Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Masterarbeit

# Systematische Portierung zur langfristigen Weiterentwicklung von Software auf mehreren Plattformen anhand von Desmo-J

Gerrit Greiert
5greiert@informatik.uni-hamburg.de
Studiengang Wirtschaftsinformatik
Matr.-Nr.: 6829930

---

| | |
|---|---|
| Erstgutachter: | Prof. Dr. Matthias Riebisch |
| Zweitgutachter: | Dr. Philip Joschko |
| Betreuer: | Tilmann Stehle |

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Quelltextverzeichnis

# 1. Einleitung

Die Technologielandschaft für die Umsetzung von Desktop-Software und Codebibliotheken wandelt sich im Laufe der Zeit stetig. Dadurch kommen neue oder verbesserte Technologien auf den Markt, welche zusätzliche und aktuellere Plattformen ansprechen. Im gleichen Zuge kann der Support und die Weiterentwicklung von älteren Technologien eingestellt werden. Für in älteren Technologien implementierte Software treten durch diese Entwicklungen verschiedene Probleme auf. Zum einen verringert ein mangelnder Support einer Technologie mit der Zeit die Softwarequalität, da später aufgefundene Fehler und neue Herausforderungen nicht mehr behandelt werden. Zum anderen kann eine in einer älteren Technologie implementierte Software die neueren Plattformen aufgrund der technologischen Differenzen oftmals nicht ansprechen. Dies ist besonderes für langlebige Software wie Codebibliotheken problematisch, deren Kern idealerweise langfristig beibehalten und weiterentwickelbar sein soll. Um die langfristige Weiterentwicklung von Software auf mehreren Plattformen zu ermöglichen, kann in Erwägung gezogen werden, den vorhandenen Quellcode in eine aktuellere und für die neuen Plattformen geeignetere Programmiersprache zu konvertieren. Dies ist besonders interessant wenn dabei der Sprung von einer reinen Desktopanwendung zu einer Webanwendung vollzogen werden kann.

Codekonversion stellt jedoch besonders bei größeren Unterschieden zwischen Quell- und Zielsprache eine Herausforderung dar. Stehle et al. haben dazu eine Portierungsmethode für die Konvertierung von Mobilanwendungen entwickelt, die ein systematisches Rahmenwerk für Portierungsvorhaben vorgibt [SR17]. Durch die Konversion einer Software anhand der Portierungsmethode wird angestrebt die Ausgangsversion und die konvertierte Version als parallel weiterentwickelbare Codestränge zu erhalten. Dabei soll ein Traceability-Modell erzeugt werden, welches logisch zusammengehörige Codeartefakte zwischen den Codesträngen verknüpft und durch verbesserte Auffindbarkeit von Codeartefakten die parallele Entwicklung vereinfacht. Natürlich muss beachtet werden, dass zwischen mobilen Anwendungen und Desktopanwendungen, sowie Codebibliotheken, paradigmatische Unterschiede bestehen. Daher muss untersucht werden, inwiefern sich die vorgestellte Methode für die Anwendung auf nichtmobile Software eignet.

## 1.1. Zielsetzung

Im Rahmen der Masterarbeit soll untersucht werden, ob die von Stehle et al. vorgeschlagene Methode zur Portierung mobiler Anwendungen auf neue Plattformen auch für nicht mobile Software anwendbar ist. Als Fallbeispiel dient dabei die Portierung des in Java programmierten Simulationsrahmenwerks DESMO-J nach JavaScript. Dies ermöglicht die Integration des Frameworks in aktuelle, webbasierte Werkzeuge. Dazu müssen nach einer Einführung in verwandte Arbeiten und die fachlichen Grundlagen folgende Schritte durchgeführt werden:

- Toolrecherche zu verfügbaren Quellcode-Konvertern, Auswahl und zielgerichtete Anpassung
- Analyse des konzeptionellen und architekturbezogenen Änderungsbedarfes der Ausgangsversion bei der Portierung
- Durchführung der Portierung nach Stehle et al.
- Evaluation der portierten Software und der Portierungsmethode

Es soll eine auf JavaScript basierende Version des Simulationsframeworks DESMO-J entstehen, welche in webbasierten Applikationen Anwendung finden könnte. Dabei soll vor allem ein in sich funktionaler Teil der Software portiert werden, der es ermöglicht Simulationen auf einer Webplattform auszuführen.

Im Anschluss an dieses Kapitel wird in die Thematik eingeführt, indem verwandte Arbeiten auf dem Gebiet der Quellcodeportierung und der Nutzung von Traceability in der Softwareentwicklung vorgestellt werden. Auch wird betrachtet, inwiefern vorherige Arbeiten nützlich für das beschriebene Verfahren sind. Zudem sollen fachliche Grundlagen zur Portierung von Software, Traceability und der hier betrachteten Sprachen Java und JavaScript dargelegt werden. Auch wird die Portierungsmethode von Stehle et al. mit ihren Arbeitsschritten vorgestellt.

Ein wichtiger Aspekt eines Portierungsvorhabens ist das verwendete Portierungswerkzeug, also der Code-Konverter. Die Reife der vorhandenen Werkzeugunterstützung hat einen großen Einfluss auf die Automatisierbarkeit der Übersetzung. Daher wird eine Toolrecherche durchgeführt und beschrieben, welche die vorhandenen Java-zu-JavaScript-Konverter vorstellt und anhand von Auswahlkriterien vergleicht. Anhand dieser Analyse wird eine Auswahl getroffen, welcher Konverter hier verwendet werden soll und welche Anpassungen im Rahmen der Aufgabenstellung vorgenommen werden müssen. Neben der Werkzeuganalyse muss des Weiteren der konzeptionelle und architektonische Änderungsaufwand im Code bei der Portierung analysiert werden. Dazu müssen grundlegend die Gemeinsamkeiten und Unterschiede zwischen Ausgangssprache Java und Zielsprache JavaScript betrachtet werden. Aufgrund der Sprachunterschiede werden manuelle Änderungen vor oder nach der Konvertierung nötig, sowie automatische Anpassungen die durch das Konvertierungswerkzeug vorgenommen werden. Idealerweise soll trotz Änderungsbedarf eine möglichst strukturgleiche Version entstehen. Daher werden Änderungsbedarfe und Lösungsalternativen aufgezeigt.

Auf Basis dieser Analysen kann anschließend die Portierung durchgeführt werden. Diese wird in Kapitel 6 beschrieben. Dabei werden entlang der Portierungsmethode eine Abhängigkeitsanalyse zu Codebibliotheken und Plattform-APIs durchgeführt und eine zu portierende Teilfunktionalität aus der Gesamtsoftware herausgelöst. Besonderes Augenmerk liegt bei der Portierung auf der Beurteilung der Methode, also ihrer Stärken und Schwächen zum Beispiel hinsichtlich der Entscheidungsunterstützung im Portierungsprozess, der erreichten Strukturgleichheit und der Einfachheit der darauffolgenden parallelen Weiterentwicklung. Diese Erkenntnisse fließen im Anschluss in die Evaluation ein, wo auch die strukturelle Gleichheit der Versionen und der Portierungsaufwand bewertet wird.

# 2. Verwandte Arbeiten

Die Thematik der Portierung von Software auf andere Plattformen wird in der Literatur von verschiedensten Seiten betrachtet. Besonders die gegebenen Umstände im Bereich der Mobilanwendungen haben aufgrund der unterschiedlichen Betriebssysteme und verwendeten Programmiersprachen verstärkte Aufmerksamkeit auf die Portierung von Software gelenkt. Durch die unterschiedlichen technischen Grundlagen von Mobilplattformen wie iOS oder Android wird die Bedeutung von Crossplatform-Software hervorgehoben. Crossplatform-Software oder auch Multiplattform-Software beschreibt dabei eine Software, die in verschiedenen Versionen für mehr als eine Plattform vorliegt [BH06]. Dies ist nötig, wenn besagte Software mehrere relevante Plattformen des Marktes abdecken soll.

Im Rahmen der Crossplatform-Entwicklung von Software klassifizieren El-Kassas et al. die verwendeten Ansätze in sechs Kategorien: Compiler-, Komponenten-, Interpretations-, Modellierungs- oder Cloud-basiert, sowie als sechste Art eine Hybridmischung aus den fünf differenzierten Ansätzen [EKAYW17]. Der Compiler-basierte Ansatz stützt sich auf die Verwendung von Codekonvertern, um eine Hochsprache oder deren Bytecode in eine andere Hochsprache zu konvertieren. Der konvertierte Code kann im Anschluss auf der neuen Plattform verwendet werden. Die Autoren identifizieren als Vorteile dieses Ansatzes den hohen Grad an Wiederverwendbarkeit von Legacy-Code, also des Codes der auf der ursprünglichen Plattform zur Anwendung kam. Zudem entsteht durch die Sprachkonversion Code in der für die Zielplattform nativen Programmiersprache, was sich positiv auf die Performance auswirkt. Als Herausforderung wird jedoch das Finden bzw. die Entwicklung eines adäquaten Crosscompilers oder Transpilers genannt, da das Mapping von Sprachkonzepten und API-Aufrufen komplex ist und zudem bei aktiv entwickelten Plattformen stetige Anpassung erfordert. Um dieses Problem zu mitigieren wird bei der Codekonversion oft darauf verzichtet sehr stark plattformabhängige Elemente wie z.B. die Benutzeroberfläche zu übersetzen. Stattdessen wird ein plattformübergreifender Kern ermittelt und konvertiert und plattformabhängige Teile nachträglich manuell in der nativen Sprache implementiert. Diese Unterteilung in verschiedene Teile der Software wird auch im komponentenbasierten Ansatz verfolgt. Dabei wird die Software in mehrere Komponenten mit klar definierten Schnittstellen unterteilt. Diese Schnittstellen werden auf allen Plattformen gleichartig definiert und eingehalten. Die genaue Implementation der Komponente bleibt hinter der Schnittstelle verborgen. Somit wird eine Strukturgleichheit zwischen den Versionen der Software auf verschiedenen Plattformen erreicht, wodurch der Support und die Wartung vereinfacht wird. Der Interpretationsansatz fasst das Ausführen der Software in einer virtuellen Maschine auf der neuen Plattform oder in einer plattformübergreifenden Umgebung wie dem Webbrowser zusammen. Es wird kein nativer Code geschrieben, stattdessen kommt die Originalversion oder eine webbasierte Fassung zum Einsatz. Dadurch kann es zu Einbußen bei der Performance kommen, allerdings kann der anfallende Aufwand minimiert werden, insbesondere wenn die Software von Anfang an als Webanwendung konzipiert wurde. Bei dem modellbasierten

Ansatz spielt die Anwendung von Codegeneratoren eine große Rolle. Der Ansatz basiert auf der Modellierung einer plattformunabhängigen Repräsentation der Software in einer Modellierungssprache wie beispielsweise UML oder einer DSL (domänenenspezifischen Sprache). Aus diesem Modell kann über Codegeneratoren nativer Quellcode für alle Zielplattformen generiert werden. Je nach Detailgrad des Modells und Mächtigkeit der Generatoren könnte sogar eine lauffähige Software generiert werden, an der keine manuellen Anpassungen vorgenommen werden müssen. El-Kassas et al. sehen aber gerade in der dabei anfallenden Komplexität Limitationen für diesen Ansatz. Zudem ermöglicht er nicht die Wiederverwendung bereits vorhandenen Quellcodes. Als fünfte Kategorie wird der cloudbasierte Ansatz erwähnt, bei dem die eigentliche Software gar nicht auf einer bestimmten Plattform läuft, sondern in der Cloud. Die Endgeräte dienen lediglich als Anzeigegeräte. Die Crossplatform-Entwicklung beschränkt sich hierbei auf eine in der Cloud operierende Version. Natürlich ist eine ständige Verbindung zur Cloud zur Verwendung der Software nötig.

Oft ergibt sich in der Crossplatform-Entwicklung eine Mischung der verschiedenen Ansätze, um die Schwächen der einzelnen Ansätze abzufangen und Stärken zu verbinden [EKAYW17]. Auch die Portierungsmethode von Stehle et al. lässt sich als ein solcher Hybridansatz einordnen. Ähnliche oder gleichartige Kategorisierungen finden sich auch bei Raj et al. [RT12], Ribeiro et al. [RS12], Heidkötter et al. [HHM12] und Perchat et al. [PDL13]. Ohrt et al. unterscheidet zwischen integrierten und nicht-integrierten Anwendungen. Erstere umfassen nativ geschriebene oder interpretierte Anwendungen wohingegen letzte plattformübergreifende Anwendungen wie Web-Apps beinhalten [OT12]. Der Crossplatform-Ansatz in Form einer Portierungsmethode für Mobilanwendungen nach Stehle et al. [SR17] soll in dieser Arbeit angewendet und untersucht werden. Daher wird sie in Abschnitt 3.2 näher erläutert. An dieser Stelle sollen jedoch andere Crossplatform-Ansätze aus der Literatur erwähnt werden.

Ein weiterer Ansatz ist das von El-Kassas et al. vorgeschlagene „Integrated Cross-Platform Mobile Development“ (ICPMD). Dabei handelt es sich einen Hybridansatz aus compiler-, modell- und komponentenbasierten Methoden. Der vorgeschlagene Ansatz transformiert eine nativ für eine Plattform implementierte Software in eine auf einer anderen Plattform lauffähigen Version. Dabei kann der bestehende Code der Originalversion wiederverwendet werden. Zudem erlaubt der Ansatz die Transformation sowohl von Geschäftslogik als auch der Benutzeroberfläche, wodurch vollständige Apps entstehen. So müssen Entwickler im Idealfall keine weiteren Sprachen und Technologien erlernen und können in ihrer präferierten Sprache entwickeln. Die Versionen für andere Plattformen werden durch die Methode transformiert. ICPMD basiert auf der Übersetzung des gesamten Quellcodes und der verwendeten Ressourcen in eine XML-basierte Zwischensprache, die als abstraktes Modell der Software dient. Bei der Transformation zum abstrakten Modell werden plattformspezifische Elemente aufgelöst und als Referenz abgespeichert. Aus dem abstrakten Modell können Codegeneratoren nun Versionen für andere Plattformen in deren nativen Sprache generieren. Die Aufrufe der Plattform-API können über die Referenzen sinngemäß aufgelöst werden. Die Implementation der Codegeneratoren erlaubt es die Vorteile und Eigenheiten einer Plattform auszunutzen, ohne die vorgegebene Struktur der Originalversion berücksichtigen zu müssen. Dennoch wird durch die Beibehaltung der Schnittstellen eine gewisse Strukturgleichheit zwischen den Plattformen erreicht. Als Limitation ihres Ansatzes identifizieren die Autoren vor allem die Komplexität die durch die Erweiterung auf weitere Plattformen auftritt. Sowohl die Entwicklung der

Konverter zum abstrakten Modell als auch der Codegeneratoren erfordert für eine zufriedenstellende Abdeckung der Sprachkonzepte viel Aufwand. Des weiteren besteht das Problem, das bestimmte Konzepte auf manchen Plattformen nicht existieren und somit nicht gleichförmig umgesetzt werden können. Als Beispiel wird das Verhalten des „Zurück"-Knopfes unter Android genannt, welches in iOS nicht existent ist [EKAYW14]. Im Rahmen von Weiterentwicklungen des ICPMD wurde die Quellcodetransformation durch die Anwendung von Quellcode-Mustern auf Basis von regulären Ausdrücken verbessert. Bestimmte Muster die in einer Sprache gefunden werden, können durch in einer Datenbank hinterlegte Mappings direkt Mustern in der Zielsprache zugeordnet werden [EKAYW16].

Ein ähnlicher Ansatz mit dem Namen „XMLVM" wurde von Puder et al. beschrieben. Dieser wird jedoch als rein compilerbasierter Ansatz klassifiziert. Dabei kommt ein Cross-Compiler zum Einsatz, welcher den Bytecode, also bereits kompilierten Code einer Programmiersprache in eine andere Sprache transformiert. Alternativ dazu wären Transpiler zu nennen, welche eine nicht kompilierte Hochsprache in eine andere umwandeln. Die Autoren sehen in der Verwendungen des Bytecodes jedoch verschiedene Vorteile: Zum einen sind Bytecodes einfacher zu parsen bzw. maschinell zu erfassen, da Sprachkonstrukte der Hochsprache bereits auf Low-Level-Anweisungen reduziert wurden. Zum anderen führt der Compiler bereits Optimierungen auf dem Code aus, wodurch die Effizienz des Generats erhöht wird. Diese Vorteile treffen zumindest auf die in dem Paper verwendete Ausgangssprache Java zu. Die Autoren fokussieren sich wiederum auf die Portierung von mobilen Apps von der Android-Plattform auf iOS. Der Java-Code der Android-Plattform wird dazu zuerst mit dem regulären Java-Compiler kompiliert um den Bytecode zu erhalten. Ein eigens für den Portierungsvorgang geschriebener Cross-Compiler transformiert den Bytecode anschließend in eine Repräsentation des Bytecodes in XML. In dieser Abstraktionsebene können nun automatisch Anpassungen vorgenommen werden, die die Generierung von Objective-C-Code für die Zielplattform iOS vorbereiten. Dazu gehört zum Beispiel die Unterstützung des unter Objective-C anders funktionierenden Speichermanagements ohne den bei Java vorhandenen Garbage Collector. Abschließend wird die angepasste XML-Repräsentation zur Generierung nativen Objective-C-Codes verwendet. Als Problem bleibt jedoch das Mapping von Aufrufen der Plattform-API bestehen. Dazu schlagen die Autoren zwei Lösungsvorschläge vor. Eine Möglichkeit ist die manuelle, nachträgliche Anpassung der Aufrufe im generierten Objective-C-Code. Die Andere wäre die Einführung einer Kompatibilitäts-Bibliothek auf iOS-Seite. Diese Bibliothek würde im Code verbliebene Aufrufe zur Android-API aufgreifen und auf native Aufrufe zur iOS-API umleiten. Beide Ansätze erfordern eine eingehende Analyse der APIs, zudem entsteht durch das Einfügen einer Bibliothek ein großer Overhead an eventuell nicht genutztem Code. Allerdings ist eine Kompatibilitäts-Bibliothek für folgende Portierungsprojekte wiederverwendbar. Die unzureichende Abdeckung durch API-Mappings wird von den Autoren als größte Limitation ihres Ansatzes betrachtet [Pud10].

In der Literatur wird vielfach darauf hingewiesen, dass nur die Verwendung der für die Zielplattform nativen Programmiersprache und API-Aufrufe eine optimale Performance und umfassende Nutzbarkeit der Plattformfeatures ermöglicht (vgl. z.B. [Pud10], [EKAYW17], [PDL13]) Perchat et al. haben aus diesem Grund einen komponentenbasierten Crossplatform-Ansatz vorgeschlagen, der nativ implementierte Softwarekomponenten mit einer plattformübergreifenden Struktur verknüpft. So soll eine einheitliche Ebene für die Entwicklung der Softwarestruktur bzw. -Architektur geschaffen

werden. Diese Struktur gibt die Schnittstellen zu einzelnen Komponenten vor, welche gekapselte Einzelfunktionen der Software implementieren. Als Repräsentation für diese einheitliche Struktur wählen die Autoren eine eigens entwickelte, deklarative Sprache auf Basis von Java-Annotationen. Die definierten Komponenten mit allen in der Schnittstelle definierten Methoden werden nativ für alle Zielplattformen implementiert. Dabei wird der Entwicklung durch einen Komponenten-Generator unterstützt, der Code-Templates für die definierten Komponenten bereitstellt. So kann Strukturgleichheit für die plattformspezifischen Implementationen sichergestellt werden, was Vorteile bei der weiteren parallelen Evolution und Wartung bietet. Gleichzeitig können alle Vorteile einer nativen Entwicklung ausgenutzt werden [PDL13].

Die oben erwähnten Ansätze wurden primär für mobile Applikationen entwickelt, obwohl die angewandten Techniken oftmals auch für andere Arten von Software angepasst werden könnten. Puder et al. haben einen Ansatz zur Portierung von Desktopsoftware zu Webanwendungen vorgeschlagen. Als Beispiel wählen sie, analog zum Untersuchungsgegenstand dieser Arbeit, die Portierung von Javacode zu im Browser lauffähigem JavaScript. Als Anwendungsfall sehen sie hierbei vor allem die Portierung von komplexen Codebibliotheken, wie etwa Frameworks für Physiksimulationen. Aufgrund der Komplexität ist eine Wiederverwendung von bereits bestehenden Implementierungen solcher Bibliotheken der Neuimplementierung vorzuziehen. Selbiger Vorteil würde auch für das in dieser Arbeit betrachtete Simulationsframework DESMO-J gelten. In ihrem Paper entwerfen die Autoren eine Toolchain aus mehreren Konvertern. Zuerst kommt der oben angesprochene Ansatz „XMLVM“ zum Einsatz, welcher Java-Bytecode in die Sprache C konvertiert. Die aus der Verwendung von Bytecode als Input entspringenden Vorteile wurden ebenfalls bereits oben erwähnt. Als zweiten Schritt wird der Cross-Compiler „Emscripten“ verwendet, der den aus C-Code kompilierten Bytecode „LLVM“ zu JavaScript übersetzt. Emscripten verfügt zudem über in JavaScript implementierte Versionen von C-Bibliotheken, die bei Bedarf eingebunden werden. Die Performance des so entstandenen JavaScript-Codes wurde über Benchmarks mit der des Portierungswerkzeuges GWT verglichen. GWT (Google Web Toolkit) ist ein von Google entwickelter Transpiler, der Java-Code direkt zu JavaScript übersetzt. In mehr als 60% der Benchmarks produzierte die vorgeschlagene Toolchain performanteren JavaScript-Code als GWT. Zudem sehen die Autoren in der Verknüpfung von Konvertern ein konkurrenzfähiges Modell gegenüber direkten Konvertern. So deckt ihre Lösung einen weitaus größeren Teil des JRE für mögliche Portierungen ab als GWT. Auch konnten durch einige mit wenig Aufwand verknüpfte Änderungen in den bestehenden Cross-Compilern signifikante Performanceverbesserungen erreicht werden [PWZ13].

Einen ähnlichen Ansatz der Portierung von Java zu JavaScript verfolgen Leopoldseder et al., mit dem Unterschied, dass ihr Ansatz JavaScript aus dem Zwischencode (Intermediate Representation) des Java-Compilers Graal generiert. Dieser Zwischencode dient als Zwischenschritt in der Kompilierung der Hochsprache zu Maschinensprache. Statt das von Graal vorgesehene Backend zur Generierung von Java-Bytecode zu verwenden, ersetzen die Autoren es durch eine eigene Implementierung zur Generierung von JavaScript. Dabei besteht die Möglichkeit auf dem Zwischencode Anpassungen und Optimierungen zugunsten der Zielsprache JavaScript auszuführen. Dies ist nötig, da nicht alle Sprachkonzepte von Java gleichartig in JavaScript umsetzbar sind. Auch diese Arbeit setzt sich in Kapitel 5 mit dieser Problematik auseinander. Zuzüglich zu Optimierungen wird auch eine Rekonstruktion des Kontrollflusses auf dem normalerweise unstrukturierten Zwischencode ausgeführt, um besser strukturierten, lesbaren und vor allem auch wartbaren JavaScript-Code zu

erhalten. Die Evaluation dieses Ansatzes zeigt, dass er durchaus funktionalen und performanten JavaScript-Code erzeugt, welcher jedoch bei auf vielen numerischen Berechnungen basierenden Aufgaben deutlich langsamer ist als handgeschriebenes JavaScript [LSWM15].

Neben den hier erwähnten Werkzeugen zur Portierung von in Java geschriebener Software zu JavaScript, gibt es noch viele weitere, die als Teil dieser Arbeit in Kapitel 4 verglichen werden. Dazu gehören unter anderem JSweet [JSw17], GWT [Goo17b], TeaVM [Tea17] oder Dragome [Dra17].

Als Beispiel für eine Portierungsprojekt von Java zu JavaScript ist die Bibliothek „libphonenumber“ von Google zu nennen, welche das Parsing, Formatieren und Validieren von internationalen Telefonnummern ermöglicht. Diese ursprünglich in Java implementierte Bibliothek wurde anhand eines eigens entwickelten Portierungsablaufes zu JavaScript und anderen Sprachen übertragen. Änderungen oder Erweiterungen werden in der Java-Version implementiert. Anschließend wird eine neue JavaScript-Version aus der neuen Codebasis erzeugt. Dabei basiert die Portierung jedoch größtenteils auf der manuellen Implementierung der Änderungen in JavaScript. Im Portierungsablauf ist dokumentiert, welche Stellen der Java-Version mit welcher Entsprechung im JavaScript-Code übereinstimmt [Goo18]. Diese Art von Traceabilty hilft dabei änderungsbedürftige Codeelemente zu identifizieren.

Wie bereits oben erwähnt kann die Wiederverwendung von bestehendem Code gegenüber einer Neuimplementierung vorteilhaft sein. In der Literatur werden eine höhere Produktivität bei der Portierung, Zeit- und Kosteneinsparungen als Vorteile genannt. Zudem verhindert die Wiederverwendung die Einführung neuer Fehler in der Geschäftslogik. Daraus leitet sich auch ab, dass Software die besonders wiederverwendbar ist, auch besonders zuverlässig langfristig verwendet werden kann. Ein entsprechender Softwareentwurf erhöht die Lebensspanne der Software und vereinfacht die Wartung und mögliche Migration. Cybulski definiert einige Charakteristiken von Softwareartefakten, die einen Beitrag zu erhöhter Wiederverwendbarkeit leisten. Seine Arbeit schlägt des Weiteren die Einrichtung einer Reuse-Library aus wiederverwendbaren Softwareartefakten vor.

Ein wiederverwendbares Artefakt sollte keine Abhängigkeiten zu einer bestimmten Programmiersprache haben, um auch plattformübergreifend wiederverwendbar zu sein. Dies erfordert eine adäquate Abstraktion von sprachspezifischen Funktionen. Auch was die Funktionalität des Artefakts angeht sollte abstrahiert werden. Der Nutzen des Softwareartefakts sollte in möglichst vielen verschiedenen Problembereichen integrierbar sein. Dementsprechend müssen für jedes Artefakt dessen Funktionalitäten und Limitationen klar definiert werden. Abhängigkeiten zu anderen Artefakten werden auf ein Minimum beschränkt bzw. vermieden. Diese Designvorgaben dienen dazu, den Transfer des Artefakts in andere Systeme und auf andere Plattformen zu vereinfachen, sowie die Komposition von Artefakten in komplexere Artefakte zu unterstützen [Cyb96].

Viele dieser Entwurfsempfehlungen finden sich auch in komponentenbasierter Softwarearchitektur wieder. Je mehr das Konzept der Wiederverwendbarkeit bereits im Softwareentwurf umgesetzt wurde, desto einfacher gelingt eine spätere Portierung.

# 3. Fachliche Grundlagen

In diesem Kapitel sollen die Grundlagen für das Verständnis der weiteren Ausführungen geschaffen werden. Dazu werden sowohl der Begriff der Portierung von Software, als auch die in dieser Arbeit verwendete Portierungsmethode erläutert. Zudem wird das Konzept der Traceability erklärt und eine Einführung in die im Folgenden betrachteten Programmiersprachen und die im Fallbeispiel genutzte Software DESMO-J gegeben.

## 3.1. Portierung von Software

Der Begriff Portierung bezeichnet im Kontext von Software den Vorgang, eine Software von einer Umgebung in eine andere zu überführen. Im Regelfall beinhaltet dies, eine bestehende Software auf einer bisher nicht unterstützen Plattform lauffähig zu machen. Laut ISO-Standard 25010 sagt die Portabilität einer Software aus, wie effektiv und effizient eine Software auf andere Plattformen übertragbar ist [ISO11]. Eine ähnliche Bedeutung hat auch der Begriff Migration. Bei einer Migration geht es darum, Legacy-Systeme auf andere Plattformen zu übertragen, um Anpassungen auf veränderte Geschäftsbedingungen oder Anforderungen durchführen zu können. Eine Migration einer Software ermöglicht die Weiterverwendung der existierenden Daten und verwendet bestehenden Code, anstatt eine Neuimplementation vorauszusetzen [Wag14]. Im Gegensatz zu Portierung wird bei einer Migration jedoch meistens davon ausgegangen, dass die Legacy-Software nach erfolgreicher Migration nicht weiter entwickelt oder gewartet wird. Die Fortführung von Legacy-Software und portierter Software wird in Migrationsmethologien nicht betrachtet [SR17]. Aufgrund dieses Bedeutungsunterschieds wird im Folgenden ausschließlich der Begriff Portierung verwendet, da die parallele Weiterentwicklung beider Versionen explizit Ziel der hier durchgeführten Portierung ist.

