#JAVAPRO #Architecture

Architektur-Hotspots aufspüren

Strukturelle Abhängigkeitsanalysen sind von begrenzter Aussagekraft. Teilweise erfassen sie die wesentlichen Architektur-Hotspots nicht, wie anhand eines Code-Beispiels exemplarisch gezeigt wird. Die vorgestellten Konzepte zur Feature-Modularität sind eine neue Art die System-Komplexität zu erfassen. Sie zielen darauf ab, Architektur-Hotspots zu identifizieren, die den (Wartungs-) Aufwand für neue Features ansteigen lassen.

Ein weit verbreiteter Ansatz zur Bewertung von Softwaresystemen und deren Architekturen besteht in der Analyse ihrer Abhängigkeitsstruktur. Ziel ist es Abhängigkeitsstrukturen zu identifizieren, die allgemein als schwer wartbar bekannt sind, zum Beispiel zyklische Abhängigkeiten und bestimmte systemspezifische Verletzungen wie schichtübergreifende Abhängigkeiten. Es besteht kein Zweifel, dass die Abhängigkeitsanalyse wertvolle Einblicke in die Struktur eines Softwaresystems liefert und tatsächlich echte Wartungsprobleme aufdecken kann. Trotz ihrer Nützlichkeit hat die klassische Abhängigkeitsanalyse aber große Schwächen.

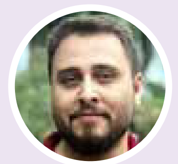
Schwächen klassischer Abhängigkeitsanalysen

Sie berücksichtigt nur strukturelle Abhängigkeiten, zum Beispiel Aufruf- und Import-Abhängigkeiten. Sie versäumt es implizite Abhängigkeiten zu erkennen, wodurch verschiedene Artefakte (Dateien, Module, Klassen etc.) gleichzeitig angepasst werden müssen, ohne dass eine sichtbare Abhängigkeit zwischen ihnen besteht. Zudem ergibt die strukturelle Abhängigkeitsanalyse oft positive aber falsche Ergebnisse. Auch wenn problematische Abhängigkeitsstrukturen sicher auf potenzielle Schwierigkeiten hindeuten, müssen sie im Sinne eines erhöhten Wartungsaufwandes keine wirklichen Probleme verursachen. Die Gründe dafür können unterschiedlich sein:

- Code-Module, die von schlechten Abhängigkeitsstrukturen betroffen sind, laufen stabil und werden während des Lebenszyklus eines Softwaresystems nicht sehr oft verändert.
- Die Entwickler sind sich der potenziell problematischen Stellen bewusst und berücksichtigen die Auswirkungen bei der Implementierung entsprechend, so dass unangenehme Überraschungen wie Folgefehler vermieden werden.
- Nicht jede zyklische Abhängigkeit ist für sich genommen problematisch. Einige könnten sogar absichtlich eingeführt werden, um große Module in kleinere aufzuteilen. Dabei sind die kleineren Module als Teil der Implementierung und weniger als Teil des Designs der Architektur anzusehen.

Autor:

Konstantin Sokolov ist Mit-Gründer der Cape of Good Code GmbH. Er hat deren DETANGLE Software Analyse Suite, die zur Software Qualitätssicherung, Qualitätsaudits und Technical Due Diligence Projekten eingesetzt werden, maßgeblich mitgestaltet.



Ural Sezer ist Senior Engineer und Experte für Frontend-Technologien, die unter anderem bei DETANGLE zum Einsatz kommen.

Im Allgemeinen versucht die klassische Abhängigkeitsanalyse Fälle einer engen Kopplung aufzudecken. Dabei bleibt sie jedoch oft zu feingranular und kurzsichtig und lässt das große Ganze außer Acht. In realen Systemen können strukturelle Kopplungen zwischen einzelnen Codemodulen oder deren Fehlen nicht als ausreichende Bedingung für eine schlechte bzw. gute Wartbarkeit angesehen werden. Es besteht kein Zweifel, dass das Vorhandensein einer engen Kopplung ein starker Indikator für Probleme bei der Wartbarkeit ist. Diese Zuversicht folgt auch aus der Erfahrung. Es scheint jedoch, dass allein die Betrachtung struktureller Kopplungen zwischen einzelnen Codemodulen zu ungenau ist. Welche Art von Kopplung ist also im Hinblick auf die Wartbarkeit wirklich wichtig?

