RemoteQueue verarbeitet worden. Beim zweiten Fall wird für eine bestimmte Zeit gewartet, bis alle Meldungen gesendet worden sind.

OA001 / OA002

Die beiden Code Smells bezüglich der Überasynchronität wurden allesamt im Test Projekt gefunden und sind dadurch bedingt, dass diese async definiert sein müssen, falls ein await gebraucht wird.

HSC001 / HSC002

In diesen Fällen wird jeweils jeder Zugriff auf ein Property innerhalb der Klasse zwar gelockt, doch das Property selbst ist als auto-Property implementiert und verfügt über keine Synchronisierung. Dadurch, dass alle Aufrufe gelockt werden, findet noch kein Fehlverhalten statt, doch es könnten sich in Zukunft Fehler einschleichen, indem neue Zugriffe nicht gelockt wären. Sicherer wäre es, des Property selbst zu synchronisieren.

Eine Beispieldeklaration eines solchen Properties findet sich wie folgt:

```
/// The file appender this locking model is attached to and working on
/// behalf of.
/// </para>
/// <para>
/// The file appender is used to locate the security context and the error handler to use.
/// </para>
/// <para>
/// The value of this property will be set before <see cref="OpenFile"/> is
/// called.
/// </para>
/// </remarks>
12 references
public FileAppender CurrentAppender
{
    get { return m_appender; }
    set { m_appender = value; }
}
```

Abbildung 20 Beispiel eines unzureichend synchronisierten Properties

Des Weiteren deuten einige Warnungen darauf hin, dass man feingranularer locken könnte. So werden Aufrufe von Methoden in einer public Methode gelockt, die aufgerufenen Methoden selbst sind aber nicht synchronisiert. So implementiert ist der Code sicher einfacher zu lesen und wohl auch performanter, er birgt jedoch Gefahr, falls einige der Methoden ihre Sichtbarkeit wechseln und man vergisst, die Synchronisation anzupassen.

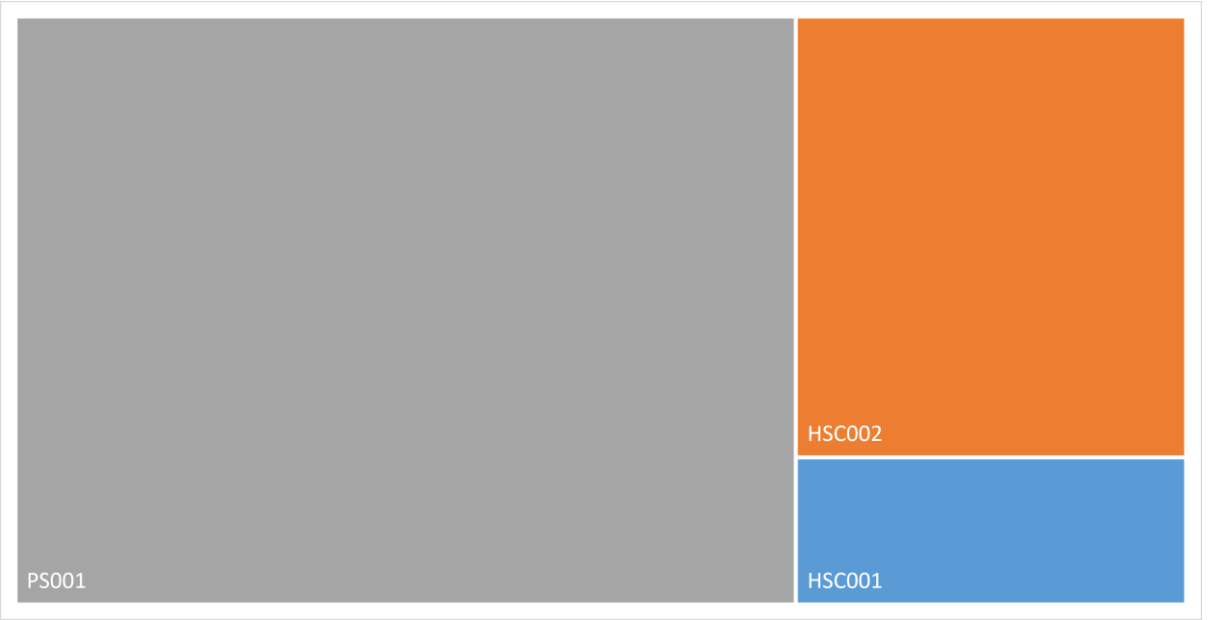
Fazit

Log4Net ist weit verbreitet und wird oft eingesetzt, dem entsprechend sind die meisten Warnungen auch nicht gravierend, beziehungsweise ist deren Verfolgung eine Stilfrage. Es tauchen bei der Analyse jedoch einige wenige Stellen auf, die echtes Gefahrenpotential bergen, welche man mit dem Wissen zudem auch leicht beheben könnte.

NHibernate

Grösse	400'000 LOC
Quelle	https://github.com/nhibernate/nhibernate-core

Übersicht Code Smells



Code Smell	Anzahl	Beschreibung
HSC001	1	Halbsynchronisierte Klasse
HSC002	3	Halbsynchronisierte Klasse
PS001	8	Primitive Sychronisierungsmechanismen

PS001

Die Warnung für Low-Level Synchronisation wird in allen Fällen wegen Verwendung von Methoden der Klasse Interlocked angezeigt. So wird zum Beispiel beim Dekrementieren des Pointers auf den nächsten auszuführenden DB-Command Interlocked.Decrement verwendet. Für ein Projekt wie NHibernate sind diese Verwendungen sicherlich legitim und könnten ignoriert werden.

HSC001 / HSC002

In diesen Fällen wird jeweils jeder Zugriff auf ein Property innerhalb der Klasse zwar gelockt, doch das Property selbst ist als auto-Property implementiert und verfügt über keine Synchronisierung. Dadurch, dass alle Aufrufe gelockt werden, findet noch kein Fehlverhalten statt, doch es könnten sich in Zukunft Fehler einschleichen, indem neue Zugriffe nicht gelockt wären. Sicher wäre es, des Property selbst zu synchronisieren.

Zudem wird die Methode IsUnlockable nur in synchronisierten Kontexten aufgerufen und in Betrachtung der Implementation ist dies auch anzuraten. Bei diesem Design kann es jedoch schnell passieren, dass bei Ergänzungen die Methode einmal in einem unsynchronisierten Kontext aufgerufen wird, sicherer wäre es also, die Methode selbst zu synchronisieren.