Portierung ist nur sinnvoll, wenn der Aufwand und die Kosten geringer sind als die einer Neuentwicklung. Aus diesem Grund sollte ein maximaler Automatisierungsgrad des Portierungsprozesses angestrebt werden, um den manuellen Aufwand gering zu halten [TV00]. Einen wichtigen Beitrag zur Senkung des Portierungsaufwandes leistet auch die Wiederverwendung von bestehendem Code. Unter Wiederverwendbarkeit versteht man in der Softwareentwicklung den Grad der Verwendbarkeit eines Softwareartefakts in anderen Zusammenhängen, wie zum Beispiel in anderen Softwaresystemen, auf anderen Plattformen oder als Baustein in anderen Artefakten [ISO11]. Zur Entwicklung einer Version für eine neue Plattform oder deren Wartung werden demnach bestehende Softwareartefakte aus der alten Version herangezogen [TGK+04]. Demnach sollte bei der Portierung eine Maximierung der Wiederverwendung zur Minimierung des Aufwandes angestrebt werden. Je nach Portierungsmethode und verfügbaren Werkzeugen kann eine dafür adäquate Abstraktionsebene vom Legacy-Code gewählt werden. Idealerweise unterstützt die Legacy-Version Wiederverwendbarkeit bereits, wie in Kapitel 2 beschrieben, durch ihre Architektur [Cyb96].

## 3.2. Portierungsmethode nach Stehle et al.

In diesem Abschnitt soll die Portierungsmethode vorgestellt werden, die wie in Abschnitt 1.1 beschrieben, verwendet und evaluiert werden soll. Dabei handelt es sich um ein systematisches Vorgehen zur Portierung einer Software, damit weitere Plattformen erschlossen und eine anschließende synchronisierte Weiterentwicklung der Softwareversionen ermöglicht werden können. Portierung einer Software auf neue Plattformen und weiterführende synchronisierte Entwicklung sind einer kompletten Neuimplementation für die neue Plattform vorzuziehen, da die daraus resultierende Gleichheit der Struktur und Geschäftslogik eine parallele Entwicklung und Wartung vereinfacht. Des Weiteren sind die aus der Wiederverwendung von Code gewonnenen ökonomischen Vorteile nicht zu vernachlässigen. Somit basiert die Methode im Gegensatz zu anderen Ansätzen auf der Portierung von bereits bestehendem Code auf neue Plattformen, anstatt auf einer plattformübergreifenden Neuentwicklung bzw. Reimplementation.

Die Vorgehensweise nach Stehle et al. besteht aus drei verschiedenen Phasen mit mehreren Schritten, die im Folgenden vorgestellt werden und schematisch in Abbildung 3.1 dargestellt sind.

Zu Beginn muss die zu portierende Funktionalität der vorliegenden Software identifiziert werden. Es muss also geprüft werden, welche Funktionalitäten auf der Zielplattform verfügbar sein sollen und ob diese auf derselben umsetzbar sein werden. Die nach diesem Schritt ausgeschlossenen Funktionalitäten werden im Portierungsvorgang nicht weiter betrachtet. Für die ausgewählten Funktionalitäten muss geprüft werden welche Artefakte diese implementieren, testen und dokumentieren.

1. In der ersten Phase der Portierungsmethode werden die Abhängigkeiten der identifizierten Artefakte zur Ausgangsplattform analysiert. Die zu betrachtenden Abhängigkeiten umfassen solche zum Betriebssystem, zur Programmiersprache, zu externen Bibliotheken und zu Elementen die im ersten Schritt von der Portierung ausgeschlossen wurden.

   Dieser erste Schritt umfasst die Bestimmung des Crossplatform-Kerns und der Plattform-Mappings. Der Crossplatform-Kern bezeichnet die Menge an Elementen die keine oder nur wenige Abhängigkeiten zur ursprünglichen Plattform aufweisen und somit mit Konvertierungswerkzeugen übersetzt werden können. Dazu muss ein Überblick über die verfügbaren Werkzeuge vorhanden sein, um festlegen zu können wie umfangreich der Crossplatform-Kern definiert sein kann. Je mehr Abhängigkeiten durch das Werkzeug unterstützt und für die neue Plattform aufgelöst werden können, desto umfangreicher kann der automatisch zu portierende Teil gefasst werden. Durch vom Konverter unterstützte Plattform-Mappings können Abhängigkeiten zur Ausgangsversion automatisch durch Entsprechungen zur Zielseite ersetzt werden. Des Weiteren muss analysiert werden, inwiefern Refactorings des Codes und der Architektur dabei helfen können eine Portierung zu unterstützen. Plattformabhängigkeiten können beispielsweise durch die Verwendung von bestimmten Entwurfsmustern isoliert werden, wodurch der Umfang des automatisch konvertierbaren Crossplattform-Kerns zunimmt. Stehle et al. schlagen dazu einige transformationsvorbereitende Refactorings wie die Umsetzung von Entwurfsmustern wie *Adapter* oder *Generation Gap* vor. Idealerweise würde die Umsetzung dieser Refactorings durch automatische Werkzeuge unterstützt.

Die ausgewählten Funktionalitäten werden in mehreren Teilen iterativ portiert. Daher wird im folgenden Schritt der Phase eine Teilfunktionalität abgespalten. Jede Teilfunktionalität soll dabei weiterhin ausführbar sein und eigenständig einen Nutzen erfüllen. Solche Teilfunktionalitäten bieten den Vorteil, dass sie und auch die Anwendbarkeit des Werkzeugs frühzeitig getestet werden können. Abhängigkeiten zu anderen Teilfunktionalitäten werden zeitweilig durch Dummy-Implementationen ersetzt.

2. In der zweiten Phase wird nun der Crossplatform-Kern der betrachteten Teilfunktionalität per Werkzeug portiert. Idealerweise sollte das Portierungswerkzeug auch die automatische Generierung von Trace Links zwischen Input- und Outputartefakten unterstützen. Die nicht automatisch portierbaren Elemente werden manuell für die Zielplattform implementiert. Um vollständige Traceability zu erhalten, müssen die manuell reimplementierten Code-Elemente auch mit ihren Pendants in der Originalversion verknüpft werden. Dies ist durch manuelle Anpassung des Traceability-Modells oder Werkzeugunterstützung auf Basis von Traceability Recovery möglich.

   Ein weiterer Aspekt dieser Phase ist die Evaluation der Qualität der portierten Teilfunktionalität und des Portierungsaufwands. Letztendlich ist eine Portierung nur sinnvoll, wenn deren Aufwand unter dem einer Neuimplementierung bleibt. Allerdings müssen auch die durch die erreichte Strukturgleichheit auftretenden Vorteile für die Weiterentwicklung in diese Überlegung einbezogen werden. Der Portierungsaufwand kann gesenkt werden, indem ein höheres Abstraktionslevel gewählt wird. So können Code-Elemente mit sehr vielen Abhängigkeiten nur als Interface portiert werden, deren Implementation anschließend manuell in der Zielsprache erfolgt. Die erste und zweite Phase der Methode werden so lange wiederholt, bis alle ausgewählten Funktionalitäten portiert und zusammengefügt wurden.

3. Abschließend ist nun eine synchronisierte Evolution sowohl der Ausgangsversion als auch der portierten Version möglich. Die Traceability zwischen den Versionen ermöglicht bei Änderungen in einer Version ein besseres Auffinden von zu ändernden Stellen in der Anderen. Des Weiteren sind bei vorhandener Unterstützung durch Werkzeuge plattformübergreifende Refactorings möglich [SR17].

Die im Folgenden auftretenden Erwähnungen einer Portierungsmethode beziehen sich, falls nicht anders kenntlich gemacht, auf die hier vorgestellte Methodik.

## 3.3. Traceability

Cleland-Huang et al. beschreiben Traceability als das Potenzial, Traces zu erzeugen und zu nutzen. Dabei geht es primär um die Rückverfolgbarkeit von zusammengehörigen Artefakten. Ein Beispiel dafür wäre die Identifikation von zu Anforderungen gehörenden Stakeholderwünschen, Designartefakten oder Testfällen. Auch die Zuordnung von Dokumentationstexten zu Programmfunktionen gehört dazu. Ein Trace besteht daher aus einem Quellartefakt, einem Zielartefakt und einem Trace Link, welcher die beiden Artefakte verbindet. Trace-Artefakte können beispielsweise eine oder mehrere Anforderungen, UML-Diagramme, Quellcodedateien oder auch Personen sein. Die Verknüpfung von Artefakten über einen Trace Link kann verschiedene Bedeutungen haben. So kann

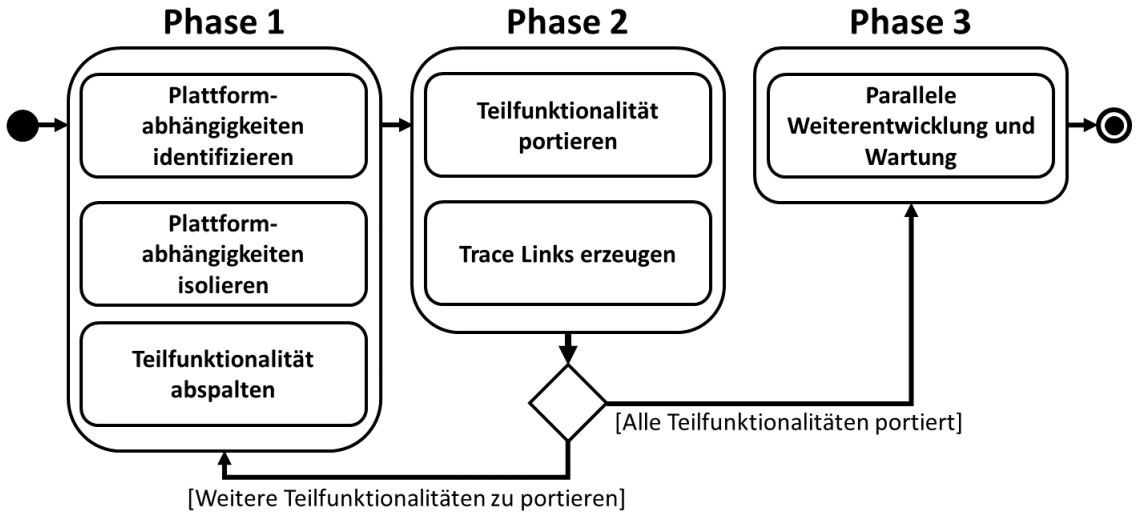


Abbildung 3.1.: Abstrakte Vorgehensweise bei der Portierung von Code nach Stehle et al. [FS16]

ein Quellartefakt ein Zielartefakt u.a. implementieren, testen, verfeinern oder ersetzen. Traceability wird verwendet um zugeordnete Artefakte zu finden. Dieser Vorgang wird als Tracing bezeichnet. Weitere der Traceability zugeordnete Aktivitäten sind das Identifizieren von relevanten Artefakten und Verknüpfungen, deren Speicherung und eine auf den Verwendungszweck zugeschnittene Repräsentation [GCHH+12].

Im Kontext dieser Arbeit soll Traceability zwischen konzeptuell gleichen Codeelementen in verschiedenen Implementationen einer Software hergestellt werden. Bei Code-Elementen kann es sich um unterschiedliche Sprachkonstrukte handeln, welche unter Umständen weitere Elemente enthalten. Beispielsweise enthält eine Klasse u.a. Methoden und Attribute. Methoden enthalten wiederum Parameter und Referenzen auf Attribute mit denen sie arbeiten. Über Tracing der gefundenen Trace Links können bei einer synchronisierten Weiterentwicklung beider Implementationen betroffene Stellen identifiziert werden.

### 3.3.1. Traceability Modell

Um Traceability im oben erläuterten Anwendungskontext greifbar und konkret anwendbar zu machen, muss es ein Modell geben, welches Typen von Codeelementen und Trace Links dazwischen abbilden kann. Ein solches Modell wurde für ein Projekt entwickelt, das dieser Arbeit vorangegangen ist [ABF+17]. Dabei ging es darum, Trace Links zwischen einer Java-Implementation und einer automatisch konvertierten C#- oder Swift-Implementation festzuhalten. Ein ähnlicher Fall ist auch für diese Arbeit relevant, da auch hier eine Codekonversion (von Java zu JavaScript) stattfinden soll. Da das Modell jedoch für eine möglichst vielseitige Anwendung konzipiert wurde und zudem programmiersprachenunabhängig ist, kann es hier problemlos wieder eingesetzt werden.

Die notwendige Datenstruktur wurde in Java implementiert und stellt Klassen zur Verfügung um Trace Links zwischen Code-Artefakten plattformunabhängig festzuhalten. Um die mögliche Schachtelung der verschiedenen Arten von Codeelementen (z.B. Attribute in Methoden) abzubilden

wurde eine rekursive Datenstruktur gewählt, in der jedes Element durch ein eigenes Traceability-Modell repräsentiert wird, welches wiederum Submodelle enthalten kann. Für die im Projekt verfolgte Konvertierung von Quellcode wurde ein Traceability-Modell für das zu übersetzende Element, in der Regel eine Klasse, vom Konverter erzeugt. Dieses enthält der Klassenstruktur entsprechend Submodelle für auftretende Subelemente. Die Verknüpfung zu den übersetzten Elementen wird durch Trace Links realisiert, welche Traceability Pointer enthalten. Ein Traceability Pointer zeigt auf ein bestimmtes Element im Modell. Durch die Zusammenführung von einem Pointer auf Seite der Originalversion und einem auf der Seite der portierten Implementation in einem Link kann eine Verknüpfung erzeugt werden. Verschiedene Code-Elemente werden durch verschiedene Arten von Pointern unterschieden, welche jedoch alle von der Oberklasse `TraceabilityPointer` erben. Die jeweiligen Unterklassen enthalten zusätzliche Attribute, die helfen das Element zu beschreiben. Das bisherige Modell kann durch die Implementation weiterer Pointer erweitert werden, um nicht abgebildete Sprachkonstrukte aus verschiedenen Programmiersprachen abzudecken. Somit ist eine Erweiterung durch Pointer für JavaScript im Anwendungskontext dieser Arbeit möglich.

Die Implementierung des Traceability-Modells enthält zudem Import- und Export-Klassen um die Datenstruktur in ein XML-Format zu externalisieren und wieder einlesen zu können [ABF+17].

## 3.4. Die Programmiersprachen Java und JavaScript

Dieser Abschnitt soll einen kurzen Überblick über die im Folgenden behandelten Programmiersprachen geben.

Die Programmiersprache Java wurde ursprünglich von Sun Microsystems entwickelt und 1995 veröffentlicht. Seit der Übernahme des Herstellers, wird die Entwicklung unter der Schirmherrschaft der Oracle Corporation weitergeführt. Produkt dieser Entwicklung ist die sich bis heute entwickelnde Sprachspezifikation *Java Language Specification*, welche von verschiedenen Beteiligten wie IBM, Red Hat oder Hewlett-Packard implementiert wird. Es existiert auch eine offizielle quelloffene Implementation der Java-Spezifikation unter dem Namen OpenJDK.

Die Java Plattform teilt sich grundlegend in zwei Teile: Die Programmiersprache und deren Laufzeitumgebung, die Java Virtual Machine (JVM). Java ist eine klassenbasierte, objekt-orientierte Programmiersprache mit starker statischer Typisierung. Sie unterstützt Multithreading, Objektkapselung und Namensräume. Bei der Erweiterung der Sprache wird darauf geachtet, größtmögliche Kompatibilität mit älteren Versionen zu wahren, was einen relativ konservativen und langsamen Weiterentwicklungsprozess mit sich führt. Diese Stabilität soll Softwareinvestitionen im Geschäftsbereich schützen, da diese bei Java langfristig und stabil wirken.

Java-Code wird auf der JVM ausgeführt und ist somit auf allen Plattformen lauffähig auf die die JVM portiert wurde. Somit kann Java als Crossplatform-Sprache angesehen werden, da mittlerweile adäquate JVM-Implementationen für eine Vielzahl von Plattformen existieren. Java-Sourcecode wird ausgeführt, indem er mit dem Programm „javac“ zu Java-Bytecode kompiliert wird, welcher wiederum von der JVM als Interpreter ausgeführt wird. Der Bytecode ist eine intermediäre Form des Quellcodes, der für die Ausführung durch die JVM optimiert wurde. Die Verwendung der JVM hat mehrere Vorteile: So dient sie als ein sichere Ausführungsumgebung, die zudem das Speichermanagement übernimmt und die Ausführung von Java auf mehreren Plattformen ermöglicht. Des Weiteren kann die JVM Laufzeitinformationen nutzen, um auf deren Basis die

Programmausführung zu optimieren (Just-in-Time-Kompilierung). Dazu werden besonders oft verwendete Methoden direkt in Maschinencode gewandelt anstatt interpretiert werden zu müssen [EF15].

Für die Entwicklung für die Javaplattformen existiert eine Vielzahl an mächtigen und reifen Werkzeugen von integrierten Entwicklungsumgebungen (z.B. IntelliJ IDEA, Eclipse, NetBeans), über Build Tools (z.B. Gradle, Maven, Ant) bis hin zu anderen auf der JVM basierenden Sprachen (z.B. Kotlin, Scala, Groovy). Die große Anzahl an Drittanbieterbibliotheken und Konnektoren ermöglicht die einfache Verwendung von Java in domänenspezifischen Kontexten.

Die zweite in dieser Arbeit thematisierte Sprache ist JavaScript. Sie wurde als Kooperation zwischen Sun Microsystems und Netscape entwickelt und 1995 im Webbrowser Netscape Navigator veröffentlicht. Heute existieren verschiedene Implementationen der Sprache wie z.B. Spidermonkey (Netscape/Mozilla) oder V8 (Google), die auf dem standardisierten Sprachkern ECMAScript basieren. Die aktuelle Version ist ECMAScript 2017 [ECM17]. Ursprünglich wurde JavaScript entwickelt, um HTML auf Webseiten dynamisch gestalten zu können. Es handelt sich um eine dynamische, untypisierte Sprache, in welcher objekt-orientiert und sowohl prozedural als auch funktional programmiert werden kann. Der Code wird von einem Interpreter ausgeführt. Da alle modernen Webbrowser einen JavaScript-Interpreter enthalten, ist JavaScript überall dort lauffähig wo ein Webbrowser vorhanden ist. Mittlerweile wird JavaScript nicht nur für dynamische Webseiten verwendet, sondern auch für clientseitige Webanwendungen, die den Webbrowser als Ausführungsplattform nutzen.

Der Kern von JavaScript definiert dabei nur eine minimale API zur Verarbeitung von Texten, Daten, Arrays und regulären Ausdrücken. Weitere Funktionalitäten wie Dateninput- und Output, Speicherung, Networking und Grafik müssen durch die Host-Umgebung, also im Regelfall dem Webbrowser als API zur Verfügung gestellt werden. Da JavaScript eine performante und allgemeingültige Sprache ist, ist ihr Einsatz jedoch nicht auf Webbrowser beschränkt. Es existieren JavaScript-Interpreter wie Node.js oder Rhino, die JavaScript ohne Bindung an einen Browser ausführen. So kann JavaScript beispielsweise auch für die serverseitige Entwicklung eingesetzt werden [Fla11].

Wie auch für Java existieren für JavaScript eine Vielzahl an Drittanbieterbibliotheken für domänenspezifische Anwendungsfälle und das Schreiben von Testfällen. Zudem existieren IDEs (Webstorm, Eclipse with JSDT, Sublime Text), Paketmanager (npm, Bower, Webpack) und andere Entwicklungswerkzeuge.

Vergleichend können folgende Unterschiede zwischen Java und JavaScript festgehalten werden [EF15]:

- Java ist statisch typisiert, JavaScript dynamisch
- Java basiert auf Klassen, JavaScript auf Prototypen
- Java verfügt über gute Objektkapselung, JavaScript nicht
- Java hat Namensräume, JavaScript nicht
- Java ist Multithreading-fähig, JavaScript nicht

## 3.5. DESMO-J

Als Fallbeispiel für die hier angestrebte Portierung dient das Simulationsframework DESMO-J (Discrete-Event Simulation and MOdelling in Java). Die Software wird seit 1999 vom Arbeitsbereich Modellbildung und Simulation der Universität Hamburg entwickelt und ist unter der Apache 2-Lizenz frei verfügbar. DESMO-J dient zur objektorientierten Entwicklung von Simulationsmodellen mit der Programmiersprache Java und deren Ausführung. Dabei fokussiert sich das Framework auf das Paradigma der diskreten Event-Simulation bei dem alle Änderungen von Systemzuständen zu diskreten Zeitpunkten stattfinden. Es können sowohl Simulationen aus ereignisbasierter als auch aus prozessbasierter Sicht modelliert werden. Als Bibliothekssoftware liefert DESMO-J eine Vielzahl an abstrakten Implementierungen von wichtigen Bestandteilen einer Simulation. Dazu gehören unter anderem Ereignisse und Entitäten, welche durch den Nutzer hinsichtlich der vorliegenden Anforderungen einer bestimmten Simulation erweitert werden. Eine Simulation in DESMO-J liegt dementsprechend als Java-Programm vor. Das Framework übernimmt die Ausführung und generiert anschließend Simulationsberichte in HTML. DESMO-J ist in Java implementiert und enthält nur wenige Abhängigkeiten zu externen Bibliotheken von Drittanbietern. Verwendet werden Funktionen aus der Apache Commons-Bibliothek und Quasar, einer Bibliothek für asynchrone Programmierung. Ansonsten werden für die Erzeugung der Simulationsberichte Grafikfunktionen von Java 3D und JFreechart verwendet. Auch eine optionale Benutzeroberfläche für die Ausführung von Simulationen und deren Auswertung ist vorhanden. Diese verursacht Abhängigkeiten zur UI-Bibliothek Swing, ist jedoch für die grundlegende Ausführung nicht notwendig. Das Framework findet in einer Reihe von Applikationen Anwendung. Dazu gehören unter anderem Modellierungswerkzeuge wie Intellivate IYOPRO oder Borland Together [GJKP13]. Bei einer Anpassung der Java-Version von DESMO-J zwecks Portierung sollte darauf geachtet werden, dass zuvor implementierte Simulationen weiterhin ausführbar bleiben. Eine Abwärtskompabilität des Codes muss also gewährleistet sein. Aus diesem Grund müssen Refactorings im Rahmen der Portierungsmethode mit Bedacht ausgeführt werden. [Uni15]

# 4. Werkzeug-Analyse

In der ersten Phase der Portierungsmethode wird der Bedarf an einer Übersicht möglicher Portierungswerkzeuge erwähnt. Dies bezieht sich auf die Werkzeugunterstützung der im Rahmen dieser Arbeit durchzuführenden Portierung. Laut Methodik sollen möglichst große Teile des zu portierenden Codes automatisch konvertiert werden. Dazu ist ein adäquater Codekonverter nötig, der aus Java-Code JavaScript generiert. In diesem Kapitel wird beschrieben, wie Werkzeuge gefunden und nach welchen Kriterien sie bewertet wurden. Nach einer Vorstellung der ermittelten Werkzeuge wird die Entscheidung für den am besten geeigneten Konverter begründet.

Zur Erstellung einer Werkzeugübersicht muss abgegrenzt werden, welche Arten von Werkzeugen einbezogen werden sollen. Somit soll sichergestellt werden, dass schlussendlich für die Auswahl Werkzeuge betrachtet werden, die die Anforderungen des Anwendungsfalls erfüllen. Für diese Werkzeuganalyse werden Tools einbezogen, die aus Code für die Java-Plattform JavaScript oder eine von JavaScript-Interpretern ausführbare Sprache (z.B. TypeScript) generieren. Der Ausgangscode kann entweder Java-Quellcode oder der daraus kompilierte Java-Bytecode sein. Auch Werkzeuge die dies innerhalb einer Toolchain über eine intermediäre Sprache erreichen werden berücksichtigt. Für die Recherche wurde eine Menge an Schlagwörtern definiert, mit denen nach passenden Werkzeugen gesucht werden kann. Die verwendeten Keywords sind:

Java, JavaScript, Java to/zu JavaScript, Converter, Transpiler, Translator, Translation, Cross compiler, Cross-Compilation, Porting, Transformation

Mithilfe dieser Keywords wurde sowohl nach Werkzeugen in wissenschaftlichen Veröffentlichungen, als auch nach Softwareveröffentlichung im Internet gesucht. Letztere konnten über Suchergebnisse bei Google, Stack Overflow und Github aufgefunden werden.

## 4.1. Bewertungskriterien

Um eine sinnvolle Werkzeugauswahl durchführen zu können müssen klare Kriterien für die Bewertung der einzelnen Werkzeuge festgelegt werden. Durch eine Evaluierung jedes gefundenen Werkzeugs anhand dieser Kriterien kann Vergleichbarkeit geschaffen werden. In diesem Abschnitt sollen die für die im Rahmen dieser Arbeit ausgeführte Werkzeuganalyse ausgewählten Kriterien vorgestellt und erläutert werden. Dabei wird dargestellt inwiefern die einzelnen Kriterien für den Anwendungszweck Relevanz haben. Die Kriterien wurden in Hinblick auf die oben geschilderten Anforderungen an die Werkzeugunterstützung gesammelt sowie aus der Aufgabenstellung übernommen. Die neun gefundenen Bewertungskriterien sind:

1. Funktionsumfang
2. Semantische Korrektheit

3. Verständlichkeit und Wartbarkeit des Generats
4. Grenzen des Konverters
5. Lizenzkosten
6. Langfristiger Support
7. Komplexität der Konfiguration
8. Integration in Entwicklungsumgebung und –Prozess
9. Erweiterbarkeit (für Traceability und zukünftige Sprachkonzepte)

Die ersten vier Kriterien beziehen sich hauptsächlich auf den konkreten Übersetzungsvorgang von Java zu JavaScript. Kriterium 1 betrifft den Funktionsumfang des Codekonverters. Der Funktionsumfang definiert die Menge an Sprachkonzepten der Ausgangssprache Java, die der Konverter in JavaScript übersetzen kann. Idealerweise würde ein Codekonverter alle grammatikalischen Konzepte einer Ausgangssprache verarbeiten und in gleichartige Konstrukte der Zielsprache umwandeln. In der Realität erreichen verschiedene Codekonverter jedoch verschiedene Grade der Sprachabdeckung. Des Weiteren muss die vom Konverter unterstützte Version der Ausgangs- und Zielsprache betrachtet werden. Bei nicht aktiv entwickelten Werkzeugen mangelt es vielfach an Unterstützung für in neueren Versionen der Sprachen hinzugekommene Sprachelemente. Es ist jedoch zu evaluieren, ob syntaktische Neuerungen tatsächliche Spracherweiterungen oder lediglich für den Programmierer gedachte Vereinfachungen sind. Letztere sind bei der Bewertung des Funktionsumfanges irrelevant, da sie für die Portierung in ihre semantisch gleiche, ursprüngliche und somit oftmals unterstützte Form überführt werden können. Der Übergang zwischen Spracherweiterungen und Syntaxalternativen ist jedoch oft fließend. Konkret muss geprüft werden, welche Sprachversion durch den Konverter maximal abgedeckt wird. Auf Java-Seite betrifft dies primär die vollständige, teilweise oder nicht vorhandene Unterstützung von Java 8. Zusätzlich zu syntaktischen Sprachelementen soll dabei auch die Fähigkeit zur Auflösung von Abhängigkeiten einbezogen werden. Wie bereits in Abschnitt 3.2 erwähnt können verschiedene Arten von Abhängigkeiten in Softwareprojekten auftreten, die unter Umständen vom Portierungswerkzeug mit übersetzt werden können. Denkbar sind hier unter anderem automatische Mappings von der Java-API zur JavaScript-API oder die Einbindung semantisch übereinstimmender Drittanbieterbibliotheken. Weitere mögliche Aspekte des Funktionsumfangs wären Möglichkeiten zur Konfiguration eigener Mappings, die Unterstützung weiterer Sprachen (z.B. JavaScript-kompatible Zielsprachen) oder beim Portierungswerkzeug mitgelieferte transformationsvorbereitende Refactorings. Dementsprechend muss geprüft werden, welche Möglichkeiten zur Portierung von Abhängigkeiten das Werkzeug bietet.

Die in Kriterium 2 angesprochene semantische Korrektheit, bezieht sich auf die Gleichheit des Verhaltens der Software in der übersetzten Form. Die Qualität einer Portierung hängt primär von der Vergleichbarkeit des Verhaltens beider Versionen ab. Sowohl die Ausgangsversion als auch die Zielversion sollen auf der Basis gleicher Daten die gleichen Ergebnisse liefern. Ein Codekonverter sollte daher die Geschäftslogik einer Software ohne Abänderungen in der Zielsprache umsetzen können. Ist dies nicht gewährleistet gehen Vorteile gegenüber einer Neuentwicklung, wie zum Beispiel die Vermeidung von Bugs in der Geschäftslogik, verloren. Je höher also die semantische

Korrektheit der Zielversion die einem Portierungswerkzeug nachgewiesen werden kann, desto geringer ist der an die Portierung anschließende Test- und Korrekturaufwand. Eine Überprüfung der semantischen Übereinstimmung von Input und Output kann durch Prüfungen mit semantisch übereinstimmenden Unit-Tests durchgeführt werden. Des Weiteren kann Testcode verwendet werden, welcher alle für den Kontrollfluss der Software notwendigen Sprachelemente beinhaltet.