Von der Modulkopplung zur Idee der Feature-Kopplung

Die Beantwortung dieser Frage erfordert eine Definition der Wartbarkeit. Eine einfache könnte sein:

> „Der Aufwand der erforderlich ist, um neue Funktionalitäten in ein wartbares System einzuführen, ist ungefähr proportional zur Komplexität der Funktionalität, jedoch nicht zur Komplexität des gesamten Systems."

Einfach ausgedrückt: In einem wartbaren System sollte es nicht der Fall sein, dass die Implementierung neuer Features immer schwieriger wird und immer mehr Zeit in Anspruch nimmt. Umgekehrt bedeutet dies, dass ein System unabhängig von der Qualität seiner Abhängigkeitsstruktur oder möglicherweise anderer Metriken als wartbar zu betrachten ist, solange der durchschnittliche Aufwand pro Feature nicht weiter steigt. Wen kümmert eine enge strukturelle Kopplung und zyklische Abhängigkeiten, sofern sie nicht den gesamten Entwicklungsprozess verlangsamen? Und was ist eine strukturelle Modularisierung mit einer schönen Abhängigkeitsstruktur wert, wenn man immer beständig Projekttermine verpasst?

Ein genauerer Indikator für den potenziellen Aufwand zur Einführung neuer Features in ein System muss gefunden werden. Und dieser Indikator sollte sich wohl am besten auf die Funktionalität und deren Implementierung fokussieren. Features sind als Hauptbausteine von Softwaresystemen zu behandeln - und nicht Codemodule, Dateien oder Klassen. Es ist zu erwarten, dass die Analyse und Verbesserung der Beziehungen zwischen Features, und nicht zwischen Code-Modulen allein, genauere Aussagen über die Wartbarkeit eines Systems ermöglichen. In der Regel kann ein Feature in einem System umso schneller implementiert werden, je weniger umgebender Code verstanden, geändert und erneut getestet werden muss und je weniger unverwandter Code, der für andere Features verantwortlich ist, betroffen ist. Je weniger sich eine Feature-Implementierung mit anderen Feature-Implementierungen überschneidet und diese möglicherweise stört, desto eher ist es möglich sie in einer angemessenen Zeit zu implementieren ohne die bestehende Funktionalität zu beeinträchtigen. Der Grad der Schnittmenge eines Features mit anderen Features im Code wird hier als Feature-Kopplung verstanden. In diesem Zusammenhang versteht man unter einem Feature:

- Eine von Menschen lesbare Beschreibung der erwarteten Funktionalität, der eine eindeutige ID zugewiesen wird. Normalerweise werden Features in Issue-Tracker-Systemen gepflegt und als Tickets mit entsprechenden Ticket- oder Issue-IDs ausgedrückt.
- Ein Satz von Commits in einem Versionskontrollsystem, die erforderlich sind um die notwendigen Code-Änderungen einzuführen, welche die Funktionalität des Features ausmachen.

Um die Feature-Kopplung programmatisch analysieren zu können muss jeder Commit, der zu einem Feature gehört, mit der entsprechenden Feature-ID verknüpft werden.

Feature-Kopplung an einem Code-Beispiel

In diesem Abschnitt wird ein Beispiel skizzenhaft in Java implementiert. Es geht darum, Objekte in der realen Welt zu erkennen und sie auf lustige Weise zu dekorieren. Zu diesem Zweck hat die Software Zugriff auf eine drehbare Kamera. Für den ersten Meilenstein sind die folgenden User-Stories zu implementieren:

- User-Story #1: Der Benutzer möchte vom System erkannte Autos lustig dekorieren (Lustige Autos)
- User-Story #2: Der Benutzer möchte vom System erkannte Personen lustig dekorieren können (Lustige Personen)

Abkürzend wird im Folgenden an manchen Stellen User-Story #1 als das Feature "Lustige Autos" und User Story #2 als "Lustige Personen" referenziert. Als Teil von User-Story #1 wird Skeleton-Code für die Darstellung und Wahrnehmung der Welt erstellt. Es werden zwei neue Dateien hinzugefügt: **`WorldModel.java`** und **`WorldParser.java`** **(Listing 1 & Listing 2)**.