Hier die Implementation dieser Methode zusammen mit der von ParaSmeller angezeigten Warnung:

```
        }

        /// <summary>
        /// Is the client's lock commensurate with the item in the cache?
        /// If it is not, we know that the cache expired the original
        /// lock.
        /// </summary>

        private bool IsUnlockable(ISoftLock clientLock, ILockable myLock)
        {
            //null clientLock is remotely possible but will never happen in practice
            return myLock != null &&
                   myLock.IsLock &&
                   clientLock != null &&
                   ((CacheLock) clientLock).Id == ((CacheLock) myLock).Id;
        }
    }
}
```

Abbildung 21 Beispiel eines gefunden Smells (Halbsynchronisierte Klasse)

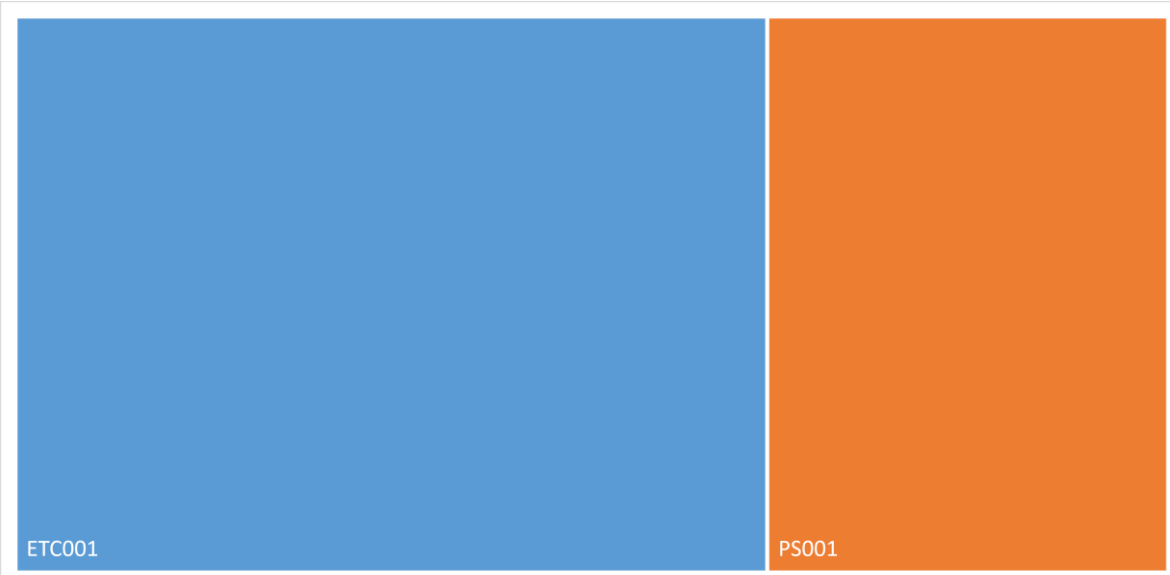
Fazit

Bei einem weit verbreiteten Projekt wie NHibernate ist zu erwarten, dass nicht sehr viele gravierenden Fehler oder potentielle Fehlerquellen im Code lauern. Abgesehen von den Warnungen vor Low-Level Synchronisation, welche hier nicht angebracht sind, wurden keine Falls Positives gefunden. Die Diagnose von nur 12 Warnungen bei solch einer Codemenge erscheint jedoch ziemlich klein. Durch verfeinerte Analysen dürfte noch mehr im Code zu entdecken sein.

OpenRA

Grösse	320'000 LOC
Quelle	https://github.com/OpenRA/OpenRA

Übersicht Code Smells



Code Smell	Anzahl	Beschreibung
ET001	15	Explicit Threads
PS001	8	Primitive Sychronisierungsmechanismen

ET001

In der Auflistung sind keine False Positives vorhanden, die Warnungen verweisen auf den Gebrauch der Thread-Klasse. Hier wäre Potential vorhanden, auf Tasks umzusteigen.

PS001

Die Warnungen für Low-Level Synchronisation wird durchwegs wegen der Verwendung von Low-Level Synchronisierungsmechanismen wie Interlocked oder volatile ausgegeben. Es befinden sich keine False Positives darunter, es ist jedoch offensichtlich, dass eine Game Engine Gebrauch von diesen Synchronisierungsmechanismen machen muss.

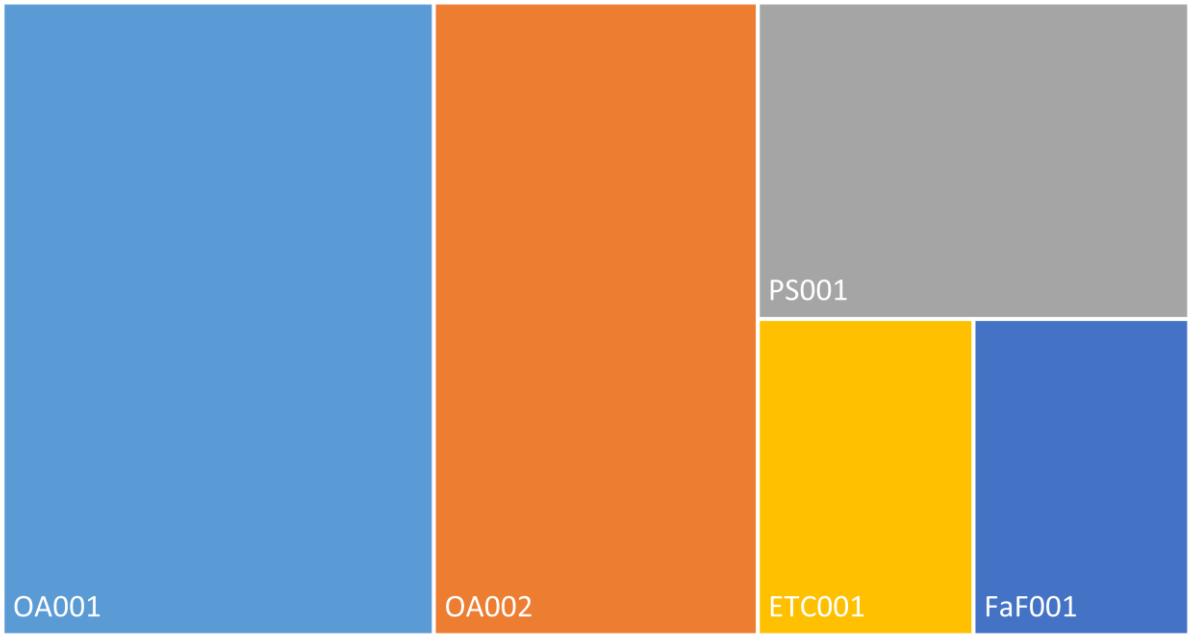
Fazit

Wie bei einer Game Engine, bei der Performance und Parallelität wichtige Anforderungen sind, zu erwarten war, meldet der Checker das Vorhandensein von einigen primitiven Synchronisierungsmechanismen sowie viele Verwendungen der Thread-Klasse. Dies sind in dieser Domäne natürlich legitim, doch es böte sich die Ablösung von expliziten Threads an. Es werden keine potentiellen Fehler im Bereich der parallelen Programmierung gemeldet, was bei einer weit verbreiteten Game Engine auch glaubwürdig erscheint, da diese wohl schnell zu Symptomen in der Anwendung führen würden.