Das dritte Kriterium betrifft die Verständlichkeit und Wartbarkeit des Generats, also des portierten Codes in der Zielsprache. Wie bereits erläutert wird nach der Portierung eine parallele Weiterentwicklung der Software auf beiden Plattformen angestrebt. Somit ist es notwendig, dass der Output nicht nur für den JavaScript-Interpreter verständlich ist, sondern auch für die Entwickler die für die Entwicklung der JavaScript-Version verantwortlich sind. Daher werden an die Präsentation des Output-Codes Anforderungen wie Lesbarkeit und Verständlichkeit gestellt. Der in Java umgesetzte Kontrollfluss muss für den Entwickler auch in JavaScript erkennbar sein. Besonders bei Portierungswerkzeugen, die auf der Basis von Java-Bytecode arbeiten geht der klar strukturierte Kontrollfluss oft verloren und muss für lesbares JavaScript erst wieder nachträglich rekonstruiert werden (siehe z.B. [LSWM15]). Code-Konverter, die keinen von Menschen wartbaren JavaScript-Code erzeugen sind für den hier definierten Anwendungsfall einer synchronisierten Weiterentwicklung nicht geeignet. Ein weiterer Aspekt für die Verständlichkeit des Codes ist die Strukturgleichheit zwischen Ausgangs- und Zielversion. Zu einer hohen Strukturgleichheit tragen plattformübergreifend gleich gewählte Bezeichner, Algorithmen und Projektstrukturen bei. Grundsätzlich soll die Kenntnis der Originalversion das einfache Verstehen der portierten Version unterstützen, da eine strukturgleiche Portierung einen gleichförmigen Aufbau hat. Wartungsarbeiten können so vereinfacht werden, da der Aufwand für das Codeverständnis sinkt bzw. entfällt. Daher muss geprüft werden, mit wie viel Aufwand Codestrukturen und Aufrufe der Java-Version in JavaScript lokalisiert werden können. Auch die oben angesprochene, mögliche Unterstützung von Sprachsyntax, die primär den Programmierer unterstützt, kann zum besseren Verständnis des Generats beitragen. Syntaxkonstrukte wie möglicherweise Kurzschreibweisen können helfen die Komplexität des Codes zu reduzieren, vorausgesetzt das Konvertierungswerkzeug unterstützt deren Generierung in der Zielsprache. Ein Beispiel für solche Konstrukte wäre in JavaScript das Ausdrücken einer „If..Else..“-Bedingung durch den ternären Operator „?“. Allerdings hängt die Wirksamkeit solcher Sprachkonstrukte für das Verständnis subjektiv vom Programmierstil des Entwicklers ab.

Das vierte, die Übersetzung betreffende Kriterium sind die Grenzen bzw. Limitationen des Code-Konverters. Darunter fallen fehlerhafte Übersetzungen der durchaus im Funktionsumfang enthaltenen Sprachkonzepte. Diese können in Code mit zu hoher Komplexität auftreten oder sich aus einer Kombination von unterstützten und nicht unterstützten Sprachkonzepten ergeben. Eine weitere Fehlerquelle sind Fehler bzw. Bugs im Konverter, die für eine inkorrekte Generierung des JavaScript-Codes sorgen. Solche Limitationen sind jedoch vor der Verwendung in einem konkreten Anwendungsfall oft schwer zu ermitteln, da ein hoher zusätzlicher Testaufwand mit einer Vielzahl an Spezialfällen anfallen würde. Allerdings sollte geprüft werden, ob solche Fehlerquellen bereits in der Community bekannt sind und veröffentlich wurden. Beispielsweise gibt die Betrachtung eines Issue Trackers der zu verwendenden Werkzeugversion darüber Aufschluss. Interessant ist auch zu ermitteln, wie das Werkzeug mit nicht konvertierbaren Codeelementen umgeht. Grundlegend wäre es denkbar, dass die Stellen automatisch auskommentiert oder ganz entfernt werden. In diesem Fall würde eine Konvertierung auch bei nicht adäquat präpariertem Code durchlaufen, wobei evtl.

anschließende Konsequenzen des fehlenden Codes zu tragen sind. Andernfalls könnte das Werkzeug eine Konvertierung bei nicht übersetzbaren Elementen auch abbrechen, was wiederum manuelle Anpassungen im Vorhinein erfordert. Natürlich sollte bei der Auswahl eines Konvertierungstools für ein bestimmtes Projekt geprüft werden, ob diese Limitationen kritische Auswirkungen auf das Projekt haben oder eventuell nicht relevant oder mit wenig Mehraufwand vermeidbar sind. Im Falle dieser Arbeit müssen also die Auswirkungen von möglichen Limitationen auf die Portierung der Codestruktures in DESMO-J betrachtet werden.

Kriterium 5 umfasst das Lizenzmodell und damit die Lizenzkosten für das Werkzeug und dessen Nutzung. Die Nutzungskosten für die verwendeten Werkzeuge müssen in der gesamten Kostenbetrachtung des Portierungsprojekts beachtet werden. Primär muss entschieden werden, ob sich eine Portierung gegenüber einer Neuimplementierung für die Zielplattform lohnt. Die Werkzeugkosten sind ein wichtiger Aspekt dieser Rechnung. Oftmals sind Portierungswerkzeuge jedoch eher im Open Source-Bereich angesiedelt, wodurch die Nutzung tendenziell kostenlos ist. Zudem wird die Anpassung des Tools für den eigenen Anwendungsfall durch freien Codezugang ermöglicht. In der Regel liegt das Risiko des Open Source-Werkzeugeinsatzes jedoch beim Nutzer und nicht beim Hersteller. Im Rahmen dieser akademischen Arbeit werden aufgrund der Anforderungen kostenlose Open Source-Werkzeuge bevorzugt. Je nach Projektkontext kann diese Einschätzung jedoch auch anders ausfallen. Für eine vollständige Kosten- bzw. Aufwandsbetrachtung müssen vor allem zusätzlich die Aspekte der Kriterien 6 bis 9 mit einbezogen werden.

Ein wichtiges Kriterium für die Benutzbarkeit eines Werkzeugs ist dessen Support durch den Hersteller oder eine dazugehörige Community. Zum Support gehören unter anderem Anleitungen zum Einsatz des Werkzeugs in Form von Tutorials oder einer Dokumentation. Letztere kann je nach Ausführlichkeit auch Aufschlüsse über den Funktionsumfang des Werkzeugs liefern. Kriterium 6 beinhaltet jedoch nicht nur die Werkzeugdokumentation sondern auch deren Aktualität und Anwendbarkeit auf den aktuellen Stand des Tools. Aktiv entwickelte Werkzeuge haben oftmals eine höhere Wahrscheinlichkeit mit aktuellen und umfassenden Informationen ausgestattet zu sein. Die Menge an verfügbarem Support hängt auch davon ab, ob der Hersteller oder eine Nutzercommunity kontaktierbar sind. Aktive Communities tragen vielfach zu einer besseren Dokumentation von Tools bei und sorgen zudem für eine hohe Anzahl an potentiellen Ansprechpartnern bei Problemen während der Nutzung. Eine weitere Möglichkeit des Supports wäre das Vorhandensein von bezahlter Beratung oder Projektunterstützung, was jedoch außerhalb von kommerziellen Werkzeugen selten vorkommt. Im Kontext dieser Arbeit muss geklärt werden, ob die Qualität der Dokumentation für die sinnvolle Nutzung des Werkzeugs und eventuelle Problemlösungen hinreichend ist.

Die Kriterien 7 und 8 beziehen sich auf die Einbindung des Werkzeugs in den Portierungsprozess. Zum einen muss die Komplexität der Konfiguration evaluiert werden. Darunter ist der Aufwand zu verstehen, die vom Tool angebotenen Einstellungsmöglichkeiten hinsichtlich des durchzuführenden Projektes korrekt zu konfigurieren. Grundlegend bietet eine Vielzahl an Konfigurationsmöglichkeiten in der Regel eine bessere Anpassung an die Anforderungen der Portierung und projektspezifische Umstände. So könnten beispielsweise vom Nutzer definierte Mappings auf selbst entwickelte Codebibliotheken eingebunden werden. Neben den optionalen Konfigurationsmöglichkeiten benötigen Tools jedoch auch notwendige Einstellungen, ohne deren Anpassung die Werkzeugnutzung nicht möglich ist. Gerade bei diesen Einstellungen kann je nach Komplexität ein großer Aufwand

auftreten. Zu bevorzugen sind daher Werkzeuge, deren notwendige Konfiguration simpel und weiterreichende Anpassungsmöglichkeiten umfangreich sind.

Zum anderen ist die Integration in den Entwicklungsprozess und die -Umgebung zu betrachten. Um den Portierungsprozess zu unterstützen sollte eine möglichst effiziente Verzahnung mit dem etablierten Entwicklungsprozess ermöglicht werden. Der portierende Entwickler sollte für die Verwendung des Tools möglichst wenige, nicht aufwendige Schritte ausführen müssen. Dies kann beispielsweise durch eine Integration des Code-Konverters in die Entwicklungsumgebung erreicht werden. Werkzeuge, die funktionsfähige Plugins für IDEs wie IntelliJ oder Eclipse mitbringen, können somit gut in den Prozess eingebunden werden. Eine weitere Möglichkeit wäre die Integration von Werkzeugen in Build-Systeme. Die Bewertung dieses Kriterium hängt besonders stark von den vorliegenden Umständen ab, da die Integrierbarkeit des Tools stark von der tatsächlich verwendeten Toolumgebung abhängt. Im Falle des hier beschriebenen Portierungsprojektes kann die Toolumgebung jedoch flexibel gewählt werden, da keine Rücksicht auf bereits etablierte Entwicklungsprozesse genommen werden muss.

Als letztes Kriterium ist die Erweiterbarkeit des Werkzeugs zu prüfen. Im konkreten Anwendungsfall dieser Arbeit soll Traceability zwischen der Ausgangs- und der portierten Version der Software erzeugt werden. Dazu müssen für übereinstimmende Codeelemente Tracelinks generiert und abgespeichert werden. Eine Möglichkeit dies zu erreichen, wäre die Integration der Tracelink-Erzeugung in den Code-Konverter. Während der Übersetzung kann das aktuell betrachtete Codeelement direkt mit der daraus erzeugten Übersetzung verknüpft werden. Neben dem Output-Code würde der Konverter in diesem Fall zusätzlich eine Repräsentation der erzeugten Tracelinks ausgeben. Um dies umzusetzen, ist jedoch die Erweiterbarkeit des Übersetzungswerkzeugs nötig. Bei kommerziellen Werkzeugen ist dies unter Umständen über einen Pluginmechanismus möglich, oftmals gestalten sich individuelle Erweiterungen jedoch schwierig. Einfacher ist die Erweiterung bei Open Source-Werkzeugen. Hier können gewünschte Funktionen durch den Nutzer implementiert werden. Allerdings ist in diesem Fall zu prüfen inwiefern sich das Werkzeug im Originalzustand für die Weiterentwicklung eignet. Ausschlaggebende Aspekte sind dabei der Umfang des Quellcodes, dessen Komplexität und das Vorhandensein von Entwurfsmustern, die Erweiterungen unterstützen. Besonders geeignet sind Werkzeuge, die über eine eigene Plugin-API verfügen, welche für die Entwicklung von Nutzererweiterungen verwendet werden kann.

## 4.2. Bewertung der vorhandenen Werkzeuge

Die oben beschriebene Toolrecherche erzielt eine Vielzahl von Treffern, wobei nach kurzer Überprüfung acht tatsächliche Werkzeuge identifiziert werden konnten. Die gefundenen Ergebnisse sollen in diesem Abschnitt genauer betrachtet und anhand der zuvor dargelegten Kriterien geprüft werden. Grundlegend sind die Portierungswerkzeuge in zwei Kategorien einzuteilen: Cross-Compiler (Java Bytecode-to-JavaScript JBC2JS) und Transpiler (Java-to-JavaScript J2JS). Bei ersteren handelt es sich um Werkzeuge, die Java-Bytecode als Input verwenden, wohingegen letztere den Java-Source Code verwenden. Wie an den Rechercheergebnissen zu sehen, setzt die Mehrzahl an Portierungswerkzeugen auf den Cross-Compiler-Ansatz. Nur drei Werkzeuge arbeiten als Transpiler, wovon eines eine Kombination aus mehreren Konvertierungswerkzeugen ist. Tabelle 4.1 bietet eine Übersicht über die gefundenen Werkzeuge.

| Werkzeugname | Hersteller | Werkzeugtyp |
|---|---|---|
| AOT JS compiler using Graal | Leopoldseder et al. | JBC2JS Cross-Compiler |
| Bck2Browser | Jaroslav Tulach | JBC2JS Cross-Compiler |
| Dragome | Dragome | JBC2JS Cross-Compiler |
| GWT Transpiler | Google Inc. | J2JS Transpiler |
| JSweet | JSweet | J2JS Transpiler |
| Kotlin-to-JS compiler | JetBrains | Kombination |
| TeaVM | TeaVM | JBC2JS Cross-Compiler |
| XMLVM | Puder et al. | JBC2JS Cross-Compiler |

Tabelle 4.1.: Verhandene Werkzeuge zur Übersetzung von Java zu JavaScript

### 4.2.1. Transpiler (J2JS)

Im Bereich der Java zu JavaScript-Transpiler lassen sich weniger Werkzeuge finden, allerdings umfasst diese Kategorie die wohl bekannteste Technologie, das **Google Web Toolkit (GWT)**. GWT wird seit 2006 von Google als Open Source-Software entwickelt. Dabei ist das Ziel die Entwicklung von Web-Applikationen ohne umfangreiche Kenntnis von JavaScript, HTML und browserspezifischen Eigenschaften zu ermöglichen. Stattdessen werden AJAX-Anwendungen in Java implementiert, wobei das GWT SDK verschiedene APIs bietet, um JavaScript-Sprachkonzepte in abstrahierter Form in Java umsetzen zu können. Dazu gehören zum Beispiel Aspekte wie die Manipulation des Document Object Model (DOM) eines HTML-Dokuments. Zudem enthält das SDK UI-Widgets, die ähnlich wie Swing-Widgets verwendet werden können, um die Benutzeroberfläche von Web-Anwendungen zu gestalten. Der so geschriebene Java-Code wird zur Ausführung im Browser mithilfe des in GWT integrierten Transpilers zu optimiertem JavaScript-Code übersetzt. Diese Funktionalität kann nicht nur für auf Basis von GWT geschriebene Webanwendungen, sondern auch zur Portierung von Java-Software zu JavaScript verwendet werden. Laut Google erzeugt GWT JavaScript-Code, welcher sowohl für moderne Desktop-Browser, als auch für Browser von Mobilgeräten optimiert ist.

Die von GWT unterstützten, übersetzbaren Elemente des JRE sind in der Dokumentation transparent dargelegt [Goo17c]. Reflection wird bis auf das Abfragen des Namens von Klassen nicht unterstützt. Wie alle anderen betrachteten Konverter unterstützt auch GWT kein Multithreading. Einige Aspekte der Java-API wie reguläre Ausdrücke und Serialisierung werden nicht unterstützt. Stattdessen liefert GWT eigene Implementationen dieser Funktionen, weshalb solche Aufrufe vor der Konvertierung manuell ersetzt werden müssen. Solche Aspekte mit einbezogen besitzt GWT im Werkzeugvergleich einen großen Funktionsumfang, wobei Java 8 nur teilweise unterstützt wird. Insgesamt entfernt GWT Code-Elemente, die nicht vom Transpiler unterstützt werden, was zu nicht funktionalem Code mit wenig Hinweisen zur Fehlerursache führen kann. Die Lesbarkeit des erzeugten JavaScript-Codes kann in der Konfiguration des Transpilers beeinflusst werden. Für maximale Wartbarkeit des Codes sollten Code-Optimierungen, Verschleierung von Bezeichnern und Verkürzungen deaktiviert werden. In diesem Fall wird eine gute Lesbarkeit des Generats erreicht.

GWT wird weiterhin aktiv entwickelt und in einer Vielzahl von Softwareprojekten verwendet. Google selbst realisiert mehrere der eigenen Web-Anwendungen wie Inbox, AdWords und Groups mit GWT und auch darüber hinaus findet das Framework produktiven Einsatz [Goo17a]. Dementsprechend steht Nutzern eine Vielzahl an aktuellem Informationsmaterial zur Verfügung. Dieses

reicht von Installations- und Nutzungsleitfäden, über Codebeispiele, Entwicklerdokumentationen und FAQs bis hin zu einer Bibliothek an Sekundärliteratur zum Beitrag und der Nutzung von GWT (z.b [THET13], [GU10]). Aktiver Projektsupport durch Google wird nicht geleistet, ist aber kostenpflichtig von anderen Anbietern erhältlich. Die Nutzercommunity umfasst laut Google zehntausende von Projektgruppen, was sich auch in der hohen Aktivität für das Schlüsselwort „gwt“ auf der Supportseite Stack Overflow widerspiegelt. Ein langfristiger Support von GWT seitens Google ist aufgrund der bisherigen Investition in das Projekt wahrscheinlich. Der Support und somit die Nutzbarkeit des Werkzeuges sind sehr gut.

Zur Verwendung von GWT und des darin enthaltenen Transpilers wird das GWT SDK sowie das Build-Tool Ant benötigt. GWT kann über ein von Google entwickeltes Plugin auch in die IDE Eclipse integriert werden. In IntelliJ IDEA ist der Support für GWT bereits als vorinstalliertes Plugin vorhanden. Die Plugins integrieren die API des GWT SDK in Codevorschläge und -Vervollständigung. Zudem liefern sie GWT-spezifische Refactorings, Code-Templates und die Integration des Transpilers in die IDE [Jet16a]. Die Funktionalitität des Transpilers ist aufgrund des offenen Quellcodes programmatisch erweiterbar, allerdings bietet der Transpiler keine spezifische API für Erweiterungen. Dadurch kann je nach Art der Anpassung, auch aufgrund des Umfangs der Codebasis, ein großer Aufwand bei der Erweiterung entstehen. [Goo17b].

Das zweite, auf Basis eines Transpilers basierende Werkzeug ist **JSweet**. JSweet übersetzt Java-Source Code in JavaScript, wobei TypeScript als intermediäre Sprache zum Einsatz kommt. Somit vermag es JSweet alternativ auch TypeScript als Output zu generieren. TypeScript ist eine von Microsoft entwickelte Sprache, die JavaScript um Klassen, Module und optionale Typen erweitert und zur Ausführung zu JavaScript kompiliert wird [Mic17]. Wie auch GWT ist JSweet als Open Source-Software verfügbar. Dementsprechend fallen für die Nutzung von JSweet keine Lizenzgebühren an. Das Werkzeug wurde 2015 erstmals veröffentlicht und wird bis heute aktiv entwickelt. Der Hersteller gibt für das Werkzeug zwei primäre Anwendungsfälle an: Zum einen soll es ermöglichen, Legacy-Anwendungen in Java zu JavaScript zu portieren und sie dadurch für die Web-Plattform aufzubereiten. Dies ist laut Hersteller jedoch nur mit Einschränkungen möglich, da eine Portierung in jeden Fall manuelle Anpassungen erfordert. Zum anderen soll JSweet die Interoperabilität zwischen Java- und JavaScript-Code verbessern. So können Server-Software in Java und Client-Software in JavaScript besser zusammenarbeiten, indem sie über JSweet auf gleichartige Transferobjekte und typisierte APIs zugreifen können. Um dies zu ermöglichen bietet JSweet eine Core-Bibliothek, die JavaScript-spezifische Konzepte wie globale Variablen oder Arrays in Java abbilden. So können Entwickler bereits in Java JavaScript-spezifische Konzepte verwenden. Bei einer Portierung kann das Umsetzen von Java-Logik durch Verwendung der JSweet Core-Bibliothek als transformationsvorbereitendes Refactoring gelten, welches die automatische Übersetzung durch erhöhte Kompatibilität erleichtert. Zudem ermöglichen sogenannte JSweet „Candies“, Adapter für unterstützte JavaScript-Bibliotheken, das Aufrufen derselben aus Java-Quellcode. So können Abhängigkeiten zu Drittanbieterbibliotheken in Java unter Umständen bereits vor der Übersetzung aufgelöst werden, indem sie durch JavaScript-Äquivalente ersetzt werden.

Im Vergleich mit den anderen in diesem Vergleich betrachteten Konvertern kann der Transpiler von JSweet mit der größten Anzahl an Sprachkonzepten von Java umgehen. Ein besonderer Vorteil für Nutzer ist die deutlich in der Sprachspezifikation von JSweet beschriebene Menge an Java-APIs,

die ohne Anpassungen übersetzt werden kann [Paw16b]. Der Funktionsumfang wird darüber hinaus jedoch mithilfe von JSweet-APIs erweitert, die die Verwendung von JavaScript-Typen in Java ermöglichen. Auch kann durch die Verwendung bestimmter Annotationen das Verhalten des Transpilers bei der Übersetzung der annotierten Code-Elemente bestimmt werden. Die Qualität der Übersetzung kann durch Einarbeitung der JSweet-API deutlich verbessert werden, was jedoch bei bestehendem Code Refactoring-Aufwand mit sich bringt. Der Transpiler erzeugt JavaScript, welches strukturell sehr nah am Java-Original liegt und somit gut lesbar und wartbar ist. Alternativ kann die Wartung auch am generierten TypeScript durchgeführt werden, was dem Java-Original strukturell noch mehr ähnelnd. Funktionen zur Minimierung des Generats können jedoch optional konfiguriert werden. JSweet verfügt im Vergleich über den größten Funktionsumfang, wobei durch die Ersetzung von Abhängigkeiten durch JavaScript-Äquivalente viele Portierungsprobleme umgangen werden können. Bei der semantischen Korrektheit konnten bis auf Unterschiede von möglichen Zahlenräumen keine Einschränkungen festgestellt werden. Dieses Problem tritt bei allen Konvertern auf, die die verschiedenen primitiven Zahlentypen unabhängig von ihrem Zahlenraum durch *var* auf JavaScript-Seite ersetzen. JSweet bietet allerdings die Möglichkeit dieses Problem durch eigens definierte Typen zu umgehen, welche vor der Übersetzung eingepflegt werden müssen.

JSweet verfügt über eine umfangreiche, online verfügbare Dokumentation. Diese enthält neben einer Installationsanleitung und FAQ auch Codebeispiele und die JavaDocs für das gesamte Projekt. Die Internetseite bietet mit einer Live-Sandbox die Möglichkeit die Übersetzung zu testen ohne JSweet installieren und konfigurieren zu müssen. Als weitere Unterstützung des Nutzers steht eine umfangreiche Sprachspezifikation von JSweet und wissenschaftliche Veröffentlichungen über den Transpiler ([Paw15a], [Paw15b]) zur Verfügung. Neben umfangreichem Informationsmaterial kann der Nutzer auch auf aktive Supportmöglichkeiten zurückgreifen. So ist der Hersteller über offizielle Kommunikationswege zu erreichen und bietet als einziger Anbieter kostenpflichtigen Support. Auch die Nutzercommunity von JSweet ist groß und aktiv, was sowohl an der Aktivität in den offiziellen JSweet-Foren als auch auf Support-Plattformen wie Stack Overflow zu erkennen ist. Dabei werden nicht nur viele Fragen gestellt, sondern auch zeitnah von anderen Mitgliedern der Community oder Entwicklern des Werkzeugs beantwortet.

Das Werkzeug setzt zur Verwendung das Java JDK 8 und Maven voraus. Um den übersetzten JavaScript-Code auszuführen, wird Node.js als JavaScript-Interpreter benötigt. Um die Einbindung in den Entwicklungsprozess zu vereinfachen, bietet JSweet ein Plugin für die Java-IDE Eclipse an. Plugins für andere IDEs sind nicht verfügbar. Das Plugin erleichtert die Konfiguration des Transpilers durch eine grafische Oberfläche. Das Verhalten des Transpilers lässt sich über eine Vielzahl von gut dokumentierten Konfigurationsoptionen anpassen. So kann zum Beispiel festgelegt werden, welche Version des ECMAScripts (ES3, ES5, ES6) durch den generierten Code angesprochen werden soll. Besonders über das Plugin ist die Komplexität der Konfiguration sehr gering. Einige der Konfigurationsmöglichkeiten lassen sich jedoch nur über Maven einstellen. Neben mitgelieferten Anpassungsmöglichkeiten ist auch die programmatische Erweiterung des Transpilers möglich. Für diesen Zweck steht ein Erweiterungsmechanismus mit Plugin-API zur Verfügung. Über diese API bekommen Entwickler Zugriff auf den abstrakten Syntaxbaum des zu übersetzenden Codes, in welchem eigene Anpassungen vorgenommen werden können. Eine weitere Art der Erweiterung durch den Nutzer ist das Definieren von Java-Annotationen, welche an markierten Stellen im Java-Code bestimmte Übersetzungsmuster auslösen können. Solche Annotationen ermöglichen

die automatische Durchführung von zuvor definierten transformationsvorbereitenden Refactorings [JSw17]. Im Vergleich verfügt JSweet über die weitreichendsten Möglichkeiten der Erweiterung.

Eine weitere Möglichkeit Java zu JavaScript zu übersetzen ist die Verkettung von Konvertierungswerkzeugen unter Verwendung von intermediären Sprachen. Die Entwicklungsumgebung IntelliJ IDEA von der Firma Jetbrains bietet die Möglichkeit Java zuerst in die von Jetbrains entwickelte Sprache Kotlin zu konvertieren und das entstandene **Kotlin zu JavaScript** zu kompilieren. Ursprünglich wurde Kotlin als Sprache für die Java Virtual Machine entworfen, allerdings kann Kotlin seit Version 1.1 neben JVM-Bytecode auch zu JavaScript kompiliert werden [Jet17]. Um diese Art der Konvertierung zu verwenden, muss das von Jetbrains frei erhältliche Kotlin-Plugin in IntelliJ IDEA eingefügt werden, wobei es bei der Einrichtung von Projekten mit Kotlin auch automatisch hinzugefügt wird. Das Plugin liefert IDE-Support für die Kotlin-Programmierung, den Java zu Kotlin-Konverter J2K und den JavaScript-Compiler. Aufgrund der Einbindung in die Entwicklungsumgebung ist die Inbetriebnahme des Werkzeugs sehr einfach. Eine weitere Konfiguration ermöglicht das Festlegen von Outputverzeichnissen und die Auswahl, ob Source Maps generiert werden sollen. Eine Einbindung in andere Entwicklungsumgebungen ist nicht vorgesehen. Eine Erweiterung des Werkzeugs ist aufgrund des offenen Quellcodes grundsätzlich möglich, wird jedoch nicht durch eine spezifische API unterstützt.

Der kritische Punkt bei der Nutzung der hier besprochenen Toolchain ist der Funktionsumfang des Java zu Kotlin-Konverters (J2K). Dieser deckt die meisten Sprachkonstrukte von Java ab, unterstützt aber im jetzigen Stand nicht alle Features von Java 8 (siehe [Uda16]). Abhängigkeiten zu genutzten Bibliotheken müssen an dieser Stelle nicht ersetzt werden, da Kotlin mit bestehenden Java-Bibliotheken interoperabel ist. Bei der Kompilierung zu JavaScript vermag es die Werkzeugunterstützung nicht, Abhängigkeiten für die neue Umgebung aufzulösen. Aufgrund der aktiven Entwicklung von Kotlin und der dazugehörigen Toolunterstützung ist eine Verbesserung der Übersetzung in Zukunft wahrscheinlich. Insgesamt ist der Funktionsumfang jedoch eher gering, da das Tool nicht alle geläufigen Sprachkonzepte in Java abdeckt und keinerlei Möglichkeiten für Mappings bietet.

Der Support für die in diesem Ansatz verwendeten Funktionen von IntelliJ IDEA beschränkt sich auf Tutorialbeiträge auf der Kotlin-Projektseite [Jet16b]. Allerdings weist die Entwicklungsumgebung eine umfangreiche Nutzerbasis auf, durch deren Hilfe auf Supportseiten Probleme gelöst werden könnten.

### 4.2.2. Cross-Compiler (JBC2JS)

Dieser Abschnitt soll einen Überblick über die bei der Werkzeugrecherche gefundenen Tools auf Basis eines Cross-Compilers geben. Als Input dient bei dieser Art von Portierungswerkzeugen JVM-Bytecode, welcher unter Umständen unter Zuhilfenahme einer intermediären Sprache zu JavaScript-Code konvertiert wird.