(Listing 1 - WorldModel.java mit Commit 1)

```
import WorldObject
import Observer

public final class WorldModel {
    public enum EventType {}
    private HashMap<EventType, List<Observer>>
        eventObserverMap = new HashMap<>();

    public WorldModel() {
    }

    public void subscribe(Observer observer, EventType event) {
        var observerList =
            eventObserverMap.getOrDefault(event, new Array-
List<>());
```

```
        observerList.add(observer);
        eventObserverMap.put(event, observerList);
    }

    private void notify(EventType event, WorldObject obj) {
        var observerList =
            eventObserverMap.getOrDefault(event, new Array-
List<>());
        observerList.forEach(observer -> observer.onEvent(event,
obj));
    }
}
```

(Listing 2 - WorldParser.java mit Commit 1)

```
import Frame
import Camera
import WorldModel

public class WorldParser {
    private Camera cam = new Camera();

    public WorldParser() {
    }

    public void mainLoop() {
        while (true) {
            processFramesFor10Seconds();
            cam.rotate(45);
        }
    }

    public void processFramesFor10Seconds() {
        var iterator = cam.getFrameIterator();
        while (iterator.hasNext()) {
            parseFrame(iterator.next());
        }
    }

    private void parseFrame(Frame frame) {
        // detect concrete objects starting from data3
        byte[] data1 = lowLevelProcessing1(frame);
        byte[] data2 = lowLevelProcessing2(data1);
        byte[] data3 = lowLevelProcessing2(data2);
    }

    private byte[] lowLevelProcessing1(Frame frame) {
        //  retrieve first-level raw data
        ...
    }
    ...
}
```

Die WorldModel-Klasse dient als Container von Objekten, die derzeit in der Welt repräsentiert werden, und implementiert das Observer-Muster, um bestimmte Ereignisse im Zusammenhang mit diesen Objekten zu verteilen. Die **`WorldParser`** Klasse beobachtet die Umgebung durch Drehen der Kamera, um konkrete Objekte zu erkennen. Weitere drei Commits implementieren die Funktionalität hinter User-Story #1, womit sich sukzessive Codes (Listing 3, Listing 4 und Listing 5) ergibt.

(Listing 3 - WorldModel.java mit Commit 2)

```
public final class WorldModel {

    public enum EventType { CAR_APPEARED }

    private ArrayList<Car> cars = new ArrayList<>();

    ...

    public void addCar(Car car) {
        cars.add(car);
        notify(EventType.CAR_APPEARED, car);
    }
}
```

(Listing 4 - WorldParser.java mit Commit 3)

```
public class WorldParser {
    ...

    private void parseFrame(Frame frame) {
        // detect concrete objects starting from data3
        byte[] data1 = lowLevelProcessing1(frame);
        byte[] data2 = lowLevelProcessing2(data1);
        byte[] data3 = lowLevelProcessing2(data2);
        var car = findCar(data3);
        car.ifPresent(value -> WorldModel.getInstance().
        addCar(value));
    }

    private Optional<Car> findCar(byte[] data) {
        ...
    }
}
```

(Listing 5 - FeatureFunnyCar.java mit Commit 4)

```
public class FeatureFunnyCar extends Observer {

    public FeatureFunnyCar() {
        WorldModel.getInstance().subscribe(
            this, WorldModel.EventType.CAR_APPEARED);
    }

    @Override
    public void onEvent(WorldModel.EventType event,
    WorldObject obj) {
        makeFunnyCar((Car)obj);
    }

    private void makeFunnyCar(Car car) {
        ...
    }
}
```

Nun wird User-Story #2 (Personen lustig dekorieren) ebenfalls mittels sehr ähnlicher drei Commits umgesetzt, so dass sich am Ende zusätzlich zum neuen Code-Modul **`FeatureFunny Person.java`** die Module **`WorldModel.java`** (Listing 6) und **`World Parser.java`** (Listing 7) ergeben.