Roslyn

Grösse	**1’500’000 LOC**
Quelle	https://github.com/dotnet/roslyn

Übersicht Code Smells



Code Smell	Anzahl	Beschreibung
ET001	1	Explicit Threads
OA001	4	Überasynchronität
OA002	3	Überasynchronität
PS001	2	Primitive Sychronisierungsmechanismen
FaF001	1	Fire and Forget

OA001 / OA002

Einzig in den Tests gibt es Überasynchronitätswarnungen. So werden Testfunktion, welche async sind, nicht public deklariert, was in einem Test auch nicht weiter schlimm ist. Hauptsächlich wird dies asynchron durchgeführt, damit der User jederzeit mit einem Tastendruck den Test stoppen kann, da dieser in der Konsole ausgeführt wird. Die tiefe Verschachtelung von asynchronen Methoden findet man auch nur in Testprojekten und nicht im richtigen Code.

PS001

Die beiden Interlocked.CompareExchange werden beim Allozieren von Memory verwendet. Dies lässt darauf schliessen, dass diese Klasse eher Lowlevel operiert und der Einsatz von solch primitiven Sychronisierungsmechanismen in Ordnung ist.

```
private T AllocateSlow()
{
    var items = _items;

    for (int i = 0; i < items.Length; i++)
    {
        // Note that the initial read is optimistically not synchronized. That is intentional.
        // We will interlock only when we have a candidate. in a worst case we may miss some
        // recently returned objects. Not a big deal.
        T inst = items[i].Value;
        if (inst != null)
        {
            if (inst == Interlocked.CompareExchange(ref items[i].Value, null, inst))
            {
                return inst;
            }
        }
    }

    return CreateInstance();
}
```

Abbildung 22 Primitive Synchronsierung

FaF001

Der Aufruf von Task.Run, auf welchen nicht gewartet wird, meldet über ein Callback selber, dass er fertig ist und somit ist dies auch korrekter Code und die Warnung ist als False Positive zu werten, hier könnte man die Erkennung noch verfeinern.

ET001

Die einzige Warnung wird durch die Benutzung von Thread.Sleep benutzt wird. Dies wird aber eingesetzt, um ein Polling zu implementieren.

Fazit

Der Checker findet bei Roslyn keine schlimmen Smells auf, welche gefährlich für dessen Ausführung sein könnten. Dies ist sicherlich dadurch bedingt, dass an diesem Projekt viele erfahrene Entwickler arbeiten und das meiste sequentiell aufgebaut ist. Die schiere Projektgrösse lassen aber vermuten, dass es doch einige kritische Stellen geben könnte, die man durch eine detailliertere Analyse erkennen würde.

Dining Philosophers

Grösse	**100 LOC**
Quelle	https://code.msdn.microsoft.com/windowsdesktop/Dining-Philosophers-in-C-9ec1dcef

Übersicht Code Smells

Code Smell	Anzahl	Beschreibung
WCT001	2	Waiting Conditions mit Tasks

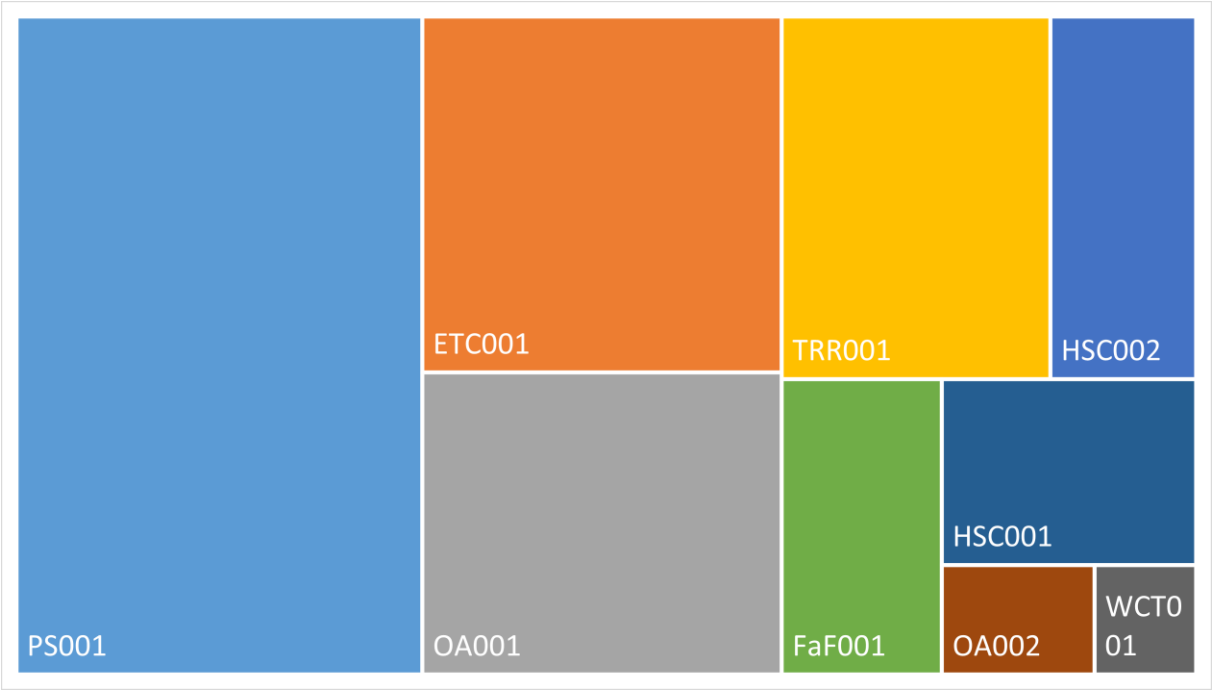
Fazit

Leider konnte die Sperrordnung nicht gefunden, was aber dadurch gegeben ist, dass diese durch Parameter in der Methode dargestellt werden und es so dem Checker nicht möglich ist vorher zu sagen, ob es zu einem Deadlock kommen kann oder nicht. Immerhin wird aber durch die Erkennung von Synchronisierungsmechanismen in Threads auf die Möglichkeit hingewiesen, dass es zu einem Deadlock kommen könnte.