Zwei der gefundenen Portierungstools sind aus akademischer Forschung zu Quellcodeübersetzung entstanden. Eines davon ist der Ahead-Of-Time JavaScript-Compiler auf der Basis von Graal (**AOT JS**), welcher auf der 2015 erschienenen Forschungsarbeit von Leopoldseder et al. basiert und bereits in Kapitel 2 angesprochen wurde. Die im Paper beschriebene Werkzeugunterstützung in Form einer Erweiterung des Java-Compilers Graal ist jedoch nicht aufzufinden und kann somit nicht

geprüft oder angewendet werden. Aus dem Text geht hervor, dass das Werkzeug vielfach versucht Java-typische Sprachkonstrukte in JavaScript zu rekonstruieren, womit angestrebt wird den vollen Umfang an Sprachkonzepten in Java zu unterstützen anstatt nicht direkt kompatible Konzepte auf Java-Seite zu verbieten. Dazu führt das Werkzeug interne Refactorings und Optimierungen des Bytecode-Inputs durch. Des Weiteren enthält der JavaScript-Generator des Tools eine Kontrollfluss-Rekonstruktion, mit welcher die Lesbarkeit und damit auch Wartbarkeit des generierten JavaScripts verbessert werden soll. Dabei wird Sprachsyntax in Java, die durch Optimierungen während der Umwandlung zu Bytecode für Menschen schwer verständlich wird, für den JavaScript-Code wieder rekonstruiert. Da der AOT JS-Compiler offenbar nicht frei verfügbar und nutzbar ist, kann er bei der Werkzeugauswahl nicht weiter betrachtet werden.

**Bck2Brwsr** (Back to Browser) ist ein weiteres Werkzeug, welches JavaScript aus Java-Bytecode erzeugt. Der Konverter wurde 2012 erstmalig als Open Source-Software veröffentlicht und wird bis heute aktiv entwickelt. Die Motivation hinter dem Projekt ist es, die Entwicklung von HTML5-Anwendungen in Java zu ermöglichen. Allerdings sei das Ziel des Projekts laut Entwickler nicht die Portierung von existierenden Codebibliotheken zu JavaScript. Für die gegenüber Java eingeschränkte Umgebung von JavaScript müssten eigene Bibliotheken entwickelt werden. Dementsprechend ist das Werkzeug darauf ausgelegt, Java-Konzepte die in JavaScript nicht verfügbar sind, im Java-Code nicht zu verwenden, anstatt Lösungen zu finden diese zu übersetzen. Daher eignet sich Bck2Brwsr hauptsächlich für Neuentwicklungen auf Basis des Tools und weniger für die Anwendung auf bestehende Java-Software. Letzteres wäre nur mit weitreichenden Anpassungen möglich, die inkompatible Code-Elemente aus der bestehenden Codebasis entfernen. Bck2Brwsr unterstützt keinerlei Möglichkeiten, Mappings für API-Aufrufe anzulegen oder durchzuführen. Über eine von Bck2Brwsr zur Verfügung gestellte Bibliothek kann aus Java-Code auf HTML-Seiten zugegriffen werden. UI-Widgets von Swing werden nicht unterstützt, stattdessen soll die HTML4Java-API von Netbeans für Benutzeroberflächen genutzt werden. Java 8 wird bis auf Lambda-Ausdrücke nicht unterstützt. Insgesamt ist der Funktionsumfang nicht ausreichend um die Portierung einer bestehenden Codebasis durchzuführen.

Die Nutzung von Bck2Brwsr erfordert entweder das Build-System Maven oder eine programmatische Integration der angebotenen API. Durch die Verwendung von Maven kann das Werkzeug in Entwicklungsprozesse mit IDEs eingebunden werden. Eine benutzerdefinierte Konfiguration des Konverters ist ohne Bearbeitung des Quellcodes nicht möglich. Dadurch ist die Komplexität der Konfiguration sehr hoch. Der offene Quellcode lässt solche Anpassungen jedoch grundsätzlich zu. Eine spezifische API für Erweiterungen des Konverters ist nicht verfügbar. Neben den JavaDocs des Projektes besteht die Dokumentation aus einigen Anwendungsbeispielen, die jedoch die Benutzung des Werkzeugs nur unzureichend erklären. Darüber hinaus ist keine Hilfestellung verfügbar, was die Benutzung deutlich erschwert. Zudem beziehen sich die vorhandenen Beispiele nicht auf die aktuelle Version des Konverters [Tul17]. Der vorhandene Support ermöglicht kaum die sinnvolle Nutzung des Tools.

Ein weiteres Werkzeug auf Cross-Compiler-Basis ist **Dragome**. Laut Hersteller eignet es sich für die Entwicklung von Anwendungen auf Client-Seite in Java, welche zu JavaScript konvertiert werden. Das Projekt befindet sich seit 2014 in Entwicklung und ist auch weiterhin aktiv, wobei der

letzte Release im März 2017 war. Der Hersteller grenzt Dragome im Gegensatz zum Transpiler-basierten GWT damit ab, dass Dragome transformationsvorbereitende Refactoring im bereits optimierten Java-Bytecode ausführen kann. Diese Refactorings können in Form von Plugins individuell eingefügt und auch selber entwickelt werden. Mitgeliefert wird eine Funktionalität, die Callbacks bei asynchronen Aufrufen durch dynamische Proxies ersetzt. Der Konverter unterstützt alle aktuellen Sprachkonzepte von Java, einschließlich Java 8. Zudem unterstützt Dragome den Zugriff auf im Java-Projekt hinterlegte HTML-Templates durch eine API für JavaScript-Konstrukte. Auch bietet Dragome eine Reihe an UI-Widgets die in Java verwendet werden können um HTML-Seiten zu konstruieren. Java-Features wie Reflection können verwendet werden, da das Werkzeug dafür nötige Teile des Java Runtime Environment (JRE) in JavaScript emuliert. Problematischer ist die Verwendung von Drittanbieterbibliotheken, da Dragome keine Unterstützung für Mappings zu äquivalenten JavaScript-Bibliotheken bietet. Demnach müssen solche Abhängigkeiten manuell durch den portierenden Entwickler aufgelöst werden. Insgesamt verfügt Dragome somit über einen guten Funktionsumfang, der jedoch primär durch die fehlende Möglichkeit für Mappings eingeschränkt wird. Der generierte JavaScript-Code basiert auf optimiertem Java-Bytecode, weshalb die Verständlichkeit gegenüber Java-Quellcode eingeschränkt ist.

Das Werkzeug ist als kostenlos nutzbare Open Source-Software erhältlich. Der Internetauftritt gibt eine kurze Einführung in die Nutzung von Dragome sowie Tutorials und einige Codebeispiele. Insgesamt deckt die verfügbare Dokumentation allerdings nicht alle Aspekte der Nutzung ab. Fragen zur Nutzung können in einer Mailingliste oder einer Gruppe bei Google Groups gestellt werden, wobei letztere nicht sehr aktiv ist und in Beiträgen beschriebene Probleme oft nicht geklärt wurden. Der Entwickler kann für professionelle Hilfe kontaktiert werden. Die Integration von Dragome ist über Maven in beliebige Arbeitsprozesse und Entwicklungsumgebungen möglich. Plugins die die Integration und Konfiguration des Werkzeugs vereinfachen sind nicht verfügbar. Erweiterungen der Funktionalität sind aufgrund des offenen Quellcodes möglich, werden jedoch nicht durch eine vorhandenen Plugin-API unterstützt [Dra17].

Ähnlich wie Dragome nutzt auch **TeaVM** Java-Bytecode um daraus performantes JavaScript zu erzeugen. Neben Java-Projekten ist TeaVM allerdings auch mit Kotlin und Scala kompatibel. Zudem kann das Konvertierungswerkzeug neben JavaScript alternativ auch WebAssembly erzeugen. Der Hersteller sieht in TeaVM primär ein Tool um Java sowohl als Sprache für das Server-Backend als auch für das Client-Frontend verwenden zu können. Durch die Verwendung derselben Sprache kann eine tiefere Integration zwischen beiden Teilen entstehen und ein einheitliches Ökosystem geschaffen werden. Demnach folgt TeaVM einer ähnlichen Philosophie wie GWT und eignet sich laut Hersteller weniger für die Portierung komplexer Codebasen zu JavaScript und mehr für die Implementation von Web-Anwendungen in Java. Dies wird primär auf die Komplexität zurückgeführt, adäquate Mappings für die auftretenden Abhängigkeiten zu finden. TeaVM bietet die Möglichkeit existierende JavaScript-Bibliotheken in Java einzubinden. Daher müssen Abhängigkeiten zu Java-Bibliotheken vor der Konvertierung manuell durch äquivalente JavaScript-Versionen ersetzt werden. Neben JavaScript-Bibliotheken kann in Java über die TeaVM-API auch auf andere Funktionen aus JavaScript zugegriffen werden. So kann das DOM eines HTML-Dokumentes über Methoden in Java manipuliert werden. Zudem ist es möglich JavaScript-Code direkt in Java einzubinden und auszuführen. Ebenfalls können Methoden in Java in JavaScript aufgerufen werden. Diese Funktionalitäten werden im

sogenannten „Flavour“-Framework zusammengefasst. Zugriff auf die Reflection-API in Java wird von TeaVM nicht unterstützt, da dies laut Hersteller zu viele Abhängigkeiten zum JRE erfordert und somit JavaScript-Dateien unnötig umfangreich machen würde. Stattdessen umfasst TeaVM eine Metaprogramming-API, welche Reflection über eine TeaVM-eigene Implementation teilweise ermöglicht. Java 8 wird bis auf Lambda-Ausdrücke nicht unterstützt. Nach GWT und JSweet verfügt TeaVM im Vergleich über den größten Funktionsumfang. Die Übersetzung mit TeaVM erzeugt sehr performantes JavaScript, welches jedoch nicht durch Kontrollflussrekonstruktion für JavaScript-Entwickler aufgearbeitet wird. Darunter leidet wie auch bei Dragome die Verständlichkeit und Wartbarkeit.

TeaVM ist Open Source-Software und kann somit kostenlos genutzt und durch den Nutzer programmatisch erweitert werden. Der genutzte Compiler ist modular aufgebaut und kann durch eigene Module erweitert werden. Diese Module ermöglichen benutzerdefinierte Interaktionen des Compilers mit dem betrachteten Bytecode. So wird eine gute Möglichkeit für die Erweiterung des Werkzeugs gegeben. Um das Konvertierungswerkzeug zu nutzen wird Maven benötigt. Weiterhin ist ein Plugin für die IDE IntelliJ IDEA verfügbar, welches die Erstellung von TeaVM-Projekten vereinfacht und zudem das Debuggen des Projektes unterstützt. In der Konfiguration des Werkzeugs kann die Erzeugung von Source Maps aktiviert werden, welche Quelldateien in Java und JavaScript miteinander verknüpfen. Diese Informationen können zum Debuggen im Browser genutzt werden, wobei für den Browser Chrome ein TeaVM-Debugging-Plugin zur Verfügung steht, welches Informationen aus dem TeaVM-Debugger für den Browser aufbereitet. Für die Konfiguration des Werkzeugs steht eine Vielzahl von Parametern bereit, die hinreichend dokumentiert sind. Dazu gehören die Wahl des Outputverzeichnisses, Einstellung der Verschleierung von Bezeichnern im Generat und der Aggressivität der JavaScript-Optimierungen. Ein Beibehalten der Bezeichner und wenige Code-Optimierungen können dazu beitragen, bei Abstrichen bei der Performance, die Wartbarkeit des JavaScript-Codes zu erhöhen. Insgesamt lässt sich TeaVM gut integrieren und einfach konfigurieren. Das Werkzeug ist auf dem dazugehörigen Internetauftritt und auf Github umfangreich und aktuell dokumentiert. Die Benutzungshinweise werden durch Erläuterungen der zusätzlichen Codebibliotheken und Codebeispiele ergänzt. Zudem kann das Werkzeug online in einer Live-Sandbox getestet werden. In der vom Hersteller betreuten Gruppe auf Google Groups herrscht nur geringe Aktivität, allerdings kann auch direkter Kontakt zum Hersteller aufgenommen werden, wobei allerdings kein Angebot für professionelle Projektunterstützung ersichtlich ist [Tea17].

Das letzte gefundene Werkzeug ist der Java zu JavaScript Cross-Compiler, der aus der Portierungsmethodik **XMLVM** von Puder et al. hervorgegangen ist. Wie bereits in Kapitel 2 erwähnt, basiert diese akademische Arbeit auf der Umwandlung von Bytecode in eine programmiersprachenunabhängige XML-Repräsentation, aus welcher wiederum mit Codegeneratoren Quellcode in anderen Sprachen erzeugt werden kann. Ursprünglich diente XMLVM der Konvertierung von Java zu Objective-C, allerdings wurde in der weiteren Entwicklung Unterstützung für weitere Zielsprachen, unter anderem auch JavaScript, implementiert. Das Projekt wird aktuell nicht mehr aktiv entwickelt, wobei das letzte Update 2014 erschienen ist. Die Dokumentation wurde 2011 das letzte Mal aktualisiert und ist somit stark veraltet. Die vorhandene Dokumentation umfasst hauptsächlich Erläuterungen zur XMLVM-Portierungsmethode, zu welcher zudem zwei wissenschaftliche Veröffentlichungen erschienen sind ([Pud10], [PWZ13]). Nutzungshinweise zur Projekteinrichtung und Anwendung

des Tools sind lediglich für die Portierung von Android-Applikationen zu iOS vorhanden, die Portierung zu JavaScript ist nicht erläutert. Auch die Konfiguration des Konverters ist nicht für den Anwendungsfall der Konvertierung zu JavaScript erklärt. Der Kontakt zur Community wird theoretisch durch eine öffentliche Mailingliste ermöglicht. Insgesamt erschwert die mangelhafte Dokumentation der Werkzeugbenutzung und die Inaktivität des Projektes eine sinnvolle Nutzung.

Die Übersetzung zu JavaScript basiert bei XMLVM auf der Emulation einer Stack-basierten virtuellen Maschine. Dies wird erreicht indem die aus Java-Bytecode erzeugte intermediäre XML-Repräsentation so angeordnet wird, dass sie das Verhalten einer solchen Maschine nachbildet. Dieses Vorgehen verändert den in Java erkennbaren Kontrollfluss und erschwert das Verständnis des erzeugten JavaScripts zusätzlich durch die Einführung von zahlreichen Hilfsvariablen. In JavaScript nicht verfügbare Sprachkonzepte wie Klassendefinitionen und Java-UI-Elemente werden durch Aufrufe des JavaScript-Frameworks Qooxdoo ersetzt. Dieses definiert ein Java-ähnliches Objektmodell und kann Swing-Komponenten durch Äquivalente in JavaScript nachbilden. Abhängigkeiten zu externen Codebibliotheken können von XMLVM nicht gemappt werden. Laut Hersteller liegt der primäre Fokus des JavaScript-Generators von XMLVM darauf, Java-Anwendungen mit Benutzeroberflächen in AWT- oder Swing im Browser lauffähig zu machen. Da die Entwicklung des Projektes eingestellt wurde, ist keine Unterstützung moderner Java-Sprachfeatures im Werkzeug vorhanden, wodurch der Funktionsumfang eingeschränkt ist. Eine programmatische Erweiterung des Konverters ist möglich, da das Projekt Open Source ist. Der Konverter kann über die Kommandozeile bedient werden. Eine Integration in Entwicklungsumgebungen ist darüber hinaus nicht implementiert [XML11].

## 4.3. Werkzeugauswahl

Aus der Menge an gefundenen Konvertern muss für ein Portierungsprojekt die am besten geeignete Werkzeugunterstützung ausgesucht werden. Dabei sollte die Auswahl von den vom Projekt vorgegebenen Umständen abhängig gemacht werden. Für die Portierung einer bestehenden Java-Software zu JavaScript eignen sich unter Umständen andere Tools als für die Neuentwicklung einer Software mit Crossplattform-Ansatz. Dieses Kapitel vergleicht die gefundenen Tools und erläutert, welches Werkzeug für die Portierung der hier betrachteten Software DESMO-J ausgewählt wurde. Tabelle 4.2 gibt eine Übersicht über die Bewertung der Kriterien für die jeweiligen Werkzeuge.

Zur Bestimmung der allgemeinen Eignung eines Tools werden wie zuvor beschrieben die in Abschnitt 4.1 festgelegten Kriterien herangezogen werden. Für die Quellcode-Übersetzung ist das Kriterium des Funktionsumfangs besonders wichtig, da dieser angibt welche Sprachkonstrukte übersetzbar sind und somit im Ausgangscode auftreten dürfen. Je geringer der Funktionsumfang, desto größer sind die Limitationen und somit der vorher aufzubringende Anpassungsaufwand bei einer Verwendung des Tools. Primär ausschlaggebend sind beim Funktionsumfang die vom Konverter übersetzbaren Sprachkonzepte und die Unterstützung der Java-API der Java Runtime. Für beides ist auch die maximal unterstützte Java-Version ausschlaggebend. Alle gefundenen Konverter unterstützen die Sprachversion Java 7, wobei einige auch Sprachfeatures des 2014 erschienenen Java 8 übersetzen können. Eine volle Abdeckungen der Neuerungen in Java 8 verspricht jedoch nur Dragome. GWT und JSweet ermöglichen über `java.lang` hinaus die Verwendung weiterer Packages des JRE. Die entsprechenden Aufrufe werden durch die Konverter in JavaScript emuliert.

Vielfach erfordern die Werkzeuge jedoch den Einsatz der von ihnen angebotenen Bibliotheken, um nicht unterstützte Aufrufe vor der Konvertierung durch konvertereigene Implementationen zu ersetzen. Diese Substitution dient dazu, Abhängigkeiten zum JRE zu minimieren und somit die Konvertierung zu vereinfachen. Allerdings erfordert diese Vorgehensweise eine Abänderung des ursprünglichen Quellcodes in Java. Die Einführung konverterspezifischer Logik kann bei einer in dieser Arbeit angestrebten, anschließenden parallelen Weiterentwicklung beider Versionen unerwünscht sein. Die Konverter Bck2Brwsr, Dragome, GWT, JSweet und TeaVM bieten Möglichkeiten für solche Emulationen. Die SDKs dieser Werkzeuge umfassen zudem APIs für die Einbeziehung von JavaScript-typischen Sprachelementen in den Java-Code. So können Variablen unter GWT und JSweet sowohl Java- als auch JavaScript-Typen verwenden. Dragome, JSweet und TeaVM bieten die Möglichkeit durch Emulation Funktionen der Reflection-API in Java zu benutzen. Des Weiteren verfügen diese Konverter über Möglichkeiten Benutzeroberflächen zu übersetzen, wenn die in Java verwendeten Swing-Komponenten zuvor durch die in den SDKs angebotenen Implementierungen von Widgets ersetzt werden. Dies ist allerdings für den Kern von DESMO-J größtenteils irrelevant, da das Framework kaum UI besitzt. Lediglich bei der Übersetzung der für die grundlegende Funktionalität nicht ausschlaggebenden grafischen Tools wäre dieser Aspekt zu berücksichtigen. Eine Einschränkung von JavaScript gegenüber Java ist der fehlende Support von Multithreading. Keines der Tools bietet eine Möglichkeit diese Limitation zu umgehen. Dementsprechender Code führt zu Fehlern oder wird ignoriert. Ein weiterer wichtiger Punkt des Funktionsumfangs ist der Umgang mit im Java-Code eingebundenen Codebibliotheken. Ein automatisches Mapping von Bibliotheksaufrufen, beispielsweise anhand einer hinterlegten (und potentiell erweiterbaren) Liste, wird von keinem der Tools unterstützt. JSweet bietet die Möglichkeit über Adapter Aufrufe auf JavaScript-Bibliotheken in Java-Code zu integrieren. Bestehende Aufrufe müssen also manuell auf äquivalente Bibliotheken in JavaScript umgeleitet werden. Die anderen Tools erfordern im Gegensatz dazu jedoch eine Anpassung des JavaScript-Codes nach der Konvertierung oder eine exklusive Nutzung der Konverter-SDKs.

Die offiziell vom jeweiligen Tool unterstützten Sprachkonzepte werden in der Regel semantisch korrekt wiedergegeben. Dennoch haben Sprachkonzepte, die nicht übersetzt werden können, zumeist Einfluss auf das Verhalten der Software bei der Ausführung. Ein semantisch mit dem Original übereinstimmendes Generat ist daher stark vom generellen Funktionsumfang eines Konverters abhängig. Zudem können Bugs im Konvertierungswerkzeug semantische Unterschiede hervorrufen, wobei diese Fälle oftmals erst bei der Nutzung in konkreten Anwendungsfällen ersichtlich sind.

Bei der Verständlichkeit des Generats und der daraus entspringenden Wartbarkeit zeigen sich deutliche Unterschiede zwischen den Werkzeugen. Dies liegt zum einen an dem vom Hersteller festgesetzten Anwendungskontext und zum anderen an der technischen Umsetzung. Die Konverter, die auf der Basis eines Cross-Compilers arbeiten, verwenden Java-Bytecode als Input. Bei der Kompilierung von Sourcecode zu Bytecode werden in der Regel Aspekte der Sprache, die der Lesbarkeit von Java-Code durch Menschen dienen, entfernt. Dies verringert den Umfang des Codes. Des Weiteren werden Optimierungen durchgeführt, welche die Performance des Codes steigern, die Nachvollziehbarkeit durch Entwickler jedoch weiter senken. Diese Nachteile für die Verständlichkeit übertragen sich auch auf das aus diesem Bytecode generierte JavaScript. JavaScript aus Transpilern basiert im Gegensatz dazu auf dem gut lesbaren und dem Entwickler vertrauten Java-Quellcode. Aspekte wie Bezeichner und der Kontrollfluss des Programms werden direkt in JavaScript-Syntax

übertragen. Somit ist die Verständlichkeit des Generats bei den Tools GWT, JSweet und dem Kotlin-zu-JS-Compiler deutlich höher als bei den anderen. Dies liegt primär an der hohen Strukturgleichheit zwischen Input und Output. Allerdings gibt es auch bei den Cross-Compilern Ansätze wie die Kontrollfluss-Rekonstruktion, um die Verständlichkeit zu verbessern. Angewendet wird dieses Verfahren jedoch nur von AOT JS, welches nicht verfügbar ist. Besonders das JavaScript aus Bck2Brwsr und XMLVM besitzt keinerlei Strukturgleichheit zum eingespeisten Java-Code. JSweet verbessert die Verständlichkeit durch eine automatische Übersetzung von JavaDoc-Kommentaren zu JSDoc. Bei den meisten Werkzeugen wurde die mangelhafte Verständlichkeit des Generats bewusst durch den Hersteller in Kauf genommen, da der angestrebte Anwendungsfall keine Wartung des JavaScripts vorsieht. Stattdessen soll lediglich die Java-Codebasis entwickelt und gewartet werden. GWT wird explizit mit dem Vorteil vermarktet, keine JavaScript-Kenntnis für die Entwicklung von Webanwendungen besitzen zu müssen. Lediglich JSweet sieht den Anwendungsfall einer Codeportierung und anschließenden Entwicklung des Generats explizit vor. In diesem Aspekt ist aus der Perspektive des in dieser Arbeit betrachteten Anwendungsfalls demnach JSweet am besten geeignet.

Das Kriterium der Lizenzkosten erübrigt sich anhand der gefundenen Werkzeuge. Bis auf AOT JS sind alle Tools als Open Source-Software veröffentlicht, wodurch sie kostenlos verwendet werden können. Dabei haften die Hersteller allerdings auch nicht für Risiken oder Probleme, die durch die Nutzung entstehen. Ein weiteres Kriterium für die sinnvolle Nutzung eines Werkzeugs ist der dafür vorhandenen Support. Die vorhandenen Supportmöglichkeiten wurden im vorherigen Kapitel aufgezeigt. Insgesamt kann festgestellt werden, dass die vom Hersteller angebotene Dokumentation bei GWT und JSweet besonders umfangreich und gut aufbereitet ist. Diese beiden Werkzeuge zeigen auch die umfangreichsten und aktivsten Nutzercommunities auf, was bei der Lösung von Problemen in der Praxis besonders wertvoll ist. Für eine Nutzung unzureichend ist der Support bei Bck2Brwsr und XMLVM, da die wenigen vorhandenen Informationen schwer aufzufinden und veraltet sind. Wo TeaVM und Dragome noch kleine Nutzercommunities in Foren besitzen, ist bei den restlichen Werkzeugen keine nennenswerte Community vorhanden. Der Aspekt der Langfristigkeit spielt beim Support eine wichtige Rolle, da sichergestellt sein sollte, dass der Nutzer über einen längeren Zeitrahmen hinweg aktuelle und umfassende Informationen zum Werkzeug erhalten kann. Dies ist besonders bei aktiv entwickelten Werkzeugen relevant, falls sich durch die Entwicklung Änderungen in der Nutzung ergeben. Die aktiver entwickelten Werkzeuge wie GWT, JSweet und TeaVM besitzen alle eine aktuelle Dokumentation und eine ausreichend große Nutzerbasis, um das Weiterbestehen der Projekte wahrscheinlich zu machen.

Die Konfiguration der Werkzeuge beschränkt sich zumeist auf das Anpassen der Parameter in einer Konfigurationsdatei. Diese Parameter umfassen in der Regel Aspekte wie Outputverzeichnisse, Version des generierten ECMAScripts, Verschleierung des Generats und Grad der Codeoptimierung. Da viele der Werkzeuge auf der Verwendung des mitgelieferten SDKs basieren, muss dieses in die zu konvertierenden Java-Projekte eingebunden werden. Dies ist in der Regel mithilfe von Maven möglich, welches die nötigen Codebibliotheken integriert. Eine Ausnahme stellt XMLVM dar, bei dem der Konvertierungsprozess lediglich durch die Ausführung einer Aufgabe im Build-System Ant ausgelöst wird. Durch die Verwendung von Maven und Ant können alle Werkzeuge in gängige Entwicklungsumgebungen wie Eclipse oder IntelliJ IDEA integriert werden. Zusätzlich bieten einige Tools jedoch IDE-Plugins, welche die Benutzung vereinfachen. Vielfach sind diese Plugins jedoch

nur für eine IDE verfügbar, sodass der Anwender für deren Nutzung diese IDE verwenden müsste. Somit kann unter Umständen nicht die gewohnte IDE verwendet werden, wenn die Vorteile dieser Plugins genutzt werden sollen. Lediglich GWT bietet Plugins für Eclipse und IntelliJ IDEA.