(Listing 6 - WorldModel.java mit Commit 7)

```
public final class WorldModel {

    public enum EventType { CAR_APPEARED, PERSON_APPEARED }

    private ArrayList<Car> cars = new ArrayList<>();
    private ArrayList<Person> persons = new ArrayList<>();

    ...
    public void addCar(Car car) {
        cars.add(car);
        notify(EventType.CAR_APPEARED, car);
    }

    public void addPerson(Person person) {
        persons.add(person);
        notify(EventType.PERSON_APPEARED, person);
    }
}
```

(Listing 7 - WorldParser.java mit Commit 7)

```
public class WorldParser {
    ...
    private void parseFrame(Frame frame) {
        // detect concrete objects starting from data3
        byte[] data1 = lowLevelProcessing1(frame);
        byte[] data2 = lowLevelProcessing2(data1);
        byte[] data3 = lowLevelProcessing2(data2);
        var car = findCar(data3);
        car.ifPresent(value -> WorldModel.getInstance().
       addCar(value));
        var person = findPerson(data3);
        person.ifPresent(value ->
            WorldModel.getInstance().addPerson(value));
    }

    private Optional<Car> findCar(byte[] data) {
        ...
    }

    private Optional<Person> findPerson(byte[] data) {
        ...
    }
}
```

Die bisherige Commit-Historie sieht wie in (Tab 1.) dargestellt aus.

	Commit-Message	Code-Module
7	US #2: Funktion "Lustige Personen" hinzufügen.	`FeatureFunnyPersons.java` hinzufügen
6	US #2: Erweitern des Parsers, um Personen zu erkennen	`WorldParser.java` anpassen
5	US #2: Erweitern des Modells, um Personen darzustellen	`WorldModel.java` anpassen
4	US #1: Funktion "Lustige Autos" hinzufügen.	`FeatureFunnyCars.java` hinzufügen
3	US #1: Erweitern des Parsers, um Autos zu erkennen	`WorldParser.java` ändern
2	US #1: Erweitern des Modells, um Autos darzustellen	`WorldModel.java` ändern
1	US #1: Skeleton-Code zur Weltdarstellung und -wahrnehmung hinzufügen	`WorldModel.java` und `WorldParser.java` hinzufügen

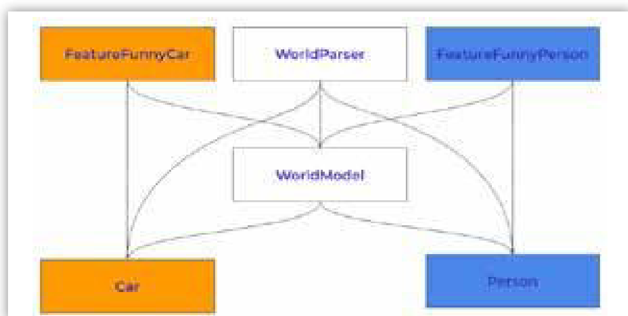
Bisherige Commit-Historie (Tab. 1)

Betrachten wir nun die Abhängigkeitsstruktur der aktuellen Codebasis (Revision 7), um erste Einblicke in die architektonische Qualität unseres Projekts zu erhalten (Abb. 1).

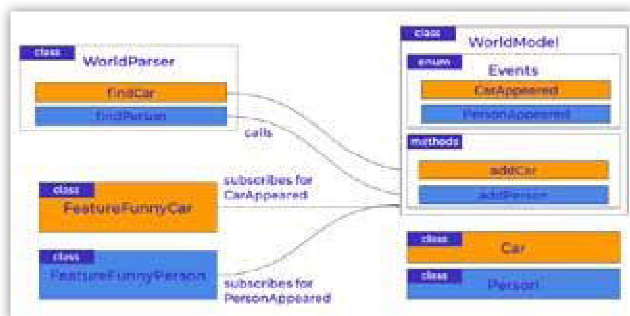
Auf den ersten Blick sieht die modulare Struktur des Code-Beispiels recht überzeugend aus. Es wurde eine hierarchische und azyklische Abhängigkeitsstruktur erreicht. Aber an den Abhängigkeiten des `WorldModel` zu `Car` und `Person` deuten sich die Schwächen schon an. Ein anderes detaillierteres Bild macht die Problematik noch ersichtlicher. Die orangen Markierungen kennzeichnen die Änderungen, die aufgrund User-Story #1 nötig waren, während die blauen Markierungen die Anpassungen gemäß User-Story #2 darlegen (Abb. 2)

Diese Architektur ist eher suboptimal:

1. Beim Hinzufügen jedes neuen Features müssen fast alle Module im System verändert werden.
2. In der `WorldParser` Klasse werden Erkennungsalgorithmen



Abhängigkeitsstruktur - hierarchisch strukturiert, azyklisch, aber der Schein trügt (Abb. 1)