SignalR

Grösse	**27'000 LOC**
Quelle	https://github.com/SignalR/SignalR

Übersicht Code Smells



Code Smell	Anzahl	Beschreibung
HSC001	17	Halbsynchronisierte Klasse
HSC002	19	Halbsynchronisierte Klasse
ET001	46	Explicit Threads
OA001	39	Überasynchronität
OA002	6	Überasynchronität
PS001	96	Primitive Sychronisierungsmechanismen
TRR001	35	Timeouts
FaF001	17	Fire and Forget
WCT001	4	Waiting Conditions in Tasks

PS001

Es finden sich sehr viele primitive Synchronisationsmechanismen. Diese werden aber hauptsächlich dazu verwendet, Variablen zu inkrementieren. Daneben existiert eine Klasse mit dem Namen «HighFrequencyTimer», welche auch Gebrauch von CompareExchange und anderen primitiven Methoden macht. Dies ist aber nicht verwunderlich, da hier bereits aus dem Klassennamen geschlossen werden kann, dass eine hohe Performance nötig ist.

ET001

Die Verwendung der Thread-Klasse ist zu 90% durch den Einsatz von Thread.Sleep bedingt. Die anderen Fälle erzeugen neue Threads, welche für Stresstests benutzt werden. Hier möchte man einen hohen Durchsatz haben, weshalb der Einsatz sicherlich gerechtfertigt ist.

OA001 / OA002

Die recht komplexe und tiefe Asynchronität in diesem Projekt ist wahrscheinlich gewünscht und auch nachvollziehbar. Es ist ein Realtime Framework, welches viele Verbindungen managed sowie Daten sendet und empfängt. Die tiefen Async sind teilweise verständlich, können aber auch einen Overhead mit sich bringen. Schwierig ist hier zu sagen, welche Tiefe in Ordnung geht und an welcher Stelle man einen Schlussstrich ziehen möchte. Zusätzlich zwingen diese Designentscheide einen Entwickler, diesen Folge zu leisten und das Framework nicht durch blockierende Funktionsaufrufe zu stoppen.

Teilweise ist aber nicht sicher, ob es nicht doch tiefgründige Probleme im Framework gibt. Das Erstellen eines Tasks (siehe Bild unten) geschieht mit dem zusätzlichen Aufruf von Task.Delay, da vermutlich früher der Ablauf nicht korrekt funktioniert hat und nun dem StartRequest zusätzlich einen Vorsprung gibt. Dies ist sehr unschön, denn es ist möglich, dass dieser Request noch langsamer unterwegs ist und eine Sekunde notwendig wäre. Viel sicherer wäre die Arbeit mit Sychronisierungsmechanismen oder Events. Aber dies kann nicht abschliessend gesagt werden, da dazu das komplette Framework verstanden sein müsste.

```
else if (context.Request.Path.Value.Contains("/poll"))
{
    return Task.Run(async () =>
    {
        // don't break the poll until start request is handled or a timeout
        // if it is a subsequent poll
        startReceived.WaitOne(3000);
        // give the start request some additional head start
        await Task.Delay(500);
        //subsequent long polling request should not break immediately
        startReceived.Reset();
        context.Response.StatusCode = 500;
        return TaskAsyncHelper.Empty;
    });
}
```

Abbildung 23 Beispiel mit diversen Smells innerhalb eines Tasks

HSC001 / HSC002

Es gibt Felder, welche zwar beim Setzen gelockt werden, aber das Lesen direkt geschieht. Dies ist nicht in Ordnung, da so alte oder halbe Werte gelesen werden könnten. Ebenfalls gibt es diverse Methoden, in welchem eine Variable ungelockt geschrieben wird, in anderen Methoden wird dies bei der gleichen Variablen nur innerhalb eines Locks gemacht.

WCT001

Die gefundenen Stellen sind nicht kritisch, da jeweils genau ein Lockblock verwendet wird und somit kein Deadlock entstehen kann. Diese Locks werden entweder gebraucht, um sicherzustellen, dass keine weiteren Aktionen ausgeführt werden, wenn die Verbindung geschlossen wird. Es ist aber dennoch speziell, für diesen Zweck Locks zu verwenden.

FaF001

Nicht bei allen Tasks welche zurückgegeben werden, wird auch auf ein Ende gewartet. Dies ist aber nicht unbedingt schlecht, da es keine Berechnungen sind, sondern Tasks, welche die Connection managen und auf weitere Requests warten.

TRR001

Den grössten Teil von Timeoutfunktionen findet man in Testklassen, welche testen, ob eine bestimmte Funktion nach einer gewissen Zeit ausgeführt worden ist. Im richtigen Code findet man dies nur, falls eine Verbindung geschlossen werden soll und mit Hilfe des Timeouts eine Meldung ausgegeben wird, falls dies innerhalb eines bestimmten Zeitraums nicht geschehen ist.
Keine der angetroffenen Warnungen könnte tatsächlich zu Starvation führen.

Fazit

Das Framework ist im grossen Ganzen gut aufgestellt. Es wird speziell darauf geachtet, dass keine Warnungen vom Compiler angezeigt werden, auch werden sonst gute Programmierpraktiken eingehalten.

Die Methoden sind aber oft lang und teilweise recht komplex. Die vielen Async-Verschachtelungen und Context-Switches machen es zusätzlich schwierig, das Programm genau zu verstehen. Zudem wurden in der Analyse einige gefährliche Stellen gefunden, welche eine lauernde Quelle für auftretende Symptome wie Data Races sind. Durch die Rückmeldungen der Analyse könnte man das Projekt an einigen Stellen sicherer und wartbarer machen.

ParaSmeller

Grösse	1’200 LOC
Quelle	https://github.com/currencyCon/ParaSmeller

Übersicht Code Smells

Code Smell	Anzahl	Beschreibung
PS001	2	Primitive Sychronisierungsmechanismen

PS001

Die Warnung wird wegen der Verwendung von Low-Level Synchronisierungsmechanismen der Interlocked-Klasse verursacht. Der Checker braucht diese Low-Level Synchronisation nur im Debug-Modus, um den Fortschritt der Untersuchung feingranular ausgeben zu können, im Production-Modus entfallen diese Code-Teile komplett.

Fazit

Wie zu hoffen war, finden sich bei der Library selber keine schwerwiegenden Fehler und die gefundenen Warnungen sind im gegeben Kontext als irrelevant zu betrachten.

Anhang B - Erweiterungen

B-1 Einstellungen

Damit bei Projekten nicht immer sämtliche Analyzer laufen müssen, wäre das Ziel gewesen, die Möglichkeit zu haben, diese Auswahl direkt im Visual Studio zu konfigurieren.

Die Übersicht fände sich wie auch sämtliche anderen Einstellungen unter Tools – Options – ParaSmeller. Zusätzlich könnte man einstellen, ab welcher Tiefe geschachtelte async Funktionen als Warnung angezeigt werden sollen. Auch weitere Konfigurationsmöglichkeiten hätten hier ihren Platz gefunden.