Wie bereits in Abschnitt 4.1 beschrieben sollte das verwendete Konvertierungswerkzeug die Erzeugung von Trace Links zwischen Code-Elementen in Java und JavaScript unterstützen. Keines der Tools beherrscht diese Funktion auf Basis des in Unterabschnitt 3.3.1 vorgestellten Konzepts. GWT, JSweet, TeaVM und der Kotlin-zu-JS-Compiler unterstützen die Erzeugung von Source Maps während der Übersetzung. Source Maps verknüpfen .java- oder .class-Dateien mit den daraus erzeugten .js-Dateien in einer XML-Ausgabe. Moderne Browser unterstützen das Debuggen von minimiertem und optimiertem JavaScript anhand von Source Maps, indem sie dort ausgewählte Stellen im originalen unverarbeiteten JavaScript anzeigen können. Source Maps zwischen Java und JavaScript stellen eine wenig detaillierte Form von Traceability dar. Eine Ergänzung der Werkzeuge um die Erzeugung detaillierter Trace Links erfordert eine programmatische Erweiterung. Grundlegend ist die Einführung eigener Logik bis auf AOT JS in allen Werkzeugen aufgrund des offenen Quellcodes möglich. Dragome, JSweet und TeaVM vereinfachen Erweiterungen jedoch durch eine spezifische Extension-API. Bei Dragome beschränkt sich die Art der möglichen Erweiterungen jedoch auf Refactorings von Bytecode vor der Konvertierung. Bei JSweet und TeaVM kann der Konverter durch benutzerdefinierte Interaktionen mit dem Code erweitert werden. JSweet ermöglicht den Zugriff auf den abstrakten Syntaxbaum des eingelesenen Codes. Besonders letzteres eignet sich für die Umsetzung eines Traceability-Generators im Konvertierungswerkzeug.

| Name | **AOT JS (Graal)** | **Bck2Brwsr** | **Dragome** | **GWT** | **JSweet** | **Kotlin-to-JS** | **TeaVM** | **XMLVM** |
|---|---|---|---|---|---|---|---|---|
| Kategorie | JBC2JS | JBC2JS | JBC2JS | J2JS | J2JS | Kombination | JBC2JS | JBC2JS |
| *++ (sehr gut), + (gut), 0 (ausreichend), - (mangelhaft), / (nicht bewertbar)* | | | | | | | | |
| Funktionsumfang | / | - | + | ++ | ++ | 0 | + | 0 |
| Semantische Korrektheit | / | 0 | + | ++ | ++ | 0 | + | + |
| Verständlichkeit/Wartung | / | - | 0 | ++ | ++ | + | 0 | - |
| Limitationen | / | - | 0 | + | + | 0 | + | - |
| Lizenzkosten | / | ++ | ++ | ++ | ++ | ++ | ++ | ++ |
| Langfristiger Support | / | - | 0 | ++ | ++ | + | + | - |
| Komplexität der Konfiguration | / | - | 0 | + | ++ | ++ | ++ | - |
| Integrierbarkeit | / | - | 0 | ++ | + | ++ | + | 0 |
| Erweiterbarkeit | / | - | - | - | ++ | - | + | - |

Tabelle 4.2.: Bewertung der Kriterien

| Name | **AOT JS (Graal)** | **Bck2Brwsr** | **Dragome** | **GWT** | **JSweet** | **Kotlin-to-JS** | **TeaVM** | **XMLVM** |
|---|---|---|---|---|---|---|---|---|
| Kategorie | JBC2JS | JBC2JS | JBC2JS | J2JS | J2JS | Kombination | JBC2JS | JBC2JS |
| Output | JS | JS | JS | JS | TypeS, JS | JS | JS, WebA | JS |
| Java 8 | Teilweise | Teilweise | Ja | Teilweise | Teilweise | Teilweise | Teilweise | Nein |
| JavaDoc zu JSDoc | / | Nein | Nein | Nein | Ja | Nein | Nein | Nein |
| JavaScript API in Java | / | Ja | Ja | Ja | Ja | Nein | Ja | Nein |
| Reflection | Nein | Teilweise | Ja | Nein | Ja | Nein | Ja | Nein |
| Source Maps | / | Nein | Nein | Ja | Ja | Ja | Ja | Nein |
| Extension API | / | Nein | Ja | Nein | Ja | Nein | Ja | Nein |
| Library-Mapping | / | Nein | Nein | Nein | Ja | Nein | Nein | Nein |
| Kontrollfluss-Rekonstruktion | Ja | Nein | Nein | / | / | / | Nein | Nein |
| Entwicklung aktiv | Nein (2015) | Ja | Ja | Ja | Ja | Ja | Ja | Nein (2014) |
| Multithreading | Nein | Nein | Nein | Nein | Nein | Nein | Nein | Nein |
| Inkrementelle Übersetzung | Nein | Nein | Ja | Nein | Ja | Ja | Ja | Nein |

Tabelle 4.3.: Weitere Features im Werkzeugvergleich

Tabelle 4.3 stellt einige Aspekte der Werkzeuge im Vergleich dar. Für das Werkzeug AOT JS sind nur diejenigen Aspekte eingetragen, die aus der dazu erschienenen Veröffentlichung hervorgehen. Der Punkt der Kontrollfluss-Rekonstruktion ist nur für Werkzeuge relevant, die Bytecode als Input verwenden. Werkzeuge, die Java 8 teilweise unterstützen, sind in der Lage eine Teilmenge der von Java 8 definierten Sprachkonstrukte zu verarbeiten. In der Regel handelt es sich dabei mindestens um Lambda-Ausdrücke. Je nach Geschwindigkeit des Konverters und Größe der Codebasis kann eine Übersetzung viel Zeit in Anspruch nehmen. Einige Werkzeuge sind daher in der Lage inkrementelle Übersetzungen durchzuführen. Dabei wird nur Code übersetzt, der seit der letzten Übersetzung geändert wurde. Diese Funktion kann den Übersetzungsaufwand nach der initialen Konvertierung deutlich senken.

Insgesamt zeigt dieser Werkzeugvergleich, dass für die Portierung von Java-Code zu JavaScript durchaus verschiedene Tools existieren, diese sich allerdings in den meisten Fällen nicht für die Portierung einer bestehenden Codebasis eignen. Im Gegensatz dazu versuchen die meisten Ansätze ein Entwicklungsframework zu schaffen, bei denen Client- und Server-Seite von Webapplikationen in derselben Sprache Java geschrieben werden können. Dies verbessert die Interoperabilität und ermöglicht es, sich auf Java zu konzentrieren und JavaScript nur als Mittel zum Zweck als Output zu bekommen. Die tatsächliche Portierung von bestehendem Java-Code zu ebenso wartbarem und weiter entwickelbarem JavaScript ist oftmals nicht angestrebt. Unter Umständen ist diese Tatsache auf die starken Sprachunterschiede zurückzuführen die zwischen Java und JavaScript auftreten. Einige dieser Unterschiede werden jedoch durchaus durch die Werkzeuge adressiert und automatisch oder mit Unterstützung von SDK-Funktionen überbrückt. Insofern ist deren Nutzung für eine Portierung durchaus möglich, allerdings würde die Toollandschaft sicherlich von einem Konverter profitieren, der die Portierung explizit unterstützt.

Ein besonders wichtiger Aspekt der Portierungsmethode ist die parallele Evolution der Original- und portierten Version. Um dies zu ermöglichen muss der JavaScript-Code von Entwicklern wartbar und erweiterbar sein. Deshalb ist die Verständlichkeit des Generats von besonderer Wichtigkeit. Gerade dieser Aspekt wird von den existierenden Werkzeugen im Grunde nicht berücksichtigt. Stattdessen wird Wert auf besonders performantes JavaScript gelegt unter der Prämisse, dass die Verständlichkeit des JavaScripts nicht ausschlaggebend ist. Unter den betrachteten Konvertern produzieren nur die Transpiler Code der zur Java-Version strukturgleich ist. Der Kotlin-zu-Java-Compiler hat jedoch den Nachteil, dass die Codebasis einer Java-Software vor der eigentlichen Portierung zu Kotlin überführt werden muss. Trotz Werkzeugunterstützung erfordert dies weiteren manuellen Aufwand. Im Falle einer Software auf Kotlin-Basis ist die Kompilierung durchaus eine sinnvolle Alternative, insbesondere wenn das Tool in Zukunft weiter an Reife gewinnt. Ansonsten sind besonders GWT und JSweet als reife Tools anzusehen. Am Funktionsumfang von GWT und aus der Dokumentation lässt sich ersehen, dass auch GWT primär für die Umsetzung von Webanwendungen in Java ausgelegt ist. Dennoch lässt sich der GWT-Transpiler als eigenständige Komponente mit guten Ergebnissen zur Portierung nutzen. Im Rahmen dieser Arbeit soll jedoch JSweet verwendet werden, da dieses Tool mit seinem Funktionsumfang und der Möglichkeit API-Mappings auf JavaScript-Bibliotheken durchzuführen die besten Voraussetzungen für eine erfolgreiche Portierung hat. Es gilt allerdings während der Portierung zu prüfen ob eine Portierung ohne Anpassungen des Originalcodes durch JSweet-APIs sinnvoll möglich ist. Dies wäre bei einer aktiv entwickelten Software erstrebenswert, da in einer Java-Umgebung produktiv eingesetzter

Code keine Abhängigkeiten zu Codegeneratoren enthalten sollte. Solche Abhängigkeiten ließen sich umgehen indem einmalig eine Version von DESMO-J durch Elemente des Konverters-SDKs ergänzt wird und Änderungen anschließend parallel in beiden Versionen implementiert werden. Dadurch wird jedoch die Möglichkeit eliminiert, den Java-Code bei Änderungen mit wenig Aufwand neu zu übersetzen. Abschnitt 6.1 setzt sich mit der Auswahl der Portierungsstrategie auseinander.

## 4.4. Weitere Toolunterstützung

Ergänzend zum Codekonverter JSweet kommen bei der Konvertierung weitere Werkzeuge zum Einsatz. Grundlegend wird für die Sichtung und Bearbeitung von Java-Code die von der Eclipse Foundation entwickelte integrierte Entwicklungsumgebung Eclipse (in der Version 4.7.1) verwendet [Ecl18a]. Primärer Grund für die Auswahl dieser IDE ist die Verfügbarkeit des JSweet-Plugins, welches es ermöglicht, JSweet-Projekte in Eclipse zu verwalten und den Codekonverter direkt einzubinden. Dadurch sinkt die Komplexität der Konfiguration und Ausführung des Konverters. Voraussetzung für den Betrieb von JSweet sind wie bereits oben erwähnt Maven und Node.js. Unterstützung für ersteres wird in Eclipse mitgeliefert, wohingegen letzteres zusätzlich installiert werden muss. Das Eclipse-Plugin liefert bei fehlendem Node.js eine deutlich verständliche Fehlermeldung. Um den im Laufe der Portierung entstehenden JavaScript-Code mit Unterstützung von IDE-Features anschauen und bearbeiten zu können, kann Eclipse durch weitere Plugins erweitert werden. Das Plugin „Eclipse Web Developer Tools“ liefert beispielsweise spezifische Editoren für JavaScript-, HTML- und XML-Dateien [Ecl18b]. Durch das Hinzufügen dieser Werkzeuge kann auch der Aspekt der nachträglichen Anpassung des Generats in Eclipse integriert werden. Somit können Programmschnittstellen wie zum Beispiel zu externen JavaScript-Editoren vermieden werden.

Für die in Phase eins der Portierungsmethode durchzuführende Abhängigkeitsanalyse bietet sich die Nutzung die in der Java-IDE IntelliJ IDEA integrierten Abhängigkeitsanalyse an. Diese ermöglicht eine schnelle Auflistung der Abhängigkeiten in einem frei wählbaren Suchraum. So können zum Beispiel alle Aufrufe einer bestimmten Bibliothek aufgezeigt werden. In Sachen Konfigurierbarkeit und Einfachheit der Nutzung sind die Abhängigkeitsanalyse-Tools von IntelliJ IDEA den in Eclipse integrierten überlegen.

In Phase zwei der Portierungsmethode sollen Trace Links zwischen äquivalenten Codeelementen in Java und der JavaScript-Konversion erzeugt werden. Dazu bieten sich verschiedene Möglichkeiten. Zum einen könnte die Generierung der Traceability als Erweiterung in JSweet integriert werden. So würde man eine sehr hohe Genauigkeit erreichen, da die Trace Links zu einem Codeelement direkt bei dessen Übersetzung erzeugt werden könnten. Allerdings ist eine solche Funktionalität für JSweet bisher nicht verfügbar, weshalb eigener Entwicklungsaufwand entsteht. Alternativ kann auf bereits existierende Tools für die Erzeugung von Traceability zurückgegriffen werden. Im Vorfeld dieser Arbeit ist im Rahmen einer Studie ein Plugin für IntelliJ IDEA entstanden, welches die nachträgliche Ermittlung von Trace Links zwischen zwei Implementationen einer Software unterstützt. Dabei werden die Bezeichner und die Struktur in den Codebasen der Projekte analysiert und Codeelemente über Information Recovery-Verfahren mit einem Ähnlichkeitswert bewertet. Je höher der Ähnlichkeitswert, desto höher die Wahrscheinlichkeit einen korrekten Trace Link gefunden zu haben [Gre17]. Durch die anschließende manuelle Prüfung der gefundenen Links kann

Traceability zwischen den Projekten erzeugt werden. Das in der Studie vorgestellte Tool ist nur für IntelliJ IDEA verfügbar, zudem verfügt lediglich über Quellcode-Parser für Java und Swift.

Tilmann Stehle hat das Plugin seitdem weiterentwickelt und es in Vorbereitung auf das hier beschriebene Portierungsprojekt um einen Parser für JavaScript erweitert. Mit dessen Hilfe kann das Werkzeug auch den erzeugten JavaScript-Code analysieren und die dort vorhandenen Codeelemente mit denen aus der Java-Version vergleichen. Zudem wurde für das Tool ein Eclipse-Plugin entwickelt um es besser in die hier verwendete Werkzeugumgebung integrieren zu können[1]. So kann die Entwicklung, die Nutzung des Konvertierungswerkzeuges und die Erzeugung von Trace Links in derselben Entwicklungsumgebung ausgeführt werden. Dadurch werden fehleranfällige Schnittstellen vermieden. Das weiterentwickelte Tool enthält einige Leistungsverbesserungen, die den Parsingvorgang gegenüber dem ursprünglichen Tool deutlich beschleunigen. Der Nutzer kann innerhalb des Codeeditors Klassen oder Methoden markieren und für die markierten Codeelemente gefundene Übereinstimmungen in der JavaScript-Version anzeigen lassen. Ein Klick auf einen Link öffnet die JavaScript-Datei, die das Codeelement enthält. So kann der Nutzer schnell in den konvertierten Code springen, um die Übersetzung zu prüfen oder bei einer Weiterentwicklung etwaige manuelle Anpassungen durchzuführen.

[1] Update-Seite für Eclipse: https://swk-www.informatik.uni-hamburg.de/ stehle/eclipseupdatesite/site.xml

# 5. Konzeptioneller und architekturbezogener Änderungsbedarf bei der Portierung

Nach dem der Abschnitt 3.4 die Programmiersprachen Java und JavaScript vorgestellt und bezüglich ihrer Nutzungskontexte eingeordnet hat, sollen im Folgenden die Gemeinsamkeiten und Unterschiede der Sprachen betrachtet werden. Dabei wird JavaScript auf Basis von ECMAScript 5 betrachtet, da dies die Zielversion des gewählten Konvertierungswerkzeugs ist. Unterstützung für neuere Versionen der Sprachspezifikation ist nicht umfassend vorhanden. Um Java zu JavaScript zu konvertieren, muss geprüft werden, ob und inwiefern sich die Sprachkonstrukte der einen in der anderen Sprache abbilden lassen. Daher sollen die Sprachfeatures verglichen und aufeinander abgebildet werden. Besonders relevant sind dabei Funktionalitäten oder Features, welche sich nicht eins zu eins abbilden lassen. Dies betrifft sowohl Sprachkonzepte, als auch Architekturen, in denen diese verwendet werden. Um diese Codeelemente dennoch abzubilden ergibt sich während der Portierung Änderungsbedarf. Dieser Änderungsbedarf unterteilt sich in drei verschiedene Kategorien:

- Transformationsvorbereitende Refactorings
- Automatisierte Änderungen bei der Übersetzung durch das Portierungswerkzeug
- Auf die automatische Übersetzung folgende Nachbearbeitung des Generats

Der Abschnitt 5.2 zeigt Beispiele für die ersten beiden Kategorien von Änderungsbedarf auf. Für die Kategorie der automatisierten Änderungen werden Änderungen, welche im Funktionsumfang des ausgewählten Werkzeugs JSweet verfügbar sind, betrachtet. Nach der Übersetzung durchgeführte Änderungen des Generats umfassen hauptsächlich JavaScript-Reimplementierungen von Bibliotheksaufrufen, deren Abhängigkeiten nicht aufgelöst werden konnten. Diese Kategorie ist daher stark projektspezifisch. Beispiele für diese Kategorie werden bei der Beschreibung der konkreten Portierung von DESMO-J in Kapitel 6 angeführt.

## 5.1. Gemeinsamkeiten und Unterschiede von Java und JavaScript

Sowohl Java als auch JavaScript können in die Kategorie der objektorientierten Programmiersprachen einsortiert werden. Das bedeutet, dass beide Sprachen das Konzept von Objekten kennen und nutzen. Ein Objekt ist dabei ein Container mit verschiedenen Eigenschaften. Grundsätzlich besitzt ein Objekt

einen Zustand und ein Verhalten. Der Zustand wird durch die Attribute eines Objekts und deren Werte bestimmt, wohingegen das Verhalten durch die Methoden bzw. Funktionalitäten definiert wird. Diese Funktionalitäten werden in Methoden (Java) oder Funktionen (JavaScript) implementiert, welche als Input Übergabewerte erhalten und einen bestimmten Output als Rückgabewert liefern. Innerhalb der Methode können die Attribute eines Objekts verwendet und manipuliert werden, wodurch sich der Zustand des Objekts ändern kann. Objekte besitzen zusätzlich zu Zustand und Verhalten eine Identität, welche über die Lebensdauer des Objekts gleichartig bestehen bleibt. Diese grundlegenden Konzepte sind sowohl in Java als auch in JavaScript existent, wobei sich die genaue Umsetzung und somit die Nutzung von Objekten im Code unterscheidet.

Auf Objekte bezogen sind zwei besonders wichtige Unterschiede zwischen den Sprachen zu erkennen: Objekte in Java basieren auf Klassen, wohingegen sie in JavaScript auf Prototypen basieren. Um ein Objekt verwenden zu können, wird in Java eine Instanz einer Klasse erzeugt. Die Klasse ist dabei der Musterrahmen für ein Objekt dieser Art. Eine Klasse besitzt einen Klassennamen, Attribute und Methoden. Diese werden im Code innerhalb des Klassenkörpers definiert. Eine Instanz dieser Klasse wird anschließend mittels des Konstruktors erzeugt, eine spezielle Methode die für die Objekterzeugung verantwortlich ist. Mithilfe von Übergabeparametern kann eine Instanz mit einem bestimmten, sich je nach Parameter ändernden, Initialzustand erzeugt werden. Die Instanziierung gilt in Java als Objekterzeugung. In JavaScript ist der grundlegende Musterrahmen für ein Objekt nicht eine Klasse, sondern ein Prototyp. Sowohl ein Prototyp als auch ein Objekt bestehen in JavaScript aus einer Menge an Eigenschaften (Properties). Eine Eigenschaft besitzt einen Namen und bekommt einen Wert zugewiesen. Dabei kann der Wert sowohl ein Variablenwert als auch eine Funktion sein. Es wird nicht konkret zwischen Attributen und Operationen unterschieden, stattdessen wird beides als Eigenschaft definiert. In JavaScript sind im Gegensatz zu Java auch Funktionen als Objekte definiert, die einer Eigenschaft als Wert zugewiesen werden können. Der Prototyp eines Objektes `O` ist ein weiteres Objekt `P`, welches seine Eigenschaften für `O` zur Verfügung stellt. Objekte in JavaScript sind dynamisch typisiert, wodurch zur Laufzeit nicht immer sichergestellt ist, dass bestimmte Eigenschaften in einem Objekt vorhanden sind. Wenn eine Eigenschaft in `O` nicht abrufbar ist, kann stattdessen dieselbe Eigenschaft des Prototypen `P` genutzt werden. Durch die Umsetzung des Prototypen wird sichergestellt, dass eine Eigenschaft im Objekt `O` abrufbar ist. Im Folgenden werden die Unterschiede der Sprachen bezüglich der Typisierung weiter erläutert. Bei der Objekterzeugung mittels Prototypen wird die erzeugte Instanz vom Prototypen geklont. Anschließend können weitere Eigenschaften hinzugefügt oder die des Prototypen überschrieben werden. In JavaScript lässt sich durch die Umsetzung von Prototypen eine Objektstruktur aufbauen, die ähnlich einem Vererbungsbaum in Java die verschiedenen Objekte verknüpft. Die meisten JavaScript-Objekte lassen sich so auf den Grundtyp `Object.prototype` zurückführen, welcher grundlegende Eigenschaften wie `toString()` oder `valueOf()` definiert. Diese stehen allen Objekten zur Verfügung die diesen Prototyp direkt oder über andere Objekte umsetzen. Weitere wichtige Prototypen sind der von Funktionen genutzte Prototyp `Function.prototype` und der von Array umgesetzte `Array.prototype`, welche grundlegende Eigenschaften für Funktionen und Arrays definieren. Diese Prototypen setzen wiederum den `Object.prototype` um. Ein ähnliches Konzept kommt in Java zum Einsatz, wobei hier alle Klassen von der Grundklasse `Object` erben. Durch Vererbung stehen einer Unterklasse die Attribute und Operationen der Oberklasse zur Verfügung. Allerdings kann ein Objekt in Java nur von einer Oberklasse erben, mehrfache Vererbung wird nicht unterstützt.

Die Zuordnung mehrerer Prototypen zu einem Objekt in JavaScript ist zwar auch nicht möglich, allerdings können Eigenschaften von weiteren Prototypen theoretisch im Konstruktor hinzugefügt werden. Dies ist als Mixin bekannt. DESMO-J nutzt Java-typische Vererbung von Klassen intensiv um eine Objekthierarchie aufzubauen, weshalb eine gleichartige Abbildung in JavaScript notwendig ist.

Der zweite deutliche Unterschied ist die statische Typisierung von Objekten in Java und die dynamische Typisierung in JavaScript. In Java werden in der Klassendeklaration alle Attribute und Methoden die eine Klasse umfassen soll angegeben. Durch die statische Typisierung kann man zur Laufzeit davon ausgehen, dass eine Instanz der Klasse diese Elemente enthält und sie somit problemlos aufgerufen werden können. Allerdings ist es während der Laufzeit nicht möglich Attribute oder Operationen hinzuzufügen, zu entfernen oder zu ändern. Lediglich die Werte von Attributen können geändert werden, nicht aber deren Namen oder andere Eigenschaften. JavaScript ist eine dynamisch typisierte Programmiersprache. Hier ist es möglich die Eigenschaften, also auch die Schnittstelle und den Wertebereich eines Objektes zu manipulieren. So können einem Objekt neue Attribute oder Operationen hinzugefügt oder bestehende entfernt werden. Auch lassen sich Operationen mit neuem Verhalten überschreiben. Nicht nur am Objekt `0` sondern auch dessen Prototypen `P` lassen sich die Eigenschaften dynamisch manipulieren. Alle Objekte, die diesen Prototypen umsetzen, bekommen daraufhin Zugriff auf die veränderten Eigenschaften. Wie oben bereits angesprochen, kann es dabei zur Laufzeit zu Problemen kommen, da nicht sichergestellt ist, dass benötigte Eigenschaften in einem Objekt verfügbar sind. Aus diesem Grund gibt es Funktionen die es ermöglichen zu prüfen, welche Eigenschaften ein Objekt zum Prüfzeitpunkt besitzt. Derartige Prüfungen sind in Java zwar über die Reflection-API möglich, durch die statische Typisierung jedoch nicht notwendig.

Ein weiterer Aspekt der Typisierung zeigt sich bei der Nutzung von Objekten und Datentypen. In Java ist der Datentyp einer Variable jederzeit eindeutig festgelegt. Im Code muss für alle Variablen und Konstanten konkret angegeben werden, um welchen Datentyp es sich handelt. Dabei kann es sich entweder um die Klasse eines Objekts bzw. Referenztypen (z.B Object, String) oder einen primitiven Datentyp (z.B. int, boolean) des JDK handeln. Bei Methoden muss in der Signatur statisch festgelegt werden wie viele Parameter in welcher Reihenfolge übergeben werden und welchen Typ sie haben. In JavaScript sind alle Variablen dynamisch typisiert, was bedeutet, dass einer Eigenschaft ein beliebiges Objekt oder sogar eine Funktion zugewiesen werden kann. Gleiches gilt auch für Funktionen bei denen beliebige Objekte in einer beliebigen Anzahl übergeben werden können. Auch wenn eine Funktion eine bestimmte Anzahl an Parametern erwartet ist es möglich mehr oder weniger Parameter zu übergeben. Daher muss innerhalb der Funktion geprüft werden, welche Parameter relevant sind und welche nicht und ob die Funktion mit den übergebenen Parametern ausgeführt werden kann. Ein Vorteil dieser Flexibilität ist die einfache Umsetzbarkeit von optionalen Parametern, was in Java nur durch Überladung einer Methode möglich ist und deutlich weniger flexibel gestaltet werden kann. Seit Java 5 können jedoch variable Parameterlisten (Varargs) mit beliebig vielen Parameter desselben Typs verwendet werden. Statisch typisierten Code in eine dynamisch typisierte Sprache zu übersetzen ist aufgrund der größeren Flexibilität der Zielsprache grundsätzlich problemlos möglich. Bei einer nativen Weiterentwicklung in der JavaScript müssen jedoch die verwendeten Typen beachtet werden, um Inkompatibilitäten zu vermeiden.

Über konkrete Klassen hinaus ermöglicht Java die Deklaration von Interfaces und abstrakten Klassen. Interfaces oder auch Schnittstellen definieren die Signatur von Operationen, die eine Klasse implementieren muss, wenn sie das Interface implementiert. Die implementierende Klasse enthält den Code für die Umsetzung der Operation, welcher dem Nutzer des Interfaces jedoch verborgen bleibt. Durch die Nutzung von Interfaces kann sichergestellt werden, dass eine implementierende Klasse den festgelegten Schnittstellenvertrag einhält. Eine Klasse kann mehrere Interfaces implementieren. Abstrakte Klassen ähneln Interfaces, indem sie Unterklassen die von ihnen erben Vorgaben für zu implementierende Operationen machen. Im Gegensatz zu Interfaces können abstrakte Klassen jedoch eigene Attribute und Methodenimplementationen enthalten. Diese werden den Unterklassen vererbt, welche zusätzlich abstrakt definierte Methoden der abstrakten Oberklasse implementieren müssen. Abstrakte Klassen dienen nur als Vorgaben und können im Gegensatz zu regulären Oberklassen nicht instanziiert werden. In JavaScript sind sowohl für Interfaces als auch für abstrakte Klassen keine Äquivalente zu finden, allerdings können Prototypen in einer ähnlichen Weise genutzt werden. Ein direktes Abbilden des Verhaltens der beiden Konzepte ist aufgrund der unterschiedlichen Paradigmen jedoch schwierig. Für diese Limitation muss bei der Portierung eine adäquate Lösung gefunden werden.

Java verfügt über eine weitere spezielle Art von Klasse, die Enumeration. Aufzählungstypen erben von der Oberklasse `Enum` und definieren eine bestimmte Menge an möglichen Werten, die eine Instanz der Aufzählung annehmen kann. Kernaspekt einer Aufzählung ist die Beschränkung der möglichen Werte auf die im Code festgelegten Konstanten. In Java soll somit verhindert werden, dass einer Variablen andere Werte als die Erlaubten zugewiesen werden können. Aufgrund der dynamischen Typisierung in JavaScript ist dieses Konzept dort nicht gleichartig wiederzufinden. Zwar kann einem Objekt eine Menge an vordefinierten Werten zugewiesen werden, diese können aber im Gegensatz zu Enumerationen in Java während der Laufzeit manipuliert werden. Dies bedeutet, dass zwar das Verhalten von Aufzählungstypen emuliert werden, aber keine Gewähr über den tatsächlichen Zustand des Objekts gegeben werden kann. Der Entwickler wird nicht wie bei Java durch den Compiler auf einen Verstoß gegen den definierten Aufzählungstyp hingewiesen, was die Benutzung von Enumerationen wie in Java erschwert.

In Java ist das Überladen von Methoden möglich. Dies bedeutet, dass mehrere Methoden denselben Namen, jedoch unterschiedliche Übergabeparameter besitzen. Dabei kann sich sowohl der Typ der Parameter als auch deren Anzahl unterscheiden. So kann eine Methode mit einer bestimmten Funktionalität mit verschiedenen Parametern aufgerufen werden, die für die Funktionalität anwendbar sind. Ein Beispiel dafür ist die Methode `notifyObservers()` der Klasse `Observable` des JDK. Diese kann sowohl ohne Parameter als auch mit einer Instanz des Typs `Object` aufgerufen werden und ist damit überladen. In beiden Fällen werden die `Observer` des `Observable` benachrichtigt, wobei sie im zweiten Fall ein `Object` als Zusatzinformationen übermittelt bekommen. JavaScript erlaubt eine solche Überladung von Funktionen nicht. Wenn in JavaScript-Code mehrere Funktionen mit dem gleichen Namen definiert werden, verwendet der Interpreter ausschließlich die zuletzt definierte. Diese Einschränkung von JavaScript gegenüber Java muss daher berücksichtigt werden, wenn der Ausgangscode überladene Methoden enthält.

Mehrfach in einer Klasse auftretende Bezeichner sind jedoch nicht nur bei der Überladung ein Problem. Auch die Verwendung desselben Bezeichners als Methodenname und als Name eines Attributs sorgt in JavaScript für Probleme. Dies liegt daran, dass sowohl Variablen als

auch Funktionen in JavaScript als Eigenschaften angesehen werden. So kann eine nach einer Attributsdeklaration definierte Methode mit demselben Bezeichner `a` das Attribut überschreiben und umgekehrt. Beides wird gleichförmig der Variablen `var a` zugewiesen.

Kontrollstrukturen regeln den Kontrollfluss eines Programms. Dies beinhaltet die bedingte Steuerung des Programmes mithilfe von Verzweigungen und Schleifen. In Java kommen die folgenden Kontrollstrukturen zum Einsatz:

- Verzweigung mit `if` und optionaler Alternative mit `else`
- Mehrfache Verzweigung mit `switch`
- Bedingungsoperator (ternärer Operator) statt `if-else`
- Abweisende Schleife mit `while`
- Annehmende Schleife mit `do-while`
- Zählschleife mit `for`
- Erweiterte Zählschleife zum Iterieren über Sammlungen (`foreach`)

Alle diese Kontrollstrukturen sind auch in JavaScript vorhanden und stimmen in ihrer Funktionalität überein. Die Bedingungen der Kontrollstrukturen werden als boolesche Werte dargestellt, wobei die Gleichheitsoperatoren und logischen Operatoren in beiden Sprachen gleich sind. Aufgrund der dynamischen Typisierung gibt es in JavaScript zusätzlich den Operator `NaN` (Not a number). Mithilfe dessen lässt sich prüfen, ob ein verwendeter Wert numerisch ist oder nicht. Ein weiterer Unterschied ist die Differenzierung zwischen gleich (==) und striktem gleich (===). Reguläres gleich prüft den Wert der Objekt wohingegen striktes gleich zudem die Art des primitiven Typs mit einbezieht. So ist `0 == ' '` wahr und `0 === ' '` falsch, da die Werte zwar gleich sind, jedoch nicht die Typen. Diese Unterscheidung ist in Java aufgrund der statischen Typisierung nicht von Nöten. Um den Inhalt von Strings zu vergleichen wird in Java die `String`-Methode `equals(String)` anstatt des Gleichheitsoperators verwendet. Dieses Vorgehen ist äquivalent zu einem Vergleich mit === in JavaScript. Sowohl in Java als auch JavaScript kann das Schlüsselwort `break` verwendet werden um aus Kontrollstrukturen auszubrechen. Die Syntax für `switch`-Verzweigungen ist in beiden Sprachen äquivalent und beinhaltet einen Standardzweig mit `default` sowie die Notwendigkeit jeden Zweig mit `break;` zu schließen um die Ausführung des folgenden zu verhindern.