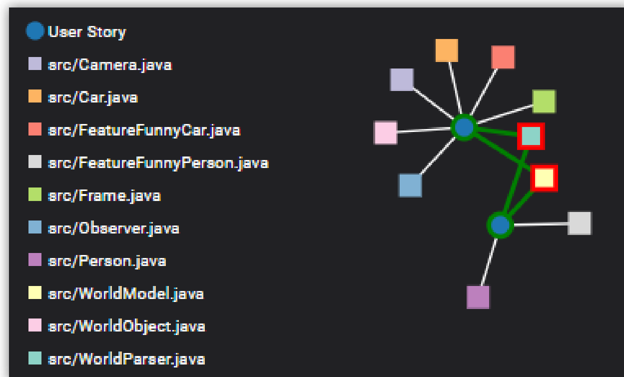


Abhängigkeitsstruktur - gut, aber mit unerkannten Architektur-Hotspots (Abb. 2)

für verschiedene Objekte vermischt, das Modul wird wachsen und unverständlich werden.

3. In der **WorldModel** Klasse ist die Situation aber noch schlimmer, weil das Code-Modul für jedes Feature an drei unterschiedlichen Stellen verändert werden muss.

Es kann argumentiert werden, dass die Architektur problematisch ist, dies aber nicht an der Abhängigkeitsstruktur erkennbar ist. Die gleichzeitig nötigen Anpassungen der Klassen **WorldModel** und **WorldParser** im Falle weiterer Features der Art „Lustige X" werden nicht sichtbar. Daher ergibt sich die Frage: woran können wir es dann erkennen? Es wird anhand der Feature-Kopplung und deren Visualisierung ersichtlich (Abb. 3).



Feature-Kopplung visualisieren - Architektur-Hostpots erkennbar (Abb. 3)

Das Netzwerk-Diagramm beinhaltet blaue Kreise, welche die ersten beiden User-Stories darstellen. Jede User-Story ist per Kante mit den Code-Modulen verbunden, die zu ihrer jeweiligen Implementierung beigetragen haben. Code-Module werden als Rechtecke visualisiert. Es wird deutlich, dass **WorldModel.java** und **WorldParser.java** als Code-Module aufgrund beider User-Stories verändert wurden. Die Umsetzung einer weiteren User-Story zur Erfassung eines neuen Features zur Dekoration anderer Objekttypen würde wiederum diese beiden Code-Module anpassen. Diese beiden Code-Module weisen daher Verbindungen zu vielen gemeinsamen User-Stories auf. Sie stellen rot markierte architekturelle Hotspots des Systems dar, die anhand der klassischen Abhängigkeitsgraphen nicht sichtbar werden. Diese beiden Code-Module weisen eine hohe Feature-Kopplung auf, d.h. sie tragen gemeinsam zu vielen gleichen User-Stories (und dem Feature dahinter) bei.

Komplizierte Abhängigkeiten, bessere Feature-Kopplung

Nun wird ein Refactoring durchgeführt. Die Arbeiten zum Refactoring

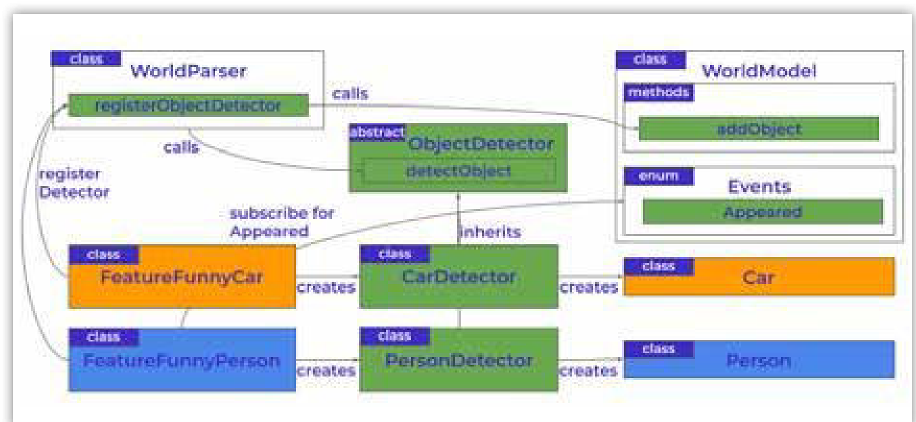
	Commit-Message	Code-Module
8	DR #3: Refactoring Modell/ Parser mittels Detector-Klassen	`WorldParser.java` ändern `WorldModel.java` ändern `FeatureFunnyCar.java` ändern `FeatureFunnyPerson.java` ändern `ObjectDetector.java` hinzufügen `CarDetector.java` hinzufügen `PersonDetector.java` hinzufügen
...	...	...
1	US #1: Skeleton-Code zur Weltdarstellung und -wahrnehmung hinzufügen	`WorldModel.java` und `WorldParser.java` hinzufügen