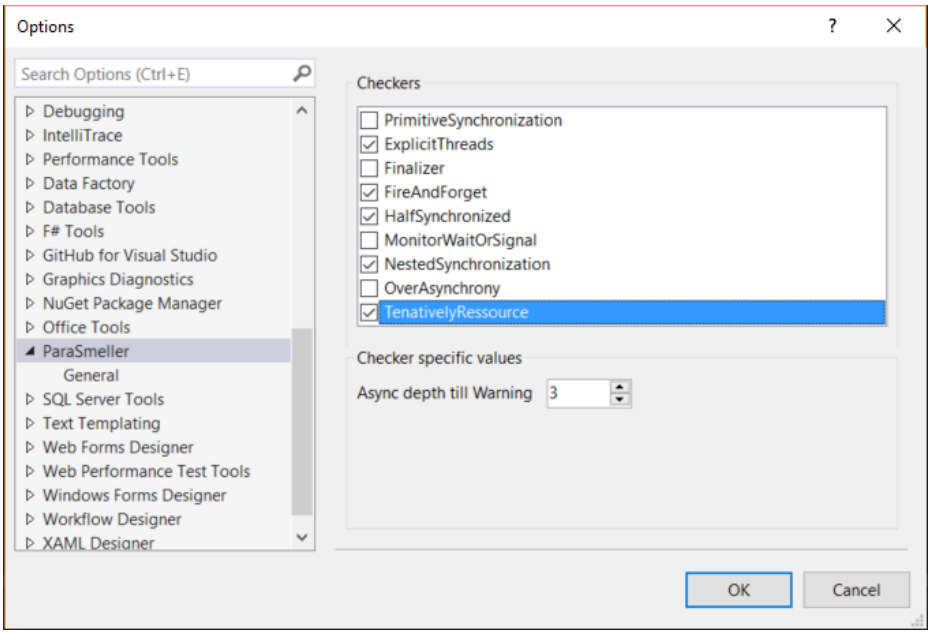


Abbildung 24 Optionen

Dieser Dialog wurde aus einigen Gründen nicht fertiggestellt. Ein grosses Problem ist es, die Einstellungen zur Laufzeit des Checkers auszulesen. Einerseits ist es nicht möglich, während der Laufzeit auf das Visual Studio API zuzugreifen. Sämtliche Möglichkeiten wie Dependency Injection sind verboten und führen zu Exceptions. Auch das setzen über eine static Variable zur Startzeit des Visual Studios funktioniert nicht, da die Checkers in einem anderen Modul geladen werden und nicht sichtbar sind. Die letzte Möglichkeit, die Einstellungen über Dateien austauschbar zu machen, wurde verworfen, da dies innerhalb von Portable Libraries mühsam ist und zu Problemen führen kann.

Daher gibt es aktuell keine Möglichkeit, den Checker anzupassen, ohne ihn neu zu kompilieren. In diesem Bereich könnte ein produktiv eingesetzter Checker massiv mehr für die Benutzerfreundlichkeit machen.

B-2 SonarQube Plugin

SonarQube bietet bereits gute Möglichkeiten, aus einem bestehenden Visual Studio Analyzer ein SonarQube Plugin zu generieren.

Benötigt werden dazu:

- SonarQube-Roslyn-SDK
- C# Plugin (>= 5.3.2)
- SonarQube Scanner for MSBuild (>=2.0)

Falls das .nupkg nur lokal vorhanden ist und sich nicht in einem Repository befindet, müssen Schritt 1 und 2 durchgeführt werden.

Allenfalls kann direkt mit Schritt 3 weitergearbeitet werden.

Plugin erstellen

1. Kopieren des SonarQube-Roslyn-SDK in den Ordner, in dem sich das .nupkg befindet.
2. Nun muss in der nupkg.config der Eintrag **localFeed** nach **./** geändert werden. So wird das Package im lokalen Ordner gesucht.
3. RoslynSonarQubePluginGenerator.exe /a:PACKAGENAME
    a. PACKAGENAME entspricht einfach dem Dateinamen des nupkg
4. Nun optional das SQALE Template ausfüllen, *.sqale.template.xml (dies würde es erlauben, die Technical Depth korrekt für jeden Code Smell zu setzen)
    a. Anschliessend RoslynSonarQubePluginGenerator.exe /a:PACKAGENAME /sqale:[*.sqale.template.xml] ausführen
5. Kopieren der jar Datei nach SONARQUBE/extensions/plugins
6. SonarQube neu starten.

Rules aktivieren

Die Rules sind standardmässig nicht dem Default Quality Profile zugeordnet. Um dies zu tun, ruft man SONARQUBEURL/coding_rules auf, sucht die gewählten Rules und fügt sie dem Profil «sonar way» zu, oder aber man erstellt dafür ein neues Profil.

SonarQube Analyse durchführen

Voraussetzungen: C# Plugin installiert (wichtig Version >=5.3.2), Projekt lässt sich mit MSBuild builden.

1. MSBuild.SonarQube.Runner.exe begin /k:"PROJECT_KEY" /n:"PROJECT_NAME" /v:"VERSION"
2. "C:\Program Files (x86)\MSBuild\14.0\Bin\MSBuild.exe" /t:Rebuild

MSBuild.SonarQube.Runner.exe end

Anhang C - Zeitauswertung

C-1 Allgemeine Übersicht

In diesem Kapitel werden einige Grafiken bezüglich des Arbeitsaufwandes und der Arbeitsverteilung im Projekt dargestellt.