Programmiersprachen verwenden Variablen zum Ablegen von Daten. Dabei werden verschiedene Datentypen unterschieden, die eine Variable annehmen kann. In Java werden Datentypen in zwei grundlegende Kategorien unterteilt: primitive bzw. einfache Typen und Referenztypen. Die primitiven Datentypen sind nicht veränderbar und haben eine bestimmte, feste Größe in Bits. Die Tabelle 5.1 führt in der Spalte „Java“ alle in Java verfügbaren primitiven Datentypen mit ihrer Bitzahl auf. Es gibt mehrere numerische Datentypen und den logischen Datentyp `boolean`. Dieser besteht aus einem Bit und drückt wahr oder falsch aus. JavaScript verfügt über ein direktes Äquivalent dieses Typs mit `Boolean`. Die numerischen Datentypen teilen sich in Java in ganzzahlige Werte und Gleitkommazahlen. Nur letztere ermöglichen das Abspeichern von Werten mit Nachkommastellen. In beiden Kategorien gibt es wie in der Tabelle ersichtlich mehrere Typen mit verschiedenen

| Datentyp | Java | | JavaScript |
|---|---|---|---|
| Numerische Werte (ganzzahlig) | byte<br>short<br>int<br>long | 8 Bit<br>16 Bit<br>32 Bit<br>64 Bit | |
| Numerische Werte (Gleitkomma) | float<br>double | 32 Bit<br>64 Bit | Number |
| Logischer Wert | boolean | 1 Bit | Boolean |
| Zeichen | char | 16 Bit<br>16 bit pro Zeichen | String |

Tabelle 5.1.: Vergleich der primitiven Datentypen

Bitzahlen. Höhere Bitzahlen erlauben einen höheren Wertebereich für den Datentyp. In JavaScript gibt es diese Unterteilung in verschiedene numerische Datentypen nicht. Stattdessen werden alle numerischen Werte mit dem Typ `Number` ausgedrückt. `Number` erlaubt das Abspeichern von 64 Bit langen Gleitkommazahlen und ist somit mit dem `double` in Java gleichzusetzen. JavaScript verfügt nicht über einen speziellen Datentyp für ganzzahlige Werte. Im Gegensatz zu dem primitiven Typ `double` in Java verfügt `Number` über die zusätzlichen Werte `+infinity` und `-infinite`, sowie `NaN` (Not a number). Zusätzliche Methoden ermöglichen das Abrufen der größten oder kleinsten möglichen Zahl. Ähnliche Funktionalität kann in Java in den Klassen `Double` (`Integer`, usw.) gefunden werden. Diese Klassen sind auf Referenztypen basierende Wrapper für die korrespondierenden, primitiven Datentypen. Ein weiterer primitiver Datentyp in Java ist `char`, welcher ein einzelnes Unicode-Zeichen speichern kann. Aufgrund der Art der Speicherung des 16 Bit langen Zeichencodes gehört auch `char` zu den numerischen Typen. JavaScript besitzt kein direktes Äquivalent für `char`, da einzelne Zeichen auch über den primitiven Typ `String` für Zeichenketten realisiert werden können. Strings gehören in Java zu den Referenztypen. JavaScript besitzt über die in der Tabelle aufgezeigten primitiven Typen hinaus noch drei weitere: `Null`, `Undefined` und seit ECMAScript 6 `Symbol`. `Null` repräsentiert eine ungültige oder fehlende Referenz und `Undefined` den Wert einer Variable nach der Deklaration, bevor ein anderer Wert zugewiesen wurde. Äquivalent dazu ist in Java für beide Fälle der Wert `null`. Neben den primitiven Typen gibt es zudem den Grundtypen `Object`, welcher als Sammlung beliebiger Eigenschaften anzusehen ist. Demnach stimmt das Typ `Object` mit dem Basis-Referenztyp `Object` in Java überein. Sowohl in Java als auch in JavaScript gibt es weitere Objekte in der Standardbibliothek, welche auf den Basistypen basieren und spezifische Funktionen erfüllen. Dazu gehören Typen für Objektsammlungen, mathematische Operationen, Datumsoperationen oder Fehlerbehandlung. Diese Typen werden in der Java-Spezifikation ([Ora17]) und in der Spezifikation des ECMAScript ([ECM17]) beschrieben. Sie lassen sich oftmals aufgrund der unterschiedlichen Schnittstellen nicht direkt von einer Sprache auf die andere abbilden.

Beide betrachteten Sprachen verfügen über Sprachkonstrukte, die mehrere Werte zusammenfassen können. Das einfachste dieser Konstrukte ist in Java das Array. Arrays sind Datentypen, die eine festgelegte Anzahl an Werten eines Typs enthalten können. Die Anzahl der möglichen Werte wird bei der Erzeugung des Arrays festgelegt. In Java als typisierte Sprache muss zudem festgelegt werden, welchen Datentyp ein Array aufnehmen können soll. Dies können primitive Typen, Referenztypen oder andere Arrays sein. Über letztere Möglichkeit können mehrdimensionale Arrays erzeugt werden. Die einzelnen Felder eines Arrays sind ab dem Zähler 0 aufwärts durchnummeriert, sodass

```
var map = {};
function put(key, value) {
  map[key] = value; //Wert value wird dem Index key zugewiesen
}

put("Key", "Value");
if("Key" in map)
  console.log(map["Key"]); //>Value
```

Quelltext 5.1: Realisierung einer Map in JavaScript

einzelne Elemente durch die Angabe eines gültigen Index abgerufen werden können. JavaScript verfügt über ähnlich aufgebaute Arrays, wobei durch die dynamische Typisierung der Sprache nicht angegeben werden muss, welchen Datentyp es enthält. Auch der Array-Typ wird nicht wie in Java konkret angegeben (z.B. `int[] a = {3, 5};`), sondern lediglich als reguläre Variable deklariert (`var a = [3, 5];`). In beiden Sprachen beginnt die Zählung der Indizes bei 0. Beim Hinzufügen von Werten zu einem Array in Java wird durch den Compiler geprüft, ob der Typ mit dem Array-Typ kompatibel ist. Inkompatible Typen sorgen für einen Fehler. Ähnliches tritt in JavaScript nicht auf, da ein Array dort beliebige Werte enthalten kann. Zudem kann ein Array in JavaScript mit der Methode `push(item)` um zusätzliche Werte erweitert werden. Dies ist in Java aufgrund der festen Länge nicht möglich. Das Verhalten von Java-typischen Arrays kann jedoch mit JavaScript-Arrays gleichartig umgesetzt werden.

In Java können neben Arrays auch andere Datenstrukturen für das Speichern von Werten genutzt werden. Diese sind Teil der Collections-API und umfassen unter anderem die Datenstrukturen Liste und Map. Listen bilden eine Menge von Daten in einer bestimmten Reihenfolge ab, wohingegen Maps assoziative Speicher darstellen, deren Werte mit einem Schlüssel verknüpft sind. Sowohl für Listen als auch Maps existieren in der API mehrere Implementationen, welche gemeinsame Interfaces implementieren. Grundlegend nehmen Collections in Java Objekte jeden Typs auf, sie können jedoch mittels Generics auf bestimmte Typen eingeschränkt werden. Durch die Angabe eines spezifischen Typs für eine Collection kann eine bessere Typsicherheit gewährleistet werden, da die Typen vom Compiler geprüft werden. Reines JavaScript verfügt nicht über äquivalente Datentypen, allerdings lässt sich das Verhalten von Listen und Maps mithilfe von Arrays nachbilden. Für Listen kann zuerst ein leeres Array erzeugt werden, welchem anschließend über die `push(item)`-Methode beliebige Objekte hinzugefügt werden können. Dies entspricht der Listen-Methode `add(Object)` in Java. Die Reihenfolge des Hinzufügens bleibt durch den ansteigenden Index erhalten. Wie bereits erwähnt kann allerdings die Typsicherheit nicht automatisch geprüft werden. Theoretisch ist es jedoch möglich, Objekte vor dem Hinzufügen auf vorhandene Eigenschaften zu prüfen, jedoch schließt auch dies nicht das direkte Hinzufügen von Objekten anderen Typs aus. Auch Schlüssel-Wert-Paare einer Map können in JavaScript-Arrays abgelegt werden. Dazu darf jedoch nicht die Methode `push(item)` genutzt werden, da diese den nächsten Index als Schlüssel für den an diesem Index liegenden Wert verwendet. Stattdessen kann dem Array der Wert an einem beliebigen und benutzerdefinierten Index bzw. Schlüssel eingefügt werden. Quelltext 5.1 zeigt den dafür notwendigen Code. Um den Wert zu einem Schlüssel aus der Map zu erhalten, wird mit dem Schlüsselwort `in` geprüft ob der Index (bzw. die Eigenschaft) in `map` vorhanden ist und anschließend der korrespondierende Wert abgerufen.

Die Collection-Typen Queue und Stack können auf ähnliche Weise in JavaScript über Arrays realisiert werden. Dabei helfen die Methoden `push()`, `pop()` und `slice()` des Prototypen von JavaScript-Arrays. Die Umsetzung von Java-Collections über Arrays in JavaScript ist für die Portierung von DESMO-J notwendig.

Ein wichtiges Konzept der Programmierung ist die Ausnahmebehandlung, also der Umgang mit Problemsituationen wie fehlschlagende Aufrufe oder fehlender Zugriff auf notwendige Ressourcen. Dabei geht es um Situationen, die vorab antizipiert werden können und bei deren Auftreten alternativer Code ausgeführt werden soll, um das Abstürzen des Programms zu verhindern. In Java können derart problematische Aufrufe in einem `try{}`-Block getätigt werden. Dieser Bereich wird überwacht um auftretende Ausnahmen abzufangen. Ausnahmen sind in Java Objekte des Typs `Exception`, wobei Unterklassen für bestimmte Arten von Ausnahmen im JDK definiert sind oder selber geschrieben werden können. Wenn im überwachten Code eine Ausnahme auftritt, wird die Codeausführung abgebrochen und stattdessen der `catch(Exception){}`-Block ausgeführt. Dabei können mehrere Catch-Blöcke definiert werden, die verschiedene Klassen von Exceptions abfangen. JavaScript bedient sich bei der Ausnahmebehandlung an demselben Mechanismus. Die Syntax für die Code-Blöcke stimmt mit der von Java überein. Beide Sprachen ermöglichen zudem einen `finally{}`-Block, welcher sowohl im Fehlerfall als auch bei normaler Ausführung abschließend ausgeführt wird. Unterschiede zeigen sich jedoch bei den verwendeten Ausnahmeobjekten. Statt `Exception` verwendet JavaScript Objekte des Typs `Error`. Wie auch in Java verfügt JavaScript über konkretere Fehlertypen, die `Error` als Prototyp umsetzen. Auch können Entwickler nach diesem Muster eigene Fehlertypen deklarieren. Allerdings erlaubt JavaScript nicht das selektive Behandeln von verschiedenen Typen von Ausnahmen in unterschiedlichen Catch-Blöcken. Sobald ein Catch-Block definiert wird, fängt dieser alle Typen von Ausnahmen und nicht nur bestimmte Arten. Daher muss innerhalb des Catch-Blocks überprüft werden, welcher Typ von Ausnahme gefangen wurde und was daraufhin geschehen soll. In beiden Sprachen können Ausnahmen explizit über das Schlüsselwort `throw` geworfen werden. JavaScript erlaubt dabei das Werfen eines beliebigen Typs als Ausnahme, allerdings enthalten nur Objekte des Typs `Error` Metainformationen über den Fehler. Für das in Java mögliche Weiterleiten einer Ausnahme in eine höhere Ebene mit `throws` in der Methodensignatur gibt es in JavaScript keine Entsprechung. Stattdessen muss eine Ausnahme eingefangen und neu geworfen werden um dieses Verhalten nachzubilden. [Ull15], [Hav14]

Insgesamt kann festgestellt werden, dass eine Vielzahl an Sprachkonzepten der beiden betrachteten Sprachen einander so sehr ähnelt, dass eine Abbildung der Syntax aufeinander möglich ist. Viele Konzepte wie Objekt-Sammlungen oder Enumerationen können mit den in JavaScript vorhandenen Sprachelementen nachgebildet werden. Dabei ist die Abbildung von ursprünglich statisch typisierten Objekten auf das dynamische Typsystem von JavaScript unproblematisch. Der umgekehrte Portierungsweg von dynamischen Typen in JavaScript zu statischen Typen in Java wäre aufgrund der wegfallenden Flexibilität deutlich komplexer [TV00].

## 5.2. Behandlung von konzeptionellen Unterschieden

Dieser Abschnitt erläutert, wie die identifizierten Sprachunterschiede und Abbildungen im Rahmen einer Portierung umgesetzt werden können. Dafür werden zuerst die Änderungsbedarfe betrachtet,

die automatisch vom verwendeten Konvertierungswerkzeug JSweet adressiert werden können. Grundsätzlich ist für die Minimierung des Portierungsaufwands eine toolgestützte Bearbeitung des Änderungsbedarfs vorzuziehen. Anschließend werden transformationsvorbereitende Refactorings betrachtet, die manuell vom portierenden Entwickler vor der Anwendung automatischer Konvertierungswerkzeuge durchgeführt werden müssen. Diese Refactorings erhöhen den manuellen Aufwand und dienen dazu, nicht vom Konverter unterstützte Codeelemente so abzuändern, dass sie anschließend durch das Werkzeug übersetzt werden können.

### 5.2.1. Änderungen durch den Konverter

Alle Arten von Klassendeklarationen in Java können von JSweet übersetzt werden. Dabei soll der vom Konverter erzeugte Code das Verhalten von Java-Klassen in JavaScript emulieren. Quelltext 5.2 deklariert eine einfache Klasse, welche durch den Konverter zu Quelltext 5.3 übersetzt wird. In JavaScript wird ein Objekt mit dem Namen der Klasse erzeugt, welchem eine Funktion als Wert zugewiesen wird. Innerhalb dieser Funktion werden die Eigenschaften der Klasse definiert. Dies sind im Beispiel zum einen der Konstruktor `BankAccount(){...}` und zum anderen die Methode `deposit(){...}`. Auffällig ist, dass das deklarierte Attribut `name` in der Konstruktor-Funktion deklariert wird. Dies liegt daran, dass die Funktion in JavaScript nicht dem Java-Konstruktor entspricht, sondern der eigentlichen Objekterzeugung. Somit werden in dieser Funktion den deklarierten Attributen entweder Standardwerte wie 0 oder eine leere Zeichenkette, vordefinierte Werte wie „Name“ oder Werte von Übergabeparametern der Funktion zugewiesen. Diese Übergabeparameter entsprechen den im Java-Konstruktor definierten Parametern. Da Variablen bei der Deklaration in JavaScript, anders als in Java keine Standardwerte sondern lediglich `undefined` beinhalten, erzeugt JSweet automatisch eine adäquate Standardwertzuweisung. Anschließend wird dieser Wert unter Umständen durch einen anderen Wert überschrieben, wie an der Variable `balance` zu sehen. Alle Variablen werden zudem mit dem Präfix `this` angesprochen, da dieses Schlüsselwort in JavaScript-Funktionen das Objekt referenziert, an dem die Funktion aufgerufen wurde. So ist es möglich aus der Funktion heraus dynamisch Attribute zu erzeugen und zu manipulieren. Die Funktion `deposit(double)` wird nicht direkt dem Objekt hinzugefügt, sondern über den Aufruf `BankAccount.prototype.deposit` dessen Prototypen. Wie bereits in Abschnitt 5.1 erwähnt soll dadurch sichergestellt werden, dass die Methode wie bei Klassen in Java jederzeit aufgerufen werden kann. Wäre sie direkt am Objekt selbst deklariert, bestünde die Möglichkeit, dass sie während der Laufzeit entfernt worden wäre. In diesem Fall kann durch die Deklaration am Prototypen darauf zurückgegriffen werden. JSweet übersetzt öffentliche und private Methoden und Attribute gleichartig, da es für Zugriffsmodifikatoren kein Äquivalent in JavaScript gibt. Bei privaten Methoden kommentiert der Konverter das Schlüsselwort `private` aus, sodass für eine spätere Weiterentwicklung erkennbar bleibt, dass die Funktion als privat definiert wurde. Der Konverter kann auch reguläre und statische innere oder anonyme Klassen im Java-Code übersetzen.

Auch Klassen in Vererbungsbeziehungen kann JSweet übersetzen. Da das Konvertierungswerkzeug als intermediäre Sprache TypeScript verwendet, welches Vererbungsbeziehungen gleichartig zu Java definiert, wird anschließend der TypeScript-eigene Übersetzungsmechanismus genutzt. Das in ECMAScript 6 eingeführte Klassenkonzept wird nicht verwendet. Insgesamt muss der konzeptionelle Unterschied zwischen Vererbung auf Basis von Klassen und auf Basis von Prototypen

```java
public class BankAccount {

    public double balance = 0;
    public String name = "Name";

    public BankAccount(double initialBalance) {
    this.balance = initialBalance;
  }

    public double deposit(double credit) {
        balance += credit;
        return this.balance;
    }
}
```

Quelltext 5.2: Definition der Klasse BankAccount in Java

```javascript
var BankAccount = (function () {

        function BankAccount(initialBalance) {
            this.balance = 0;
            this.name = "Name";
            this.balance = initialBalance;
        }

        BankAccount.prototype.deposit = function (credit) {
            this.balance += credit;
            return this.balance;
        };

        return BankAccount;
    }());
```

Quelltext 5.3: Die von JSweet erzeugte Übersetzung der Klasse BankAccount

```
var __extends = this.__extends || function (subclass, superclass) {
    for (var propertyName in superclass) {
        if (superclass.hasOwnProperty(propertyName)) {
            subclass[propertyName] = superclass[propertyName];
        }
    }
    function subclassPrototype() { this.constructor = subclass; }
    subclassPrototype.prototype = superclass.prototype;
    var newPrototype = new subclassPrototype();
    subclass.prototype = newPrototype;
};
```

Quelltext 5.4: Nicht verkürzter Code der _extends-Funktion [Rim13]

in JavaScript überbrückt werden. Bei der Übersetzung von TypeScript zu JavaScript wird daher die zusätzliche Funktion `_extends` generiert, die jeder Datei, die ein erbendes Objekt enthält hinzugefügt wird. Der Code der Methode ist dabei allgemeingültig und ändert sich auf Basis des betrachteten Objekts nicht. Zudem wird für das Objekt eine bei der Erstellung aufgerufene, anonyme Funktion erzeugt, welche `_extends` aufruft und notwendige `super`-Aufrufe auf die Oberklasse tätigt. Ab Seite 101 im Anhang ist ein Beispiel für die Übersetzung einer Superklasse und deren Subklasse zu finden. Über der Definition der Subklasse wird die Funktion `_extends` deklariert, welche jedoch stark minimiert eingefügt wird. Quelltext 5.4 zeigt den Code der Funktion in nicht verkürzter Form.

Die Funktion `_extends` kopiert innerhalb der For-Schleife zunächst alle aufzählbaren Eigenschaften der Superklasse in das Objekt der Subklasse. So wird sichergestellt, dass diese wie bei Java-typischer Vererbung in der Subklasse verfügbar sind. Allerdings werden nicht die Eigenschaften des Prototyps der Superklasse kopiert. Daher müssen die Prototypen der betroffenen Objekte anschließend korrekt miteinander verknüpft werden, um eine Vererbungsbeziehung wie in TypeScript bzw. Java abzubilden. Der Prototyp der Superklasse wird dabei als Prototyp für die Prototypen der Subklasse festgelegt. Über diese Verkettung ist aus der Subklasse der Zugriff auf vererbte Eigenschaften aller Superklassen möglich. In der anonymen Funktion werden zudem Konstruktoren definiert, welche mögliche Aufrufe auf Konstruktoren der Superklasse enthalten können [Rim13]. Auch dieser Fall ist am Codebeispiel zu erkennen. Durch den vom Konverter generierten Code ist es anschließend möglich, Objekte mit `var book = new Book("Title", 1, "Author");` zu erzeugen, welche die Eigenschaften des Super-Objekts `Item` besitzen. Durch die Verknüpfung der Prototypen ergeben Prüfungen mit `instanceof` korrekterweise, dass ein Objekt des Typs `Book` ein `Item` ist, ein `Item` jedoch kein `Book`.

Abstrakte Klassen werden auf die gleiche Art und Weise übersetzt. Interfaces sind in JavaScript jedoch konzeptionell nicht vorhanden. JSweet erzeugt bei der Übersetzung eines Interfaces lediglich eine leere Datei. Das Objekt, welches das Interface implementiert enthält natürlich die dort implementierten Methoden, allerdings geht die strikte Bindung an die definierte Schnittstelle verloren. Jedoch wird in dem Objekt eine Eigenschaft mit dem Namen `_interfaces` erzeugt welche als Wert ein Array mit den Namen der ursprünglich in Java implementierten Interfaces enthält (z.B. `TestImplement["__interfaces"] = ["packagename.TestInterface"];`).

Enumerationen werden in JavaScript nicht durch besondere Codeelemente unterstützt. Stattdessen besteht, wie in Kapitel 5 beschrieben, die Möglichkeit das Verhalten von Java-typischen Enumerationen in JavaScript nachzubilden. Einem Objekt werden dazu Eigenschaften mit den Namen

```
1 var ratio = enums.ScreenRatio.RATIO_3_2;
2 var value = ScreenRatio["_$wrappers"][ratio].getValue();
```

Quelltext 5.5: Aufruf einer Methode einer komplexen Enumeration

der einzelnen in der Enumeration definierten Konstanten hinzugefügt. Diesen Eigenschaften wird zudem eine Ordinalzahl zugeordnet, die die Konstante zur Laufzeit numerisch identifiziert. JSweet kann auf diese Weise einfache Enumerationen übersetzen. Dennoch bleibt die Möglichkeit bestehen, zur Laufzeit des Programms weitere Konstanten hinzuzufügen oder bestehende zu verändern. Dies weicht aufgrund der dynamischen Typisierung von JavaScript vom Verhalten von Enumerationen in Java ab. Java erlaubt über einfache Enumerationen mit namentlich gekennzeichneten Konstanten hinaus auch komplexere Enumerationstypen mit Werten, die den Konstanten zugewiesen werden und Methoden, die die Funktionalität der Enumeration erweitern. Auch diese Aspekte werden bei der Übersetzung durch JSweet berücksichtigt. Sobald diese Elemente im Java-Code einer `Enum`-Deklaration auftreten, generiert JSweet eine zusätzliche Klasse, die als Wrapper für die zusätzlichen Funktionalitäten agiert. Auf Seite 102 im Anhang ist der Java-Code einer Enumeration und die daraus generierte Übersetzung zu sehen. Für die Enumeration mit dem Namen `ScreenRatio` heißt die Wrapper-Klasse `ScreenRatio_$WRAPPER` und deklariert in erster Linie einen dazugehörigen Konstruktor. Dieser nimmt als Übergabeparameter zusätzlich zu dem Namen der Konstante und der Ordinalziffer auch Werte an, die den Konstanten zugeordnet werden sollen. Des Weiteren werden an dieser Stelle die zusätzlichen Methoden übersetzt. Das Enumerationsobjekt `ScreenRatio` bekommt die Eigenschaft `_wrappers` zugewiesen, welche ein Array der Wrapper-Objekte enthält. Für jede Konstante wird mithilfe des zuvor definierten Konstruktors ein Wrapper-Objekt erstellt, welches die im Java-Code festgelegten Werte übergeben bekommt. Quelltext 5.5 zeigt, wie Methoden eines so übersetzten, komplexen Enumerationstypen aufgerufen werden können. JSweet übersetzt sowohl die Enumerationsklasse als auch die damit durchgeführten Aufrufe automatisch auf Basis der beschriebenen Methode.

JSweet unterstützt die Nutzung von überladenen Methoden in Java um optionale Parameter in JavaScript-Funktionen abzubilden. Dies deckt jedoch nicht alle möglichen Fälle von Überladung im Java-Code ab, weshalb für diese Fälle transformationsvorbereitende Refactorings benötigt werden. Das JSweet-Plugin für Eclipse weist auf die Notwendigkeit manueller Anpassungen in diesen Fällen mit einer Fehlermeldung hin.

Optionale Parameter in Methodensignaturen sind in Java nicht möglich, stattdessen wird die Methode mit verschiedenen Parameterlisten überladen. JSweet kann überladene Methoden in Java auf zwei verschiedene Arten zu JavaScript übersetzen. Dabei wird unterschieden, ob der Parameter, der optional definiert werden soll, über einen Standardwert verfügt oder nicht. In dem Fall, dass der optionale Parameter nicht angegeben wird, würde stattdessen der Standardwert verwendet. Im Methodenkörper der Methode ohne optionalen Parameter wird somit die Methode mit optionalen Parameter aufgerufen, wobei der definierte Standardwert übergeben wird. Quelltext 5.6 zeigt die Implementation dieses Falles in Java. Der von JSweet erzeugte JavaScript-Code definiert die Methode in diesem Fall einmalig und prüft im Methodenkörper ob der optionale Parameter einen Wert hat oder undefiniert ist. Im letzteren Fall wird der Standardwert zugeordnet. Im Endeffekt wird sowohl

```java
public class OverloadDefault {

  //double d is optional, has default value
  public String method(String s) {
    return method(s, 0d);
  }

  public String method(String s, double d) {
    return s + d;
  }
}
```

Quelltext 5.6: Überladen einer Methode mit optionalem Parameter mit Standardwert in Java

```javascript
OverloadDefault.prototype.method = function (s, d) {
  if (d === void 0) { d = 0.0; }
  return s + d;
};
```

Quelltext 5.7: Überladen einer Methode mit optionalem Parameter mit Standardwert in JavaScript

in Java als auch in JavaScript die Methode `method(s, d)` aufgerufen, wodurch es nicht möglich ist unterschiedliche Funktionalitäten je nach Parameter auszuführen.

Durch Überladen ist es in Java durchaus möglich, je nach übergebenem Parameter unterschiedlichen Code auszuführen. Wenn kein Standardwert für optionale Parameter definiert wird, ändert sich der von JSweet erzeugte JavaScript-Code. In diesem Fall werden im übersetzten JavaScript eigene Methoden mit den jeweils möglichen Konstellationen an Übergabeparametern definiert. Da diese jedoch nicht denselben Bezeichner tragen dürfen um sich nicht zu überschreiben, leitet der Konverter neue Methodennamen aus dem ursprünglichen Namen und den Typen der Übergabeparameter ab (z.B. `method$java_lang_String$double()`). In der Methodendefinition der ursprünglich überladenen Methode `method()` wird Code generiert, welcher prüft, welche Parameter übergeben wurden und anschließend an die zusätzlich erzeugten Methoden delegiert. Wenn keine passenden Methoden für die übergebenen Parameter vorhanden sind, wird eine Ausnahme geworfen, die auf eine ungültige Überladung hinweist. Dies ist besonders bei der nativen Weiterentwicklung in JavaScript hilfreich, da der Entwickler so Rückmeldung bei nicht vorgesehener Nutzung der Methode bekommt. Quelltext 5.8 zeigt in Java überladene Methoden mit unterschiedlichen Funktionalitäten. Diese werden zu zwei individuell benannten Methoden konvertiert und von einer dritten Methode mit dem ursprünglichen Namen aufgerufen (siehe Quelltext 5.9).