Commit-Änderungen (Tab. 2)

werden als eigenes Development-Requirement mit der ID #3 und dem Titel **Refactoring zur Entkoppelung des Modells und des Parsers** erfasst. Damit gehen Commit-Änderungen einher (Tab. 2).

Es wurde für jeden zu erkennenden Objekttyp eine eigene Detector-Klasse **CarDetector** und **PersonDetector** eingeführt, die von einer gemeinsamen abstrakten Klasse **ObjectDetector** abgeleitet ist. Bestehende und künftige Features, z.B. **Feature FunnyCar**, zur Dekoration von Objekten instanziieren nun einen jeweiligen **ObjectDetector** (z.B. **CarDetector**), um ihn beim Modell zu registrieren. Der jeweilige Detector erkennt das zugehörige Objekt und erzeugt das entsprechende Modell-Objekt, zum Beispiel **Car**. Das Diagramm (Abb. 4) skizziert die Kollaboration der Klassen nach dem Refactoring.

Der Vollständigkeit halber seien noch die revidierten Code-Stände der Klassen **WorldModel** und **WorldParser** aufgezeigt (Listing 8 und 9).



Feature-Kopplung visualisieren - Architektur-Hostpots ersichtlich (Abb. 4)

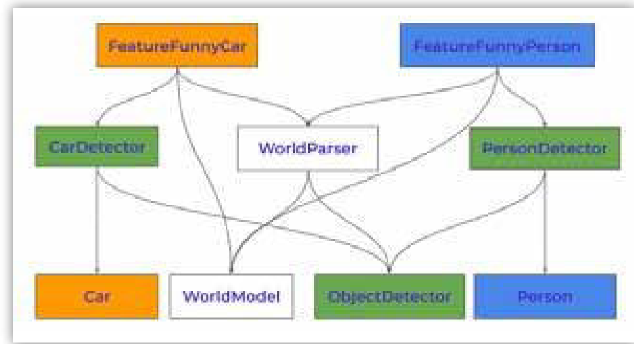
(Listing 8 - WorldParser.java mit Commit 8)

```
public class WorldParser {
    ...
    private ArrayList<ObjectDetector> objectDetectors = new
    ArrayList<>();

    public void registerObjectDetector(ObjectDetector
    objectDetector) {
        objectDetectors.add(objectDetector);
    }

    private void parseFrame(Frame frame) {
        // detect concrete objects starting from data3
        byte[] data1 = lowLevelProcessing1(frame);
        byte[] data2 = lowLevelProcessing2(data1);
        byte[] data3 = lowLevelProcessing2(data2);
        var car = findCar(data3);
        car.ifPresent(value -> WorldModel.getInstance().
         addCar(value));
        var person = findPerson(data3);
        person.ifPresent(value ->
            WorldModel.getInstance().addPerson(value));
        for (ObjectDetector objectDetector : objectDetectors) {
            var obj = objectDetector.detectObject(data3);
            obj.ifPresent(value ->
                WorldModel.getInstance().addObject(value));
        }
    }
```

(Listing 9 - WorldModel.java mit Commit 8)

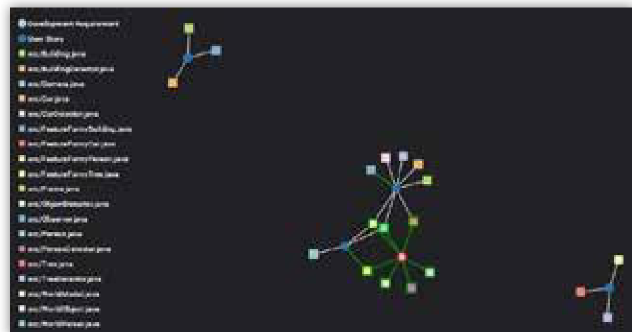
```
public final class WorldModel {

    public enum EventType { APPEARED }
    private ArrayList<Car> cars = new ArrayList<>();
    private ArrayList<Person> persons = new ArrayList<>();

    private HashMap<EventType, List<Observer>> eventObserverMap =
        new HashMap<>();

    private HashMap<Class<? extends WorldObject>, ArrayList<Wor-
ldObject>>
        typeObjMap= new HashMap();
    ...

    public void addObject(WorldObject obj) {
        var objList =
            typeObjMap.getOrDefault(obj.getClass(), new Array-
List<>());
        objList.add(obj);
        typeObjMap.put(obj.getClass(), objList);

    public void addCar(Car car) {
        ...
    }
    public void addPerson(Person person) {
        ...
    }
}
```