Zeit pro Kategorie

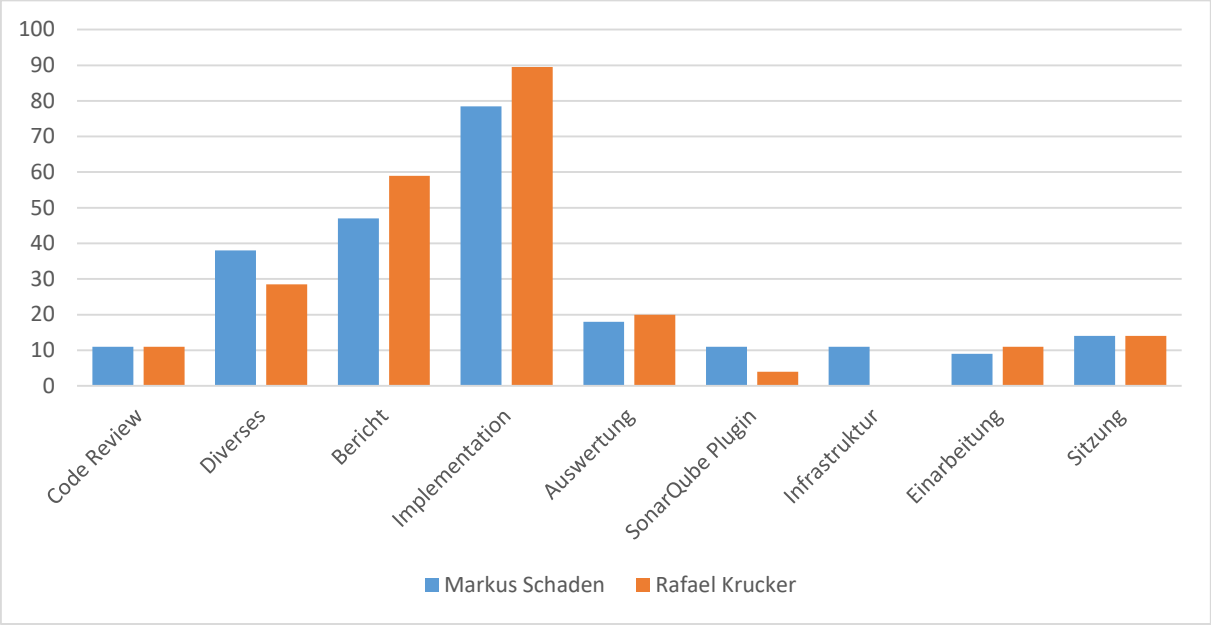


Abbildung 25 Zeitauswertung: Zeit pro Kategorie pro Person

Totaler Aufwand

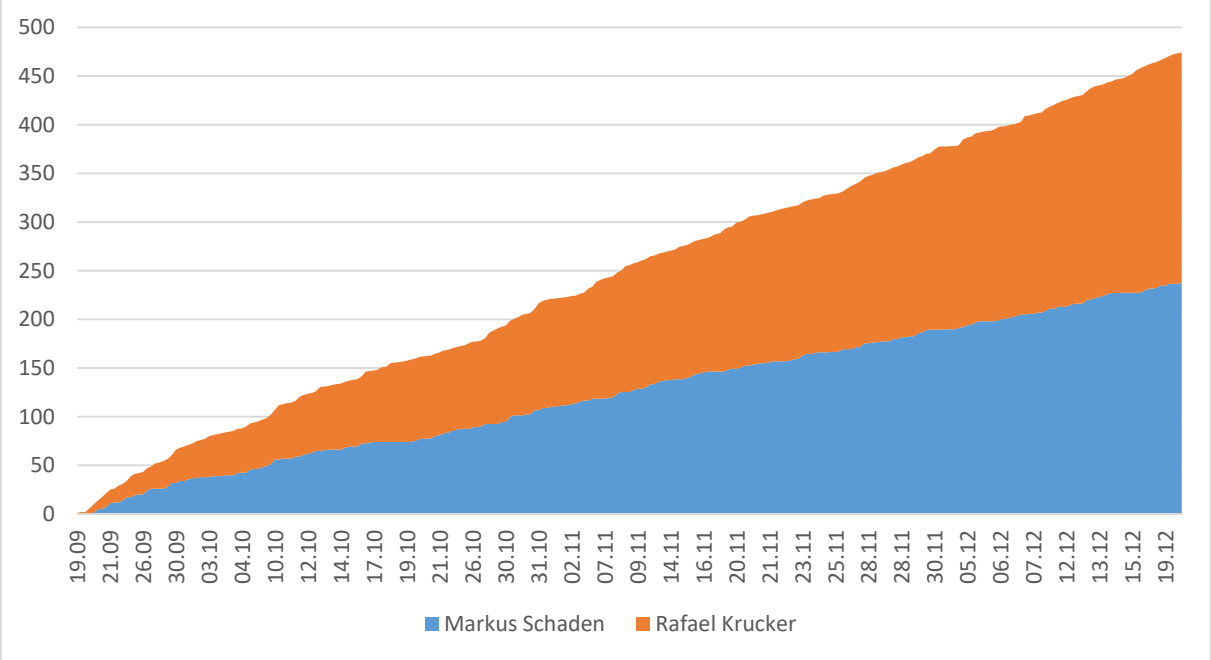


Abbildung 26 Zeitauswertung: Aufsummierte Zeit pro Person

Anhang D – Code Metriken

D-1 Auswertung SonarQube

Während der ganzen Entwicklung war SonarQube immer ein Teil des Continuous Integration Prozesses. Nach jedem Commit konnte so schnell festgestellt werden, in welche Richtung sich die Code-Qualität entwickelt. Zudem wurden bei schlimmen Vulnerabilities Warnungen verschickt, sodass schnell reagiert werden konnte.

Es wurden alle Standardeinstellungen von SonarQube für .Net belassen, das heisst die Codebasis wurde mit 422 Regeln auf Qualität untersucht. Einzig für 5 Methodensignaturen musste eine Ausnahme gemacht werden, da sie als async deklariert waren, jedoch einen Rückgabewert von void anstatt Task hatten, was nicht guter Praxis entspricht und so auch von SonarQube bemängelt wurde. Diese Signaturen werden jedoch vom Code Analyzer Framework von Visual Studio vorgegeben, wenn man Analysen registrieren will. Deshalb blieb nichts Anderes übrig, als Ausnahmen hinzuzufügen.

Nachfolgend einige Eckdaten der Analyse des Projektes durch SonarQube.