Wie oben beschrieben sorgen mehrfach auftretende Bezeichner innerhalb einer Klasse für Probleme. JSweet kann einen Teil dieser Fälle jedoch mit automatischen Refactorings umgehen. Falls in einer Java-Klasse sowohl ein Attribut als auch eine Methode (z. B. der Getter des Attributs) denselben Bezeichner `a` tragen, benennt das Konvertierungswerkzeug das Attribut zu `_a` um. Somit wird verhindert, dass die Methodendeklaration in JavaScript das Attribut überschreibt. Der Bezeichner wird in der gesamten Klasse automatisch ersetzt. Diese Art des Refactorings gelingt jedoch nur automatisch, wenn die Deklarationen in derselben Klasse erfolgen. Auf Java-Seite ist es jedoch durchaus möglich, dass Attribute oder Methoden von einer Oberklasse vererbt werden. JSweet unterstützt keine automatische Umbenennung von Bezeichnern in anderen Klassen. Wenn eine Unterklasse also eine Methode mit dem gleichen Namen wie ein Attribut aus der Oberklasse definiert, kann dies nicht

```java
public class OverloadNoDefault {

  //double d is optional, has no default value
  public String method(String s) {
    return s;
  }

  public String method(String s, double d) {
    return s + d;
  }
}
```

Quelltext 5.8: Überladen einer Methode mit unterschiedlichen Funktionalitäten in Java

```javascript
OverloadNoDefault.prototype.method$java_lang_String = function (s) {
  return s;
};

OverloadNoDefault.prototype.method$java_lang_String$double = function (s, d) {
  return s + d;
};

OverloadNoDefault.prototype.method = function (s, d) {
  if (((typeof s === 'string') || s === null) && ((typeof d === 'number') || d === null))
    {
    return this.method$java_lang_String$double(s, d);
  }
  else if (((typeof s === 'string') || s === null) && d === undefined) {
    return this.method$java_lang_String(s);
  }
  else
    throw new Error('invalid overload');
};
```

Quelltext 5.9: Überladen einer Methode mit unterschiedlichen Funktionalitäten in JavaScript

ohne manuelle Anpassung gelöst werden. Allerdings weist JSweet mit der Fehlermeldung „method 'a' has the same name as a field in 'packagename.Superclass'" auf den Konflikt hin. Als Abhilfe können entweder unterschiedliche Bezeichner gewählt werden oder die Methode zusätzlich auch in der Oberklasse definiert werden. Letzteres sorgt dafür, dass das oben beschriebene automatische Refactoring ausgelöst wird. Allerdings ist die lediglich für die automatische Portierung eingefügte Deklaration einer Methode und somit Änderung des Programmstruktur eher ungeeignet, weshalb die Wahl unterschiedlicher Bezeichner vorzuziehen ist.

Wie im vorherigen Kapitel beschrieben sind alle in Java vorhandenen Kontrollstrukturen gleichartig auch in JavaScript zu finden. Demnach kann JSweet diese Strukturen automatisch übersetzen. Quelltext 5 im Anhang beinhaltet Beispiele für die verschiedenen Formen von Kontrollstrukturen. Für die For-Schleife ist sowohl die reguläre Zählschleife als auch die syntaktisch unterschiedliche For-Each-Schleife definiert. Beide Varianten können durch den Konverter übersetzt werden. Quelltext 6 zeigt den für bessere Lesbarkeit etwas umformatierten Output von JSweet. Es ist zu erkennen, dass die Kontrollstrukturen ohne weitreichende Veränderungen übertragen werden können. Die Verständlichkeit des Codes und des Kontrollflusses wird aufgrund der sehr ähnlichen Syntax nicht negativ beeinflusst. Anzumerken ist, dass der Konverter bei Vergleichen wie in der Bedingung der If-Verzweigung automatisch zum strikten Gleich mit === transformiert. Die For-Each-Schleife wird zu einer JavaScript-typischen Zählschleife transformiert, wobei ein Hilfsindex zum Einsatz kommt. Dieser iteriert über das zu durchlaufende Array und weist den Wert für den jeweiligen Index einer Variable zu. Der Name dieser Variable entspricht dem des in Java verwendeten Bezeichners für das betrachtete Array-Element. JSweet verwendet nicht die seit ECMAScript 5 vorhandene Methode des Array-Prototypen `forEach()`, auch wenn ECMAScript 6 als Zielversion konfiguriert ist.

Primitive Datentypen werden von JSweet wie in Tabelle 5.1 zu erkennen zu ihren jeweiligen Entsprechungen übersetzt. Dabei werden alle numerischen Datentypen bis auf `char` zu `Number` in JavaScript konvertiert. Dies bedeutet, dass datentypspezifische Limitationen wie der Zahlenraum bei der Übersetzung verloren gehen. Demnach ist es nach der Übersetzung auch möglich, Nachkommastellen zu einer in Java rein ganzzahligen Variable hinzuzufügen. Laut JSweet-Spezifikation ist „Präzision in JSweet-Anwendungen in der Regel nicht relevant" (vgl. [Paw16b]). Dennoch muss insbesondere bei rechenintensiven Anwendungen geprüft werden, ob Codelogik portiert wird, bei welcher derartige Zahleneigenschaften eine integrale Rolle spielen. Um die ursprünglichen Zahlenräume einigermaßen einzuhalten übersetzt der Konverter Casts eines numerischen Datentyps zu einem Anderen mit einer Rundung des Wertes auf eine Ganzzahl der erlaubten Größe. Die gleiche Art der Konversion gilt auch für die in Java vorhandenen Wrapper-Klassen für die primitiven numerischen Datentypen. `Integer`, `Double`, etc. werden gleichförmig zu `Number`, wobei `Number` über einige der Methoden und Attribute dieser Klassen verfügt. Allerdings beziehen sich diese jedoch auf JavaScript-Zahlen, ohne den ursprünglichen Kontext des Datentyps in Java mit einzubeziehen. Da der numerische Datentyp `char` Unicode-Symbole ausdrückt wird dieser nicht zu `Number` sondern zu einem entsprechenden `String` der Länge 1 übersetzt. Auch der Referenztyp `String` von Java wird zu `String` in JavaScript transpiliert. Die statischen Methoden von `String` wie z.B. `substring()` werden auf Entsprechungen im JavaScript-`String`-Objekt gemappt. Beide Sprachen

```
throw Object.defineProperty(
  new Error("Message"), '__classes',
    {
      configurable: true,
      value: ['java.lang.Throwable', 'java.io.IOException', 'java.lang.Object', 'java.io.
     FileNotFoundException', 'java.lang.Exception']
    }
 );
```

Quelltext 5.10: Werfen einer Ausnahme in konvertiertem Code

verfügen über übereinstimmende Boolean-Typen, wodurch `boolean` direkt zu `Boolean` in JavaScript übersetzt werden kann. Boolesche Operationen bleiben dabei funktionsgleich erhalten.

Über die primitiven Datentypen hinausgehende Datentypen und Klassen des JDK können teilweise von JSweet übersetzt werden. Der Konverter nutzt dazu Makros, welche unterstützte Aufrufe in Java durch vordefinierten JavaScript-Code ersetzen, der die Funktionalität emuliert. Eine Auflistung unterstützter Klassen und Methoden ist in der Sprachspezifikation von JSweet zu finden. Die mitgelieferten Makros umfassen beispielsweise:

- **Object.clone()**: Das Klonen von Objekten mithilfe der `clone()`-Methode ist möglich, wenn die direkte Oberklasse des zu klonenden Objekts in Java `Object` ist. In diesem Fall generiert JSweet Code, um eine neue Instanz des zu klonenden Objektes zu erstellen und alle aufzählbaren Eigenschaften zu kopieren. Bei einer längeren Vererbungskette muss der `clone()`-Aufruf in jeder Oberklasse bis zu `Object` weitergereicht werden um die automatische Übersetzung nutzen zu können.

- **Calendar**: Objekte des Typs `Calendar` werden bei der Übersetzung zum JavaScript-Objekt `Date` übertragen. Methoden, die funktionsgleich in `Date` existieren, werden automatisch aufeinander gemappt. So wird zum Beispiel `getTimeInMillis()` zu `getTime()` übersetzt. Aufgrund des geringeren Funktionsumfangs von `Date` können viele Aufrufe auf `Calendar` jedoch nicht automatisch übertragen werden.

JSweet kann die grundlegende Struktur von Ausnahmebehandlungs-Code übersetzen und bildet die dabei verwendeten Ausnahmetypen auf Basis von Java-Exceptions nach. Das bedeutet, dass JSweet nicht versucht ein Mapping von Exception-Typen auf Error-Typen in JavaScript durchzuführen. Stattdessen werden in Java-Code geworfene Ausnahmen zu `Error`-Objekten übersetzt, welche zusätzliche Eigenschaften basierend auf der ursprünglichen Java-Exception zugewiesen bekommen. Für den Aufruf `throw new FileNotFoundException("Message")` ergibt sich der in Quelltext 5.10 gezeigte Code.

Hierbei wird ein Objekt des grundlegenden Fehlertyps `Error` erzeugt, welchem eine Eigenschaft mit dem Namen `_classes` zugewiesen wird. Diese Eigenschaft besteht wiederum aus einem Objekt mit einer Liste der in Java verwendeten Exception-Klasse und deren Oberklassen. Über diese Information kann innerhalb eines Catch-Blocks ermittelt werden, um welche Ausnahme es sich im ursprünglichen Java-Code handelt. So kann JSweet eine Struktur aus If-Else-Verzweigungen im Catch-Block generieren, welche dafür sorgt, dass wie in Java der richtige Code für einen bestimmten Ausnahmetyp ausgeführt wird. Diese Art der Übersetzung tritt bei Java-eigenen Ausnahmen auf, deren Code zur Java-API gehört und somit nicht in der Codebasis des Portierungsprojekts

liegt. Bei benutzerdefinierten Ausnahmeklassen wird stattdessen ein neues Objekt diesen Typs geworfen. Die Syntax von Try-Catch- sowie von Finally-Blöcken wird gleichbleibend übersetzt. Die größten Änderungen bei der Übersetzung treten im Catch-Block auf, da mehrere Catch-Blöcke für verschiedene Ausnahmetypen in Java durch einen Block mit Verzweigungen ersetzt werden. Strukturell bleibt der Code sehr ähnlich, allerdings ist die Lesbarkeit des Codes aufgrund der komplexeren Bestimmung des Ausnahmetyps geringer. Die `throws`-Angabe an Methoden wird in JavaScript nicht unterstützt und bleibt nach der Übersetzung auch nicht als Kommentar erhalten. Besonders anzumerken ist, dass der Übersetzer nicht die nativen Fehlertypen in JavaScript benutzt, sondern eine Hierarchie aus Fehlern auf Basis von Java-Ausnahmen aufbaut. Dadurch bleiben im Generat viele Java-spezifische Bezeichner erhalten. Beispielcode für die Übersetzung der Fehlerbehandlung durch JSweet ist ab Seite 106 im Anhang zu finden.

[Paw16b]

### 5.2.2. Transformationsvorbereitende Refactorings

Obwohl das Konvertierungswerkzeug JSweet viele in Java verwendete Codeelemente ohne manuellen Eingriff durch einen portierenden Entwickler übersetzen kann, ergeben sich aufgrund der Sprachunterschiede und Werkzeuglimitationen dennoch Bedarfe für transformationsvorbereitende Refactorings. Diese werden sowohl in diesem Kapitel als auch in der in Kapitel 6 beschriebenen Durchführung der Portierungsmethode erläutert.

Obwohl die automatische Konvertierung von überladenen Methoden mit JSweet grundsätzlich möglich ist, ist eine fehlerfreie Übersetzung nicht in allen Fällen möglich. Auf diese nicht unterstützen Fälle weist JSweet mit einer Fehlermeldung hin, woraufhin ein manuelles Refactoring durchgeführt werden muss. Grundlegend kann das Problem behoben werden, indem den einzelnen Überladungen bereits im Java-Projekt unterschiedliche Methodennamen zugewiesen werden. Somit wird die Überladung vor der Konversion aufgelöst. Statt `print(String)` und die Überladung `print(PrintableObject)` würden zwei verschiedenen Methoden `printString(String)` und `printPrintableObject(PrintableObject)` definiert. Die explizite Nennung des Übergabeparameters im Methodennamen ist jedoch oftmals unpraktisch oder nicht erwünscht.

Eine weitere Möglichkeit eine Überladung aufzulösen, ist die Einführung eines Objekts zur Übergabe der Parameter. Statt die unterschiedlichen Parameter direkt zu übergeben, werden sie in ein zusätzliches Objekt gekapselt, welche anschließend der Methode übergeben wird. Innerhalb der Methode muss dann geprüft werden, welche Typen und Werte im Übergabeobjekt vorhanden sind. Anhand dieser Information können innerhalb der Methode unterschiedliche Ausführungspfade gewählt werden. Statt mehrfacher Überladung wird die Methode nur einmal mit dem neuen Objekt als Übergabeparameter definiert.

Wie bereits erwähnt werden die verschiedenen numerischen Datentypen in Java allesamt mit `Number` übersetzt. Dies gilt auch für die Wrapper-Klassen, die Hilfsmethoden für die verschiedenen Datentypen liefern. Ein Beispiel dafür ist das Attribut `MAX_VALUE`, welches die höchste Zahl eines numerischen Typs zurückgibt. Dieser Aufruf wird unabhängig vom Datentyp in Java zu `Number.MAX_VALUE` übersetzt, was das Maximum einer JavaScript-Zahl liefert. Dieser Wert stimmt nur mit dem maximalen Wert eines `double` überein, nicht jedoch aber mit denen der anderen numerischen Typen. In Anwendungsfällen in denen die spezifischen Attribute eines Zahlentyp relevant sind,

muss daher manuell eingegriffen werden, um das gewünschte Verhalten nachzubilden. Für das Beispiel des Maximalwertes, wäre es demnach möglich eine Hilfsklasse zu schreiben, welche für `MAX_VALUE`-Aufrufe den korrekten Maximalwert für den Anwendungskontext liefert. Dazu müssen die Aufrufe im Code auf Entsprechungen in der Hilfsklasse abgeändert werden. In Fällen, in denen diese Art von Präzision gefragt ist, muss natürlich auch darauf geachtet werden, dass das Verhalten bei Überlauf des Wertebereichs abgebildet wird. Nach Übertreten des Maximums wird ab dem Minimum weiter gezählt. Diese Schwelle und das Minimum müssen mit einer If-Abfrage und einem Aufruf des Minimalwertes z.B. aus der Hilfsklasse geprüft werden. Abgesehen von solchen Fällen muss abhängig vom Anwendungskontext evaluiert werden ob ein Mapping auf Methoden von JavaScript-`Number` semantisch gleichbleibenden Code ermöglicht. JSweet bietet auch eine Adapter-Klasse für `Number` an, über welche die exakten Methoden und Eigenschaften einer Zahl in JavaScript abgerufen werden können. Mithilfe dieses Adapters kann alternativ auch versucht werden, die gesamte Logik bereits auf Java-Seite auf die Gegebenheiten in JavaScript anzupassen. Dieses Vorgehen ist jedoch je nach Portierungsansatz nicht anwendbar, da die Ausführbarkeit der Java-Version stark eingeschränkt wird.

Über `Number` hinaus liefert die von JSweet angebotene Konverter-API reichhaltige Möglichkeiten transformationsvorbereitende Refactorings durchzuführen, um die Portierbarkeit von Code zu verbessern. Dabei dient die API dazu, bereits in Java-Code mit JavaScript-Objekten und Konzepten zu arbeiten. Für diesen Zweck enthält die JSweet-Core-Bibliothek Adapter für gängige JavaScript-Objekte wie zum Beispiel `Object`, `Array`, `String`, `Math`, `Date` oder `Error`. Die Java-Adapter für diese Objekte enthalten die in JavaScript zur Verfügung stehenden Methoden. Wenn sie in Java-Code verwendet werden, können sie bei der Übersetzung mit JSweet direkt auf die entsprechenden Objekte abgebildet werden. Vor der Übersetzung ist daher zu überlegen, ob nativer Java-Code alternativ auch mit Aufrufen dieser API realisiert werden kann. So wäre die Übersetzbarkeit gewährleistet und zudem könnten problematische Aufrufe von JDK-Klassen vermieden werden. Natürlich ist zu berücksichtigen, dass Refactorings mittels der Konverter-API Auswirkungen auf die Ausführbarkeit der Originalversion haben. Die Adapter-Klassen verfügen in Java nicht über ihre eigentliche Funktionalität in JavaScript, sondern agieren lediglich als Übersetzungsreferenzen für das Übersetzungswerkzeug. Daher muss bei der Portierung entschieden werden, ob Refactorings mithilfe der angebotenen Konverter-API vorgenommen werden sollen. Eine detailliertere Argumentation dazu folgt in Abschnitt 6.1. Grundsätzlich übersetzt JSweet Aufrufe bei äquivalenten Objekten automatisch, allerdings werden oftmals nicht alle Methoden vollständig abgedeckt. In diesen Fällen treten Probleme bei der automatischen Übersetzung auf. Um diese Probleme zu umgehen, kann von vorne hinein mithilfe der JSweet-Adapter die JavaScript-API der betroffenen Objekte genutzt werden.

Beispielhaft kann dieses Vorgehen an dem Java-Aufruf `System.currentTimeMillis()` aufgezeigt werden, welcher die zum Zeitpunkt des Aufrufs vergangene Zeit seit dem 1. Januar 1970 in Millisekunden als `long` zurückgibt. Dieser Aufruf kann von JSweet nicht automatisch übersetzt werden und wird bei Nutzung des Eclipse-Plugins als Problemstelle identifiziert. Quelltext 5.11 zeigt, wie der Typ `Date` der JSweet-API genutzt werden kann, um den gewünschten Aufruf nachzubilden. Das Erzeugen eines `Date`-Objektes über den parameterlosen Konstruktor weist dem Objekt das

```
1  //Urspruenglicher Aufruf
2  long milisTime = System.currentTimeMillis();
3
4  //Refactoring mittels JSweet-API
5  long millisTime = SystemAdapter.currentTimeMillis();
6
7  public class SystemAdapter {
8    public static long currentTimeMillis(){
9      jsweet.lang.Date date = new jsweet.lang.Date();
10     return (long) date.getMilliseconds();
11   }
12 }
```

Quelltext 5.11: Verwendung der JSweet-API zum Refactoring von Problemstellen

aktuelle Datum des Systems zu. Anschließend kann über die Methode `getMilliseconds()` dieselbe Differenz in Millisekunden abgerufen werden, wie mit `currentTimeMillis()` in Java.

Ähnlich funktioniert das Refactoring von Abhängigkeiten zu nicht unterstützten Plattform- oder Bibliotheksaufrufen mit JSweet-Candies. Candies definieren weitere Adapter für JavaScript-APIs und externe JavaScript-Bibliotheken. Sie können über das Build-Management-Tool Maven in JSweet-Projekte integriert werden. Weitere Ausführungen zur Nutzung von JSweet-Candies folgen anhand eines Beispiels in Unterabschnitt 6.3.3.

Die Konverter-API erlaubt auch das Entfernen oder Ersetzen von Methoden bei der Übersetzung. Dazu können Methoden in Java mit den Annotationen `@Erased` oder `@Replace` annotiert werden. Zu entfernende Methoden verbleiben funktional in der Java-Version, werden allerdings nicht vom Konverter übersetzt. Demnach fehlt die entfernte Methode im JavaScript-Generat. Dies kann dazu verwendet werden um Code, der nicht für die JavaScript-Version relevant ist, zu entfernen. Fehlermeldungen des Konverters und des dazugehörigen Eclipse-Plugins werden in diesen Methodenkörpern unterdrückt. Somit können für die Übersetzung problematische Codeelemente in Methoden extrahiert und entfernt werden, ohne die Ausführbarkeit des Codes in Java zu beeinträchtigen. Oftmals muss die entfernte Funktionalität jedoch auch in JavaScript implementiert werden. Daher kann die Nutzung der `@Replace`-Annotation sinnvoll sein. Diese erlaubt die Angabe von TypeScript-Code im Annotationskörper. Bei der Übersetzung wird der Java-Code der annotierten Methode ignoriert und durch den zu JavaScript transpilierten TypeScript-Code ersetzt. Alternativ kann durch die Angabe einer leeren Replace-Annotation lediglich die Funktionssignatur oder eine Fehlermeldung zur späteren Implementation im Generat übersetzt werden.

# 6. Durchführung der Portierung und Bewertung der angewendeten Methode

Dieses Kapitel beschreibt die Portierung von Code des Simulationsframeworks DESMO-J von der Ausgangssprache Java zu JavaScript. Dabei orientiert sich das Vorgehen und die Einteilung des Kapitels an den Schritten der in Abschnitt 3.2 beschriebenen Portierungsmethode. Zuerst werden die Plattformabhängigkeiten von DESMO-J innerhalb einer Abhängigkeitsanalyse identifiziert. Anschließend wird eine Teilfunktion abgespalten und deren plattformunabhängiger Kern isoliert. Dieser kann dann mithilfe der in Kapitel 4 beschriebenen Werkzeugunterstützung konvertiert werden. Des Weiteren wird in diesem Kapitel der manuelle Aufwand vor und nach der automatischen Übersetzung analysiert.

## 6.1. Wahl der Strategie zum Propagieren von Änderungen am Ursprungscode

Da es sich bei DESMO-J um ein weiterhin aktiv entwickeltes Framework handelt, muss davon ausgegangen werden, dass in Zukunft bestehender Code verändert oder neuer Code hinzugefügt wird. Zudem soll die Software nicht migriert werden, weshalb die Originalversion in Java weiterhin aktiv verwendet und entwickelt wird. Durch die Portierung zu JavaScript wird eine Erweiterung der unterstützten Plattformen angestrebt und nicht ein Wechsel der Plattform. Aus diesem Grund soll eine im Anschluss an die Portierung mögliche parallele Entwicklung und Wartung der Versionen möglich sein. Dieses Konzept wird von der Portierungsmethode nach Stehle et al. explizit unterstützt. Eine fortwährende Nutzung der Java-Version bedeutet jedoch auch, dass deren Codebasis im Laufe der Portierung möglichst wenig verändert werden sollte. Solche Änderungen treten beispielsweise bei der Umsetzung von transformationsvorbereitenden Refactorings auf. Dabei wird der Code in Java so verändert, dass er durch die automatischen Konvertierungswerkzeuge besser und korrekter übersetzt werden kann. Je nach verwendetem Werkzeug müssen unter anderem nicht unterstützte Sprachkonstrukte vermieden oder in JavaScript nicht verfügbare Funktionen wie Multithreading entfernt oder andersartig implementiert werden. Dementsprechend verursacht ein Portierungsvorhaben in den meisten Fällen eine Anzahl portierungsspezifischer Anpassungen. Diese portierungsspezifischen Anpassungen erleichtern bzw. ermöglichen oftmals erst die Portierung der Codebasis mit Übersetzungswerkzeugen. Zumindest in dem in dieser Arbeit durchgeführten Werkzeugvergleich konnte kein Tool identifiziert werden, welches komplexeren Java-Code ohne derartige Anpassungen übersetzen konnte. Besonders die von vielen Werkzeugen angewandte Methodik, für die Übersetzung problematische Sprachkonstrukte, Abhängigkeiten und Typen durch

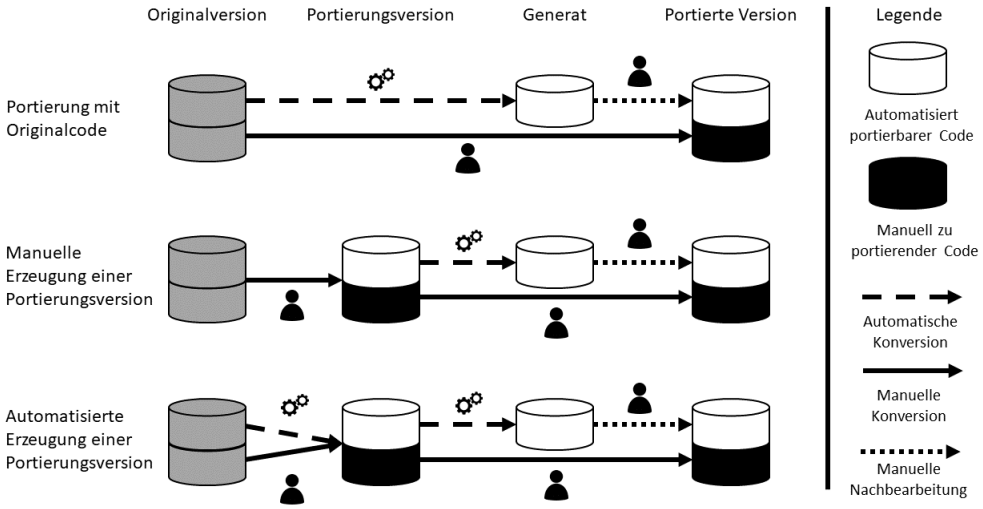


Abbildung 6.1.: Vergleich der möglichen Portierungsstrategien

eigene Implementationen innerhalb einer Hilfsbibliothek oder Konverter-API zu ersetzen, basiert auf der Durchführung portierungsspezifischer Anpassungen.

Derartige Anpassungen sind allerdings für die fortzuführende Originalversion in Java nicht sinnvoll. Je nach Art der Anpassungen können diese sogar dazu führen, dass die Originalversion nicht mehr ohne weiteres ausführbar ist. Dies betrifft vor allem Änderungen in Java, welche Abhängigkeiten durch andere, mit dem Konverter kompatible Abhängigkeiten ersetzen. Im Falle des für diese Portierung ausgewählten Konverters JSweet kann das die Ersetzung einer Java-Bibliothek durch eine JavaScript-Bibliothek sein. Probleme mit diesen Abhängigkeiten können somit vor der eigentlichen Portierung umgangen werden, indem der Bibliotheksaufruf bereits in Java definiert wird, allerdings ist die reine Java-Version durch diese Anpassung nicht mehr funktional. Ähnlich sieht es bei der Verwendung von Adaptern aus, die eingefügt werden, um Abhängigkeiten aus dem plattformunabhängigen Kern zu entfernen. Für die automatisierte Portierung ist diese Abstraktionsebene zwischen Aufrufen durchaus sinnvoll, jedoch wird die reine Java-Version durch eigentlich unnötigen Code angereichert und verkompliziert. Aus diesem Grund muss vor dem Beginn des Portierungsprozesses festgelegt werden, wie mit portierungsspezifischen Anpassungen umgegangen werden soll.

Dazu sollten zunächst die Entscheidungsmöglichkeiten und deren Folgen für die anschließende parallele Evolution der Software betrachtet werden. Die im Folgenden beschriebenen Portierungsstrategien sind in Abbildung 6.1 grafisch dargestellt. Idealerweise sollte die Java-Version der Software in einem unangetasteten Zustand verbleiben. Dies ist vor allem auch aufgrund der Abwärtskompatibilität erstrebenswert, da bestehende Projekte, welche auf dem Framework DESMO-J beruhen auch weiterhin lauffähig bleiben sollen. Davon sind in DESMO-J implementierte Simulationen sowie die Integration des Frameworks in andere Software wie Modellierungstools betroffen. Diese Projekte sollen nicht durch portierungsspezifische Anpassungen der Originalversion negativ beeinflusst werden. Naheliegend ist demnach die Vermeidung jeglicher Anpassungen im Java-Code, wodurch Rückwirkungen auf bestehende Nutzer des Frameworks vermieden werden. Allerdings ist dabei zu

bedenken, dass der Anteil an automatisiert portierbarem Code dadurch deutlich sinken kann, da eine Vorbereitung der Codebasis bezüglich der Anforderungen des Werkzeugs nicht möglich ist. Anstatt problematische Codeelemente und Sprachkonstrukte durch kompatible Alternativen zu ersetzen, müssen diese im Anschluss manuell in JavaScript implementiert werden. Das Ausmaß des ansteigenden manuellen Aufwands hängt dabei stark vom Funktionsumfang des Konverters ab. Je mehr Codeelemente im Originalzustand verarbeitet werden können, desto geringer der anschließende Implementierungsaufwand. Durch die teils deutlichen Sprachunterschiede zwischen Java und JavaScript ist eine hinreichend vollständige Portierung von Code ohne portierungsspezifische Anpassungen jedoch unwahrscheinlich. Besonders bei dieser Option muss evaluiert werden, ob der anfallende Portierungsaufwand noch unterhalb des Aufwands einer Reimplementation in JavaScript bleibt.

Eine weitere Option wäre die Erstellung einer Parallelversion zur Originalversion, welche die portierungsspezifischen Anpassungen implementiert. Diese Version dient dann als Basis für die automatische Portierung, wohingegen die Originalversion unangetastet bleibt. Dabei fällt einmalig Aufwand bei der Implementierung der Anpassungen für den Konverter an. Durch die Möglichkeit transformationsvorbereitende Refactorings durchzuführen, steigt jedoch in der Regel auch die Qualität und Menge des automatisch zu portierenden Codes. Dadurch sinkt der anfallende Aufwand bei der nachträglichen, manuellen Nachbearbeitung des Generats. Allerdings muss beachtet werden, dass sich bei der parallelen Evolution der Java- und JavaScript-Version nicht die Möglichkeit ergibt, bei Änderungen in der originalen Java-Version lediglich eine erneute, automatische Übersetzung durchzuführen. Änderungen müssten zuvor in die parallel zur Originalversion angelegten „Portierungsversion“ eingepflegt werden, was vor allem bei umfangreichen portierungsspezifischen Anpassungen komplex sein kann. Im schlechtesten Fall müsste eine gänzlich neue Portierungsversion entwickelt werden. Daher bietet sich für diese Option eine parallele Entwicklung und Wartung in beiden Versionen an. Das bedeutet, dass Änderungen der Java-Version gleichartig und nativ in der JavaScript-Version implementiert werden. Besonders in diesem Fall ist es relevant, dass das Konvertierungswerkzeug lesbaren und wartbaren JavaScript-Code erzeugt. Neben dem Codeverständnis müssen in diesem Fall natürlich nicht nur Entwickler mit Java-Wissen, sondern auch mit JavaScript-Wissen vorhanden sein. Auch die im Rahmen der Portierungsmethode erzeugte Traceability zwischen den Versionen und eine möglichst hohe Strukturgleichheit helfen bei der parallelen Evolution beider Versionen. Natürlich muss beachtet werden, dass Trace Links zwischen nach der Portierung implementierten Codeelementen erzeugt werden müssen, um die vollständige Abdeckung durch Traceability aufrechtzuerhalten und deren Vorteile weiterhin nutzen zu können. Dies kann entweder manuell während der Entwicklung oder mit Werkzeugunterstützung erreicht werden.

Das primäre Problem der zuvor beschriebenen Option ist der hohe Aufwand, der für eine erneute Portierung einer veränderten Originalversion anfällt. Insgesamt würde der Aufwand für die parallele Evolution geringer ausfallen, wenn nur die Java-Version angepasst werden müsste und die JavaScript-Version jedes mal daraus generiert werden würde. Als Ansatz dafür ist die automatisierte Erzeugung der Portierungsversion aus der Originalversion in Erwägung zu ziehen, um den dabei ansonsten anfallenden manuellen Aufwand zu minimieren. Konkret muss also Toolunterstützung gefunden oder entwickelt werden, welche den Quellcode analysiert und die für den zu verwendenden Konverter notwendigen transformationsvorbereitenden Refactorings automatisch vornimmt. Diese

Aufgaben sind stark auf bestimmte Konvertierungswerkzeuge bezogen und können in der Regel nicht allgemein implementiert werden, insbesondere wenn die vom Konverter angebotene API verwendet wird. Für JSweet sind solche Werkzeuge nicht verfügbar, wodurch für diese Option ein hoher Entwicklungsaufwand anfallen würde. Natürlich kann sich die Entwicklung eines solchen Präprozessors für zu konvertierenden Code bei großen Codebasen auch schon bei der einmaligen Erzeugung einer Portierungsversion lohnen.

Für die im Folgenden beschriebene Portierung wurde die zweite Option, also die einmalige Erzeugung einer Portierungsversion und anschließende parallele Evolution beider Versionen gewählt. Dies liegt vor allem an der mangelnden Werkzeugunterstützung für die automatische Generierung der Portierungsversion. Zudem würde die umfangreiche Entwicklung derartiger Werkzeuge über den Umfang und Fokus dieser Arbeit hinausgehen. Dennoch wäre eine weitere Untersuchung dieser Option in zukünftigen Arbeiten denkbar. Insbesondere wäre festzustellen, ob der auftretende Implementierungsaufwand für derartige Toolunterstützung gegenüber dem durch die Automatisierung eingesparten Aufwand vorteilhaft ist [Ter01]. Die ausgewählte Option bietet den Vorteil, dass sie keine negativen Auswirkungen auf bestehende Anwendungsfälle von DESMO-J hat und die Möglichkeiten des Konverters JSweet optimal ausnutzen kann. Die erreichte Strukturgleichheit und Lesbarkeit des Generats von JSweet eignet sich zudem für eine manuelle Nachbearbeitung sowie Weiterentwicklung und Wartung in JavaScript.

## 6.2. Abhängigkeitsanalyse

Die Portierungsmethode nach Stehle et al. beginnt mit der ersten Phase, der Identifizierung und Isolation von Plattformabhängigkeiten. Zur Identifizierung von Abhängigkeiten innerhalb des zu portierenden Projekts wird eine Abhängigkeitsanalyse durchgeführt. Dabei müssen die folgenden Arten von Abhängigkeiten analysiert werden:

- Abhängigkeiten zur API des Betriebssystems der Plattform
- Abhängigkeiten zur API der in der Originalversion verwendeten Programmiersprache
- Abhängigkeiten zu externen Code-Bibliotheken
- Abhängigkeiten zu von der Portierung ausgeschlossenen Teilen der Software

Die Ergebnisse der Abhängigkeitsanalyse dienen der Bestimmung des plattformunabhängigen Kerns, welcher automatisiert durch Konvertierungstools übertragen wird. In den meisten Fällen können Abhängigkeiten nicht ohne weiteres übersetzt werden und müssen daher entfernt oder abgeändert werden. Durch die Abhängigkeitsanalyse erlangt der portierende Entwickler Kenntnis über die Art und Anzahl der auftretenden Abhängigkeiten. Zudem wird ermittelt, an welchen Stellen innerhalb des Codes Abhängigkeiten auftreten, wodurch eine für die folgenden Schritte adäquate Isolationsstrategie entwickelt werden kann.

Im Folgenden wird dargestellt, wie die Abhängigkeitsanalyse des Simulationsframeworks DESMO-J durchgeführt wurde und welche Ergebnisse dabei herausgekommen sind. Bezogen auf die Abhängigkeiten zu von der Portierung ausgeschlossenen Teilen der Software wird festgelegt, welche

Softwareteile in dem hier betrachteten Portierungsprozess vorerst betrachtet werden und welche nicht.

Wie bereits in Abschnitt 4.4 erwähnt, wurden für die Durchführung der Abhängigkeitsanalyse die in die Entwicklungsumgebung IntelliJ IDEA integrierten Analysetools verwendet. Dazu wird die Codebasis von DESMO-J im Originalzustand in einem Projekt geladen. Die analysierte Version der Software ist die zum jetzigen Zeitpunkt aktuelle Version 2.5.1e (erschienen März 2017). In IntelliJ kann über das Menü „Analyze“ mit dem Menüpunkt „Analyze Dependencies“ der Dependency Viewer für das geöffnete Projekt aufgerufen werden. Dieser schlüsselt die im Projekt auftretenden Abhängigkeiten in den Kategorien Bibliotheks-, Produktions- und Testklassen auf. Der Nutzer kann so die gesamten Abhängigkeiten des Projektes betrachten. Alternativ können Filter gesetzt werden, die es ermöglichen Abhängigkeiten für Pakete oder einzelne Klassen anzuzeigen. Zudem lässt sich Anzeigen, in welchem Anwendungskontext die Abhängigkeit auftritt. Mögliche Anwendungskontexte wären unter anderen die Deklaration einer lokalen Variable, die Nutzung als Rückgabewert oder als Oberklasse/Interface. So kann sehr detailliert erkannt werden, wie die auftretenden Abhängigkeiten in der Software genutzt werden. Auf Wunsch kann der Nutzer aus dem Dependency Viewer direkt in den Code springen, wo die Abhängigkeit auftritt.

Grundsätzlich strukturiert sich die Codebasis von DESMO-J in zwei Pakete: `core` und `extensions`. `core` enthält die notwendigen Klassen und Funktionen für die Implementation einer Simulation und der Erzeugung eines Simulationsberichts als HTML-Ausgabe. Die Extensions, also Erweiterungen dieser Grundfunktionalität umfassen Visualisierungen der Simulationsergebnisse, eine grafische Oberfläche für die Ausführung von Simulationen und mitgelieferte Implementation von Elementen der Anwendungsdomänen Hafen und Produktion. Die Erweiterungen bauen auf dem Kern des Frameworks auf und werden selbst nicht im Paket `core` genutzt. Aus diesem Grund wird das Paket `extension` von der hier durchgeführten Portierung ausgeschlossen. Die Erweiterungen können im Nachhinein jedoch auch mittels des beschriebenen Portierungsprozesses übersetzt werden. Für die exemplarische Darstellung der Portierung sind sie jedoch nicht relevant, da eine grundlegende Ausführung einer Simulation auch lediglich mit dem Framework-Kern möglich ist.

Wird die Abhängigkeitsanalyse nun für das Paket `core` durchgeführt, lassen sich die darin auftretenden Abhängigkeiten zur Sprach-API von Java und zu externen Bibliotheken erkennen. Direkte Abhängigkeiten zum Betriebssystem sind nicht zu erkennen, da auch diese über Aufrufe der Sprach-API realisiert werden. Dies liegt an dem in Java umgesetzten Konzept der Plattformunabhängigkeit, wodurch Java-Entwickler immer die Sprach-API verwenden und diese Aufrufe falls nötig intern an das Betriebssystem weitergeleitet werden. Das JDK bietet für solche Aufrufe entsprechende Schnittstellen [Ull15]. Auch treten keine Abhängigkeiten zu von der Portierung ausgeschlossenen Softwareteilen auf, da das Paket `core` keine der Extensions voraussetzt. Die Abhängigkeiten von `extensions` auf `core` sind unidirektional.

Das `core`-Paket von DESMO-J hat Abhängigkeiten zu sieben Oberpaketen des JDK, also der Sprach-API von Java. Die Tabelle 6.1 stellt dar, wie viele Abhängigkeiten jeweils zu diesen JDK-Paketen vorhanden sind. Dabei wird sowohl die Gesamtzahl der Abhängigkeiten in `core` angegeben, als auch die Anzahl der Abhängigkeiten zu den einzelnen Unterpaketen von `core`. So kann ermittelt werden, welche Pakete und somit grob aufgeschlüsselt welche Funktionalitäten welche Arten von Abhängigkeiten besitzen. In der Gesamtsumme der Abhängigkeiten für `core` werden Abhängigkeiten zu einer bestimmte Klasse aus verschiedenen DESMO-J-Paketen nur einmalig gezählt. Grundlegend

ist zu erkennen, dass fast alle Pakete von DESMO-J Abhängigkeiten zu `java.lang` und `java.util` haben, was nicht weiter verwunderlich ist, da diese Pakete grundlegende Elemente wie einfache Datentypen und Dateistrukturen enthalten. Das Paket `java.lang` wird bei allen Java-Projekten automatisch eingebunden. DESMO-J macht intensiven Gebrauch von diesen grundlegenden Java-Klassen. In der tabellarischen Übersicht besonders hervorgehoben sind die Unterpakete `lang.reflect`, welches Klassen für Reflection enthält und `util.concurrent`, welches Funktionen für Nebenläufigkeit in Java enthält. Wie bei der Toolanalyse ermittelt wurde, bereiten sowohl Reflection als auch Nebenläufigkeit besondere Probleme bei der automatischen Übersetzung. Besonders die mangelnde Unterstützung von Nebenläufigkeit in JavaScript wird an diesen Stellen problematisch sein. Besonders das Unterpaket `simulator`, welches einen Großteil der ausführungsrelevanten Basisklassen enthält, beinhaltet mehrfach Abhängigkeiten zu diesen JDK-Paketen. Die jeweils genutzten Klassen sind auf Seite 107 im Anhang aufgelistet. Weitere Abhängigkeiten zur Sprach-API sind die Nutzung des `java.io`-Paketes zur Erzeugung und Ausgabe der Simulationsreports als Datei. An dieser Stelle kommen durch das JDK gekapselte Aufrufe der Plattform-API zum Einsatz, um mit dem Dateisystem der Plattform zu interagieren. `java.beans` wird im Paket `advancedModellingFeatures` dazu genutzt um mithilfe von JavaBeans Vorräte in einem Lagerbestand abzubilden. Die in `java.text` mitgelieferten Formatierungshilfen werden genutzt um Zahlen und Daten in Reports sinnvoll formatiert darzustellen. Die Abhängigkeiten zu den Grafik-APIs `java.awt` und `javax.swing` sind für die Darstellung eines Fortschrittsbalkens bei laufenden Simulationen notwendig. Dies ist jedoch im DESMO-J-Kern die einzige grafische Programmkomponente. Gerade an dieser Stelle können durch den Verzicht auf eine grafische Darstellung des Fortschritts oder eine native Reimplementierung in JavaScript einige Abhängigkeiten vermieden werden.

Über die beschriebenen Abhängigkeiten zur Sprach-API hinaus, nutzt das `core`-Paket von DESMO-J zwei externe Bibliotheken. Dabei handelt es sich zum einen um die Apache Commons-Bibliothek der Apache Software Foundation und zum anderen um Quasar von der Softwarefirma Parallel Universe. Apache Commons ist eine umfangreiche Sammlung von verschiedenen wiederverwendbaren Komponenten, von denen DESMO-J jedoch nur einen Teil verwendet. Die verwendeten Unterbibliotheken sind `org.apache.commons.collections`, eine Sammlung von im JDK nicht vorhandenen oder erweiterten Kollektionstypen und die Mathematikbibliothek `org.apache.commons.math`. Letztere findet hauptsächlich im Paket `dist` Anwendung, wo sie die Berechnung von verschiedenen Arten von Zufallsverteilungen übernimmt. Quasar bietet eine performante Implementation von Threads in der Form von sogenannten Fibers. Diese Fibers können verwendet werden, um nebenläufige Prozesse umzusetzen und sollen laut Entwickler deutlich effizienter arbeiten als Java-eigene Threads [Par18]. In DESMO-J werden Fibers dazu verwendet, Prozesse in Simulationen mit prozessbasierter Sicht abzubilden. Die Klasse `SimProcess` ist als Entität innerhalb einer Simulation definiert, welche einen eigenen prozessartigen Lebenszyklus hat. Dieser interne Lebenszyklus wird über Fibers realisiert, die allerdings nicht nebenläufig ausgeführt werden. Stattdessen ermöglichen Fibers das Unterbrechen und Wiederaufnehmen der Ausführung an beliebigen Code-Anweisungen. Auf Seite 108 im Anhang ist aufgelistet welche Klassen der externen Bibliotheken verwendet werden. Insgesamt besitzt DESMO-J als Codeframework eine geringe Anzahl an Abhängigkeiten zu externen Bibliotheken, was die Portierung begünstigt. Eine Ausweitung der Portierung auf einzelne oder alle Funktionalitäten des Pakets `extensions` würde die Anzahl der auftretenden Abhängigkeiten jedoch deutlich vergrößern. So werden zusätzlich weitere Grafikbibliotheken wie JFreeChart und Java 3D

sowie die XML-Werkzeuge SAX und Apache Xerces vorausgesetzt. Die DESMO-J-Erweiterungen für Visualisierungen und XML-Reports machen intensiven Gebrauch von diesen externen Bibliotheken, wodurch deren Portierung besonders erschwert wird.

Die identifizierten Abhängigkeiten müssen im weiteren Vorgehen der Methode analysiert werden, um festzustellen, wie sie sich auf die Definition des plattformunabhängigen Kerns auswirken. Wie aus der Spezifikation des Konvertierungswerkzeugs JSweet zu entnehmen ist, können einige Abhängigkeiten, insbesondere zu Klassen der Pakete `java.lang` und `java.util` automatisch konvertiert werden. Allerdings sind diese automatischen Übersetzungen oftmals eingeschränkt und unterstützen beispielsweise nicht alle Methoden einer Klasse. So kann JSweet beispielsweise Aufrufe der Methode `equals()`, `getClass()` und `toString()` der JDK-Klasse `Object` übersetzen, jedoch keine weiteren vorhandenen Methoden wie `notify()` oder `wait()` [Paw16b]. Somit muss während der Portierung geprüft werden, welche Abhängigkeiten aufgrund von Konverterfunktionen im automatisch zu portierenden Kern verbleiben können.

Für die anschließende Auftrennung der Gesamtsoftware in Teilfunktionalitäten ist eine relevante Erkenntnis der Abhängigkeitsanalyse auch die Identifizierung von Paketen und Klassen mit besonders vielen Abhängigkeiten. Diese Klassen sind besonders schwer aus der Gesamtsoftware herauszulösen, da sie zur Realisierung ihrer Funktionalität viele weitere Klassen benötigen. Im Gegensatz dazu, werden derart zentrale Klassen auch von vielen anderen Elementen vorausgesetzt. Besonders problematisch sind bei der Aufteilung in Teilfunktionalitäten demnach zyklische Abhängigkeiten zwischen Paketen und Klassen. Im `core`-Paket von DESMO-J ist festzustellen, dass insbesondere Klassen des Unterpaketes `simulator` viele Abhängigkeiten besitzen. Vor allem die für eine Simulation zentralen Klassen `Model`, `Experiment` und deren Oberklassen sind stark miteinander verzahnt und teilweise über zyklische Abhängigkeiten verknüpft. Die meisten anderen Aspekte des Simulationsframeworks wie die Zufallsverteilungen oder die Report-Erstellung bauen auf der Funktionalität dieser zentralen Klassen auf.

| **Paket** | advancedModellingFeatures | dist | exception | observer | report | simulator | statistic | util | **core.*** |
|---|---|---|---|---|---|---|---|---|---|
| java.awt | • | • | • | • | 1 | 6 | • | • | 7 |
| java.beans | 13 | • | • | • | • | 2 | • | • | 13 |
| java.io | • | 1 | • | • | 6 | 2 | • | • | 6 |
| java.lang | 9 | 15 | 5 | • | 13 | 34 | 13 | 4 | 36 |
| java.lang.reflect | 1 | • | • | • | • | 3 | • | • | 4 |
| java.text | • | • | • | • | 2 | 3 | • | • | 5 |
| java.util | 4 | 4 | • | 2 | 9 | 27 | 7 | 3 | 33 |
| java.util.concurrent | • | 1 | • | • | • | 5 | • | • | 5 |
| javax.swing | • | • | • | • | • | 6 | • | • | 6 |

Tabelle 6.1.: Anzahl der Abhängigkeiten von `core`-Unterpaketen zum JDK

| **Paket** | advancedModellingFeatures | dist | exception | observer | report | simulator | statistic | util | **core.*** |
|---|---|---|---|---|---|---|---|---|---|
| *co.paralleluniverse* | | | | | | | | | |
| .common.util | • | • | • | • | • | 1 | • | • | 1 |
| .fibers | 1 | • | • | • | • | 3 | • | • | 3 |
| .strands | • | • | • | • | • | 2 | • | • | 2 |
| .strands.concurrent | • | • | • | • | • | 1 | • | • | 1 |
| *org.apache.commons* | | | | | | | | | |
| .collections | • | • | • | • | • | 3 | • | • | 3 |
| .math | • | 15 | • | • | • | • | 5 | • | 18 |

Tabelle 6.2.: Anzahl der Abhängigkeiten von `core`-Unterpaketen zu externen Bibliotheken

## 6.3. Abspalten einer Teilfunktionalität

Wie bereits erwähnt, stellen Abhängigkeiten zur Sprach-API von Java und zu Drittanbieterbibliotheken große Probleme bei der Portierung einer Software dar. Die hier angewandte Portierungsmethode beinhaltet deswegen eine Abhängigkeitsanalyse, um zu prüfen, welche Abhängigkeiten vorhanden sind. Anhand dessen muss geprüft werden, welche Abhängigkeiten automatisch durch den ausgewählten Konverter übersetzt werden können. Die übrigen Abhängigkeiten müssen vom portierenden Entwickler manuell für die Zielsprache aufgelöst werden. Dieser Arbeitsschritt erfordert in der Regel Refactorings des Ausgangscodes, um die Abhängigkeiten entweder zu isolieren und aus dem plattformunabhängigen Kern auszuschließen oder in eine mit dem Konverter kompatible Form zu überführen. Letzteres ist möglich, falls der Konverter über eine API zur Emulierung von Bestandteilen des JDK verfügt oder die Einbindung von Aufrufen zu JavaScript-Bibliotheken in Java ermöglicht. JSweet unterstützt beide Ansätze, welche im Folgenden erläutert werden.

### 6.3.1. Entwurfsmuster zur Isolation von Abhängigkeiten

Stehle et al. schlagen in ihrer Veröffentlichung zur Portierungsmethode zwei Entwurfsmuster vor, welche bei der Isolation von Abhängigkeiten aus dem Teilfragment helfen können: die Muster Adapter und Generation Gap. Beide dienen grundlegend dazu, Abhängigkeiten zur Sprach- oder Plattform-API, sowie zu Drittanbieterbibliotheken aus dem plattformunabhängigen Kern herauszunehmen und durch einen plattformunabhängigen Aufruf zu ersetzen. Bei beiden Pattern wird eine anschließende Neuimplementierung der Funktionalität in der Zielsprache erfordert. Dabei können dann die nativen APIs der Zielsprache und Plattform benutzt werden. Allerdings muss der Entwickler selbstständig ermitteln, wie die ursprünglichen Abhängigkeiten in der Zielsprache aufgelöst werden können.

Das Adapter-Entwurfsmuster basiert auf der Extraktion des plattformabhängigen Codes in ein Interface. Statt Aufrufe zu nicht unterstützten APIs direkt im ursprünglichen Code durchzuführen, werden sie in ein plattformunabhängiges Interface gekapselt. Dieses Interface kann anschießend auf verschiedenen Plattformen unterschiedlich implementiert werden. In der Originalversion verwendet die Implementierung die zuvor verwendeten APIs und Bibliotheken, mit dem Unterschied, dass durch das eingeschobene Interface eine plattformunabhängige Abstraktionsebene eingeführt wird. Der ursprüngliche Code ruft nach der Umsetzung des Adapter-Musters die Methoden des Interfaces auf, von welchem je nach Plattform unterschiedliche Implementationen verwendet werden können. Für den Konvertierungsvorgang bedeutet das, dass der ursprüngliche Code durch Refactorings so verändert werden muss, dass er das neu definierte Interface verwendet. Der so angepasste Code sowie das Adapter-Interface können in den plattformunabhängigen Kern aufgenommen und somit automatisch konvertiert werden. Lediglich die plattformspezifische Implementation des Adapter-Interfaces muss exkludiert werden. Die Implementation muss anschließend an die automatische Konversion manuell für die Zielplattform entwickelt werden. Bei Konvertierungwerkzeugen, die das Mapping von Typen unterstützen, kann ein Mapping der Implementation der Originalversion auf die Implementation der Zielversion angelegt werden. JSweet unterstützt das Anlegen solcher Mappings nicht, wodurch der JavaScript-Code manuell angepasst werden muss, um die nativ entwickelte Implementation zu verwenden.

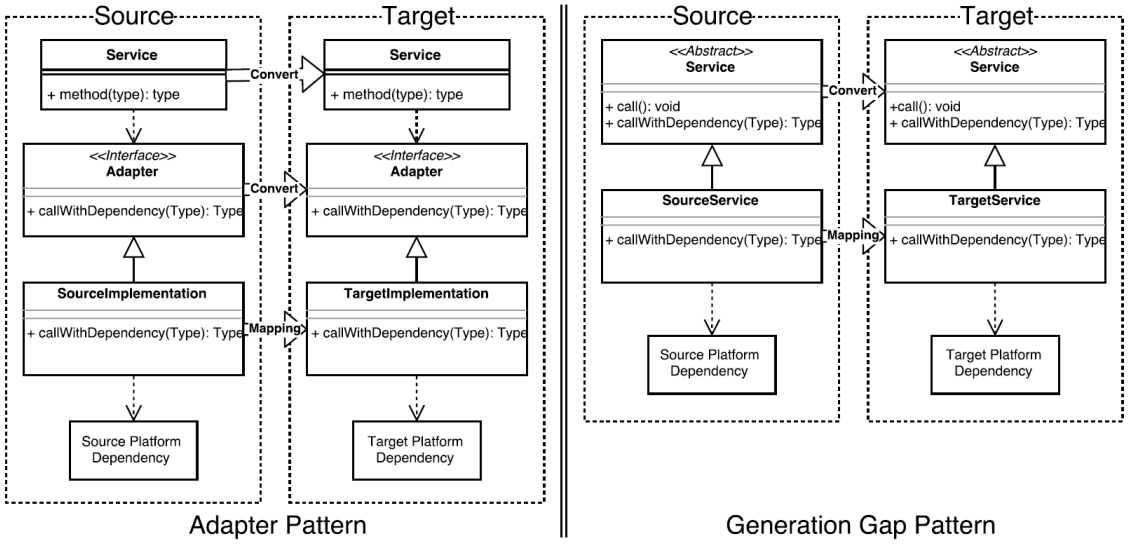


Abbildung 6.2.: Entwurfsmuster zur Isolation von Abhängigkeiten

Einen sehr ähnlichen Ansatz verfolgt das Generation Gap-Pattern. Statt aus einer Klasse die plattformspezifischen Aufrufe zu extrahieren wird hierbei die gesamte Klasse $A$ abstrakt definiert. Methoden, die keine plattformspezifischen Aufrufe enthalten können in dieser abstrakten Klasse implementiert werden. Methoden mit zu isolierenden Abhängigkeiten werden als abstrakt definiert. Die Originalversion definiert eine Unterklasse $A'$ dieser Klasse, welche wiederum die Methoden mit plattformspezifischen Aufrufen implementiert. Für die portierte Zielversion wird eine eigene Unterklasse $B'$ definiert, welche die nötigen Abhängigkeiten nativ in der Zielsprache auflöst. Der Codekonverter kann die abstrakte Oberklasse automatisch als Teil des plattformunabhängigen Kerns übersetzen. Zudem muss ein automatisches oder manuelles Mapping der Unterklasse $A'$ der Originalversion auf die Unterklasse $B'$ der Zielversion vorgenommen werden. Abbildung 6.2 zeigt eine grafische Darstellung der vorgestellten Isolationsmuster.

Grundlegend raten Stehle et al. zur Nutzung des Adapter-Patterns anstatt des Generation Gap-Patterns. Dies liegt primär an der Wiederverwendbarkeit des Adapter-Interfaces und der daran hängenden plattformspezifischen Implementierungen. Das plattformspezifische Verhalten kann durch Nutzung desselben Interfaces auch in anderen Fällen wiederverwendet werden. So kann derselbe Adapter zur Isolierung aller Abhängigkeiten dieses Typs in der zu portierenden Software genutzt werden. Bei der Nutzung des Generation Gap-Patterns ist dies aufgrund der Integration des plattformspezifischen Codes in einer konkreten Klasse nicht möglich [SR17].

Die oben genannten Entwurfsmuster sind insbesondere bei der Betrachtung von statisch typisierten Ausgangs- und Zielsprachen hilfreich. JavaScript als dynamisch typisierte Sprache verfügt jedoch im Gegensatz zu Java generell nicht über die Konzepte eines Interfaces oder einer abstrakten Klasse. Dennoch können Abhängigkeiten auf Java-Seite mithilfe der vorgestellten Entwurfsmuster isoliert werden, um die automatische Übersetzung mit JSweet zu ermöglichen. Der Konverter leistet die sinnvolle Übersetzung der Konzepte zu Code, wobei die manuell zu implementierenden Stellen strukturgleich zur Originalversion aufzufinden sind. Der Vorteil der Isolation von Abhängigkeiten mittels der vorgestellten Entwurfsmuster ist der nur geringfügige Änderungsbedarf und die

weiterhin vorhandene Ausführbarkeit der Originalversion. Letzteres ist bei einer Entfernung der problematischen Abhängigkeit in der Regel nicht möglich.

### 6.3.2. Identifikation und Abspaltung der Teilfunktionalität

Dem Vorgehen der Portierungsmethode folgend, wird die betrachtete Software inkrementell portiert. Dazu werden Teilfunktionen identifiziert und abgespalten. Eine Teilfunktion soll idealerweise in ihrem herausgelösten Zustand weiterhin eine spezifische Funktion erfüllen, die mittels Tests geprüft werden kann. Die Testbarkeit der Teilfunktion ist besonders bei der Evaluation der Portierung vorteilhaft, da sie die semantische Übereinstimmung der portierten Teilfunktionalität mit dem Original aufzeigen kann. Der hier beschriebene Schritt beinhaltet primär die folgenden drei Aufgaben:

- Identifikation geeigneter, testbarer Teilfunktionalitäten
- Identifikation der an der gewählten Teilfunktionalität beteiligten Klassen
- Ersetzen der verwendeten, aber nicht direkt beteiligten Klassen durch Dummy-Implementationen

Die an der Teilfunktionalität beteiligten Klassen sollen mithilfe des Konvertierungstools übersetzt werden. Dies umfasst Klassen, in denen direkt die zu erbringende Funktionalität implementiert ist. Im Blick auf die Gesamtsoftware enthalten diese Klassen jedoch in der Regel Abhängigkeiten zu anderen Klassen der Software, die nicht Teil der Teilfunktion sind. Da die portierten Teilfunktionalitäten in Anschluss wieder zusammengesetzt werden sollen, ist es nicht sinnvoll diese Abhängigkeiten durch Refactorings aus dem Code zu entfernen. Stattdessen werden Dummy- bzw. Mock-Klassen angelegt, die alle notwendigen Aufrufe aus der Teilfunktionalität beinhalten. Bei der Nutzung des JSweet-Plugins in Eclipse ist es notwendig Fehler im Java-Code vollständig zu beheben, bevor der Konvertierungsprozess gestartet werden kann. Demnach müssen Fehler wie fehlende Klassen oder Methoden von Klassen mittels Dummy-Klassen behoben werden. Methoden müssen dabei nicht tatsächlich implementiert werden, es genügt bei Rückgabewerten in der Regel die Rückgabe eines sinnvoll gewählten Dummy-Werts.

In DESMO-J lässt sich als sinnvolle, testbare Teilfunktionalität die Erzeugung von