In (Abb. 5) ist eindeutig zu erkennen, dass die Abhängigkeitsstruktur komplizierter geworden ist. Dennoch stellt das Refactoring eine gravierende Architekturverbesserung dar, was sich



Abhängigkeitsstruktur - komplizierter nach dem Refactoring (Abb. 5)

beim Hinzufügen eines weiteren Features erkennbar macht. Im Netzwerk-Diagramm in (Abb. 6) sind zwei neue User-Stories aufgeführt. Darunter User-Story #4: Der Benutzer möchte vom System erkannte "Gebäude lustig dekorieren" (Lustige Gebäude) und eine entsprechende User-Story zu „Lustigen Bäumen". Es lässt sich sofort erkennen, dass die Klassen `WorldModel` und `WorldParser` keiner weiteren Anpassung für User-Story #4 und #5 (der blaue Kreis links oben und rechts unten) bedürfen. Es werden nur neue Code-Module für User-Story #4 hinzugefügt, wie zum Beispiel `FeatureFunnyBuildings.java`, `BuildingDetector.java` und `Building.java`. Sie sind nun recht straight forward zu implementieren, so dass deren Listings aus Platzgründen weggelassen wurden.



Feature-Kopplung - neue Features sind nicht mehr gekoppelt (Abb. 6)

Diese neuen Code-Module weisen keine Feature-Kopplung auf, denn sie tragen zu genau einer User-Story bzw. einem Feature bei. In dem Cluster in der Mitte ist auch das Development-Requirement mit dem Refactoring als rot eingerahmtes Issue markiert. Da Refactorings meistens systemübergreifend sind, wurden dafür viele der bereits bestehenden Code-Module angefasst. Daher sind Refactorings für eine Evaluierung der Feature-Kopplung nicht zu berücksichtigen. Aus diesen und anderen Gründen sollten Refactorings übrigens als eigene Issues erfasst werden[1].

Feature-Modularität und seine visuellen Muster

Das höhere Ziel der Softwareentwicklung besteht im Erreichen von Code-Modularität, soweit herrscht weitestgehend Einigkeit.

Eine bisher oft betonte Ausprägung ist die bereits erwähnte Qualität der Abhängigkeitsstruktur, die sich durch eine hierarchische und azyklische Ausprägung auszeichnet. Die Beschränkungen dieser Sichtweise wurden bereits dargelegt. Eine weitere Art Code-Modularität zu erfassen, ist sicherlich das Single-Responsibility-Prinzip (SRP), von dem zwei Formulierungen bekannt sind:

"Ein Modul sollte einem, und nur einem,
Akteur gegenüber verantwortlich sein."
Robert C. Martin[2]

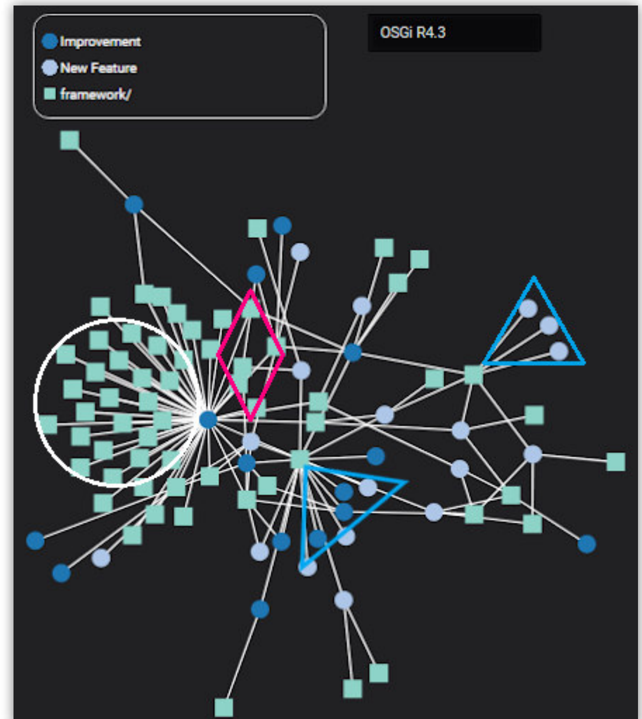
"A functional unit on a given level of abstractionshould
only be responsible for a single aspect
of a system's requirements. An aspect of requirements
is a trait or property of requirements,
which can change independently of other aspects."
Ralf Westphal[3]