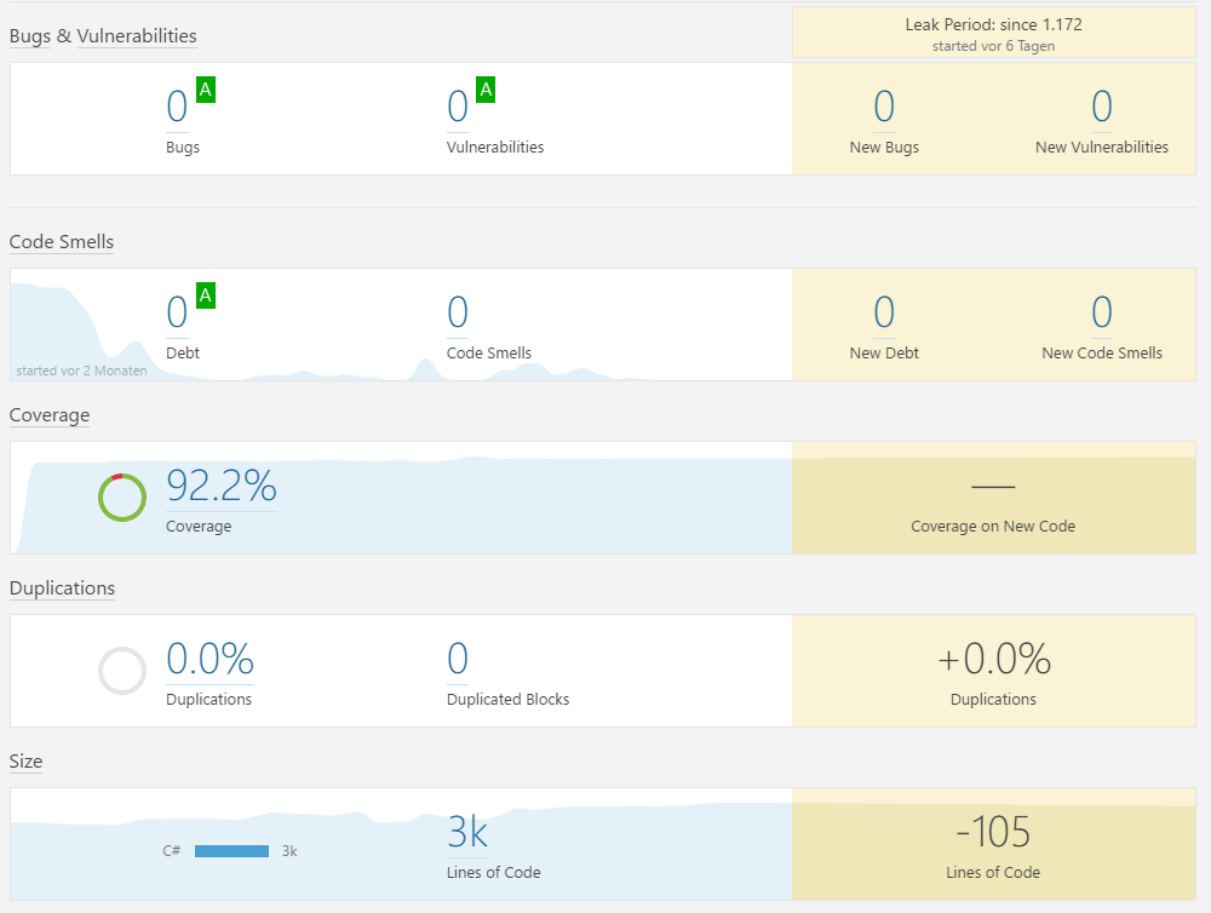


Abbildung 27 Projektübersicht

In der Gesamtübersicht des Projektes ist schnell ersichtlich, dass SonarQube keine Bugs und Vulnerabilities entdeckt. Auch konnten etwa ab der Hälfte des Projektes durch stetige Verfeinerung des Designs und Anpassungen der Review-Prozesse sämtliche von SonarQube entdeckbaren Code Smells vermieden werden.

Die hohe Abdeckung von gut 90% durch Unit Tests sagt natürlich nur teilweise etwas über die Korrektheit des Codes aus, unterstreicht aber die rigide Testkultur und erlaubte angenehmes und sicheres Refactoring der Codebasis.

Während des Entwicklungsprozesses wurde mit Hilfe von Design Reviews und IDE-Tools wie dem Resharper stark darauf geachtet, keinerlei Codeduplikation zu produzieren, SonarQube zeigt, dass diese Bemühungen Früchte getragen haben.

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Coverage	Duplications
Concurrency Checker	3k	0	0	0	92.2%	0.0%
ParaSmeller.Test	452	0	0	0		
ParaSmeller.Vsix	211	0	0	0		0.0%
ParaSmellerAnalyzer	241	0	0	0	90.8%	0.0%
ParaSmellerCore	2.5k	0	0	0	92.4%	0.0%

Abbildung 28 Umfang des Projektes

In der Übersicht von SonarQube ist gut der Gesamtumfang von etwa 3'400 Zeilen Code ersichtlich, der sich auf 4 C#-Projekte aufteilt. Dabei nimmt die ParaSmellerCore Library den grössten Teil ein, da sie die gesamte Logik enthält. Die zwei Projekte ParaSmellerAnalyzer und ParaSmeller.Vsix sind für die Visual Studio Integrierung zuständig und sind wie im technischen Bericht beschrieben sehr schlank gehalten. Letztlich umfasst das Testprojekt etwa 450 Zeilen Code, wovon das meiste allerdings auf Testsourcecode in Strings fällt, welcher auf Smells untersucht wird. Die Logik selbst ist auch im Testprojekt sehr schlank gehalten.

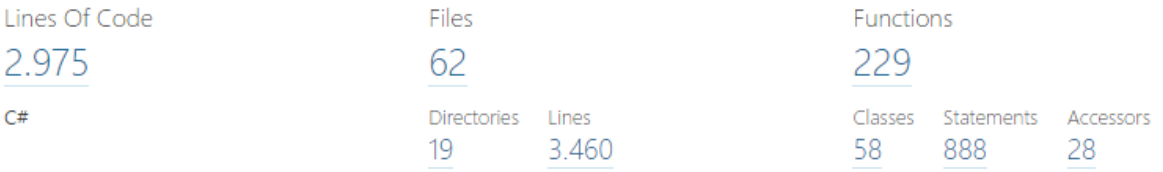


Abbildung 29 Detailaufteilung Codeumfang

Diese Statistik zeigt die Verteilung von Codezeilen, von der Analyse ist das Testprojekt ausgeschlossen. Die knapp 3000 Zeilen Code teilen sich auf knapp 60 Klassen auf, welche zusammen auf etwa 230 Funktionen kommen. Dies ergibt Durchschnittswerte von 50 Zeilen Code pro Klasse, oder 10 Zeilen pro Funktion. Davon ausgeschlossen sind Leerzeilen oder Signaturen. Diese Werte befinden sich aus Sicht des Software Engineerings in einem gut wartbaren Bereich.

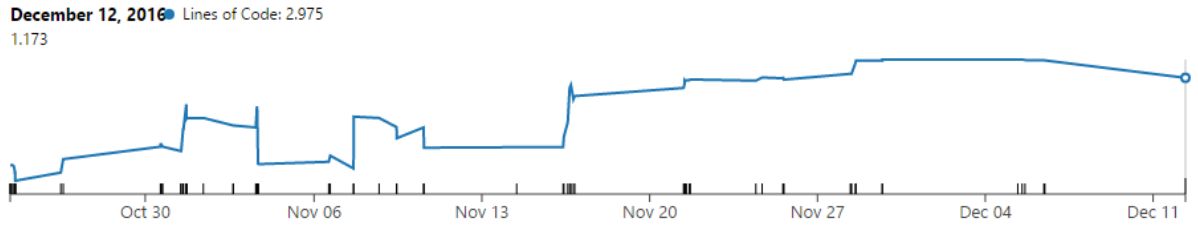


Abbildung 30 Codeumfang über die Zeit

Wie in dieser Abbildung ersichtlich, erhöhte sich der Codeumfang während der Entwicklung stetig. Durch Refactorings und Verbesserung des Designs konnte der Code jedoch auch immer wieder vereinfacht werden, wodurch auch überflüssiger Code wegfiel.

Abbildung 31 Wartbarkeit

In dieser Abbildung zeigt sich, dass jegliche Technical Debt, die SonarQube ausmachen kann, ausgemerzt wurde. Des weiteren berechnet SonarQube aus einigen Metriken wie der Grösse der Komponenten und der zyklomatischen Komplexität und vielen weiteren einen Wartbarkeitsindex. ParaSmeller kommt hier auf den besten möglichen Wert.

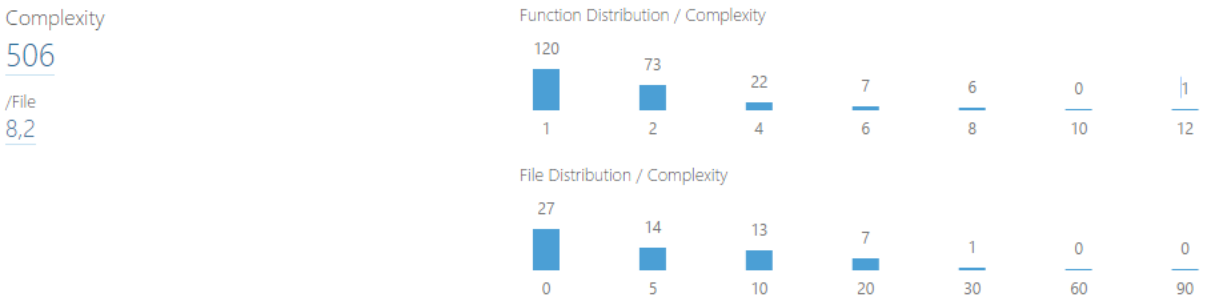


Abbildung 32 Detailübersicht Komplexität

SonarQube erstellt auch detailliert Statistiken über die zyklomatische Komplexität, die es auf einzelne Funktionen und Dateien aufbricht. Die Grafik ist dabei so zu lesen, dass 120 Funktion eine Komplexität von 1 haben, 73 Funktionen eine von 2 und so weiter. Man sieht, dass die allermeisten Komponenten eine sehr tiefe Komplexität haben. Jedoch auch der höchste gemessene Wert, 12, der nur einmal gefunden wurde, ist immer noch als in Ordnung.

Abschliessend kann gesagt werden, dass aus Sicht von SonarQube das Projekt einen sehr guten Stand hat, keinerlei Duplikationen vorkommen, eine hohe Wartbarkeit gegeben ist und die Komplexität tief gehalten wurde. Des Weiteren konnte jegliches Auftreten von Code Smells, Bugs und Vulnerabilities, die SonarQube erkennen kann, vermieden werden.

Anhang E – Glossar

.NET

Eine Laufzeitumgebung von Microsoft, für die Programme in verschiedenen Sprachen geschrieben werden kann. 5, 9, 10, 12, 22, 26, 28, III, XXXV, XXXIX

API

Application Programming Interface. 7, 9, 10, I, XXXII, XXXVIII, XXXIX

C#

Eine multi-Paradigma Sprache die strenge Typisierung beinhaltet und deklarative, imperative, funktionale, generische, objektorientierte und componentorientierte Programmierdisziplinen vereint. 5, 7, 11, 23, 24, XXXIII, XXXVI, XXXIX

Code Smell

Strukturen im Code die auf einen Verstoss gegen fundamentale Design Prinzipien hinweisen und die Code Qualität negativ beeinflussen.

Continuous Integration

Entwicklungsstrategie, bei der Entwickler ihren Code mehrmals täglich in eine bestehende Basis integrieren, woraufhin automatisiert Analysen und Artefakte produziert werden. 5, 8, 20, XXXIV, XXXV

Data Races

Situation bei der mindestens 2 Threads in einem Prozess gleichzeitig auf eine Speicherstelle zugreifen, wobei mindestens ein Zugriff schreibend ist und zudem keine Synchronisierungsmechanismen benutzt werden. 5, 8, 12, 26, 28, 30

Deadlock

Situation bei der jedes Mitglied einer Gruppe von Threads darauf wartet, dass jeweils ein anderer Thread ein Lock freigibt. 5, 8, 10, 12, 25, 30, XIV, XV, XXVII, XXIX

IDE

Integrated Development Environment. 9, 10, XXXVI

ET

Explicit Threads Code Smell, Explizite Threads.

FaF

Fire and Forget Code Smell.

FS

Finalizer Code Smell.

HSC

Half Synchronized Class Code Smell, Halb Synchronisierte Klasse.

MWS

Monitor or Wait Signal Code Smell, Einfaches Monitor Wait oder Signal.

NuGet

Paketmanager für die Entwicklerplattform von Microsoft. 18, 20

Node

Ein Element in einem Syntaxbaum. 10, 17

NSMC

Nested Synchronized Method Class Code Smell, Geschachtelte synchronisierte Methodenaufrufe.

OA

Over Asynchrony Code Smell, Über Asynchronität.

PS

Primitive Synchronization Code Smell, Primitive Synchronisierungsmechanismen.

Roslyn

Compiler, den Microsoft für ihre .Net Umgebung einsetzt. 5, 7, 9, 10, 24, I, XXV, XXVI, XXXII, XXXIII, XXXVIII, XXXIX

SDK

Software Development Kit. XXX

SonarQube

Open Source Qualitätsmanagementplattform für die kontinuierliche Analyse von Source Code. 5, 7, 20, XXXIII, XXXV, XXXVI, XXXVII

Starvation

Situation, in der einem Thread fortwährend der Zugriff auf eine Ressource verwehrt wird, welche er für die Weiterführung braucht. 13, 30

TFS, Team Foundation Server

Produkt von Microsoft für das managen von Source Code. 5

TRR

Tentatively Resource Reference Code Smell, Versuchsweiser Ressourcenbezug.

VSIX

Visual Studio Integration Package. XXXVI

WCT

Waiting Conditions between Tasks Code Smell, Warteabhängigkeiten unter Tasks.

Anhang F – Verzeichnisse

F-1 Abbildungsverzeichnis

F-2 Tabellenverzeichnis

F-3 Quellenverzeichnis

Bläser, L. (28. 8 2016). *Heise*. Abgerufen am 1. 12 2016 von Heise: https://www.heise.de/developer/artikel/Parallel-Code-Smells-Eine-Hitparade-3248406.html

Fowler, M. (1999). *Refactoring. Improving the Design of Existing Code.* Addison-Wesley.

Stackoverflow. (15. 12 2016). *Stackoverflow*. Von Stackoverflow: http://stackoverflow.com/questions/36603119/how-to-switch-off-solution-wide-analysis-in-visual-studio-2015/37946306#37946306 abgerufen

Suryanarayana, G. (2014). *Refactoring for Software Design Smells.* Morgan Kaufmann.