Laut Robert C. Martin geht es beim SRP nicht nur um die einzelnen Klassen oder Funktionen, sondern vielmehr um, durch die Anforderungen eines Akteurs definierten Sammlungen an Funktionalitäten und Datenstrukturen. Ralf Westphal verweist auf den ähnlichen Punkt, dass sich das SRP auf Requirements bezieht. Bisher gibt es keine allgemein akzeptierte Fassung dieses Prinzips trotz seiner intuitiv eingängigen Aussage. Ralf Westphal bezieht sich eher auf eine eigenständige Eigenschaft, die mehreren Requirements zugrunde liegt. Robert C. Martin bezieht sich hingegen eher auf einen Satz von Funktionalitäten, die auf einen Akteur bzw. User zurückgehen. Eine pragmatische Ausprägung wäre:

„Ein Code-Modul sollte weitestgehend
zu einem Requirement, einer Funktionalität oder
einem Feature beitragen."

Analysen zeigten, dass Code-Module die dem zuwiderlaufen, nach dem Hinzufügen neuer Features oftmals von vielen Bugs und Wartungsaufwänden betroffen sind.

Die eingangs erwähnte Systemkomplexität, eine Messzahl die ausdrückt wo im System mit wesentlich mehr Aufwand zur Feature-Umsetzung zu rechnen ist, wird mit Kennzahlen der Feature-Modularität berechnet. Die Konzepte der Feature-Kopplung und Feature-Kohäsion lassen sich wieder anhand visueller Muster im Netzwerkgraphen erklären **(Abb. 7)**, hier am Beispiel des Open-Source-Projekts Apache-Felix. Wiederum werden dunkelblaue und hellblaue Kreise für die Issues vom Typ **`Feature`** und **`Improvement`** sowie türkise Rechtecke für die **`Code-Module`** genutzt. Feature-Kohäsion misst den Beitrag einer Datei zu verschiedenen Features. Je größer die Anzahl der Features zu denen eine Datei beiträgt, desto stärker verletzt sie das Single-Responsibility-Prinzip, und desto kleiner ist ihre Kohäsion. Im Netzwerkgraphen sind das visuelle Dreiecke, die in **(Abb. 7)** blau markiert sind. Dieses visuelle Muster wird auch



Feature-Modularität - visuelle Muster (Abb. 7)

Module-Tangle genannt. Tangle steht für Knäuel, Wirrwarr, was eben dieses Modul mit seinen Beiträgen zu mehreren Features/ Improvements auch darstellt. Für die Weiterentwicklung dieser Module muss erhöhter kognitiver Aufwand aufgebracht werden, um unbeabsichtigte Seiteneffekte zu vermeiden.

Die Feature-Kopplung wiederum misst die Überlappung oder Überschneidung von Features über Dateien hinweg. Viele Module tragen gemeinsam zu vielen Features auf eine chaotische Art und Weise bei. Ein Beispiel ist in **(Abb. 7)** als pinkes Trapez markiert. Dieses visuelle Muster wird als Diamond-Tangles bezeichnet. Diamanten deswegen, weil dies die architekturellen Hotspots des Systems darstellen. Feature-Modularität lässt sich auch dazu nutzen, um die technischen Schulden der Features, die Feature-Qualitätsschulden, im Code zu messen. Das ist auch ein Ansatz, um den grundsätzlichen Konflikt im Software-Engineering zwischen Features und Qualität der Software anzugehen.

Quellen:

1 Refactoring Issues: https://bit.ly/issues-i1
2 Robert C. Martin: "Clean Architecture: A Craftsman's Guide to Software Structure and Design"
3 Single-Responsibility-Principle – https://bit.ly/principle-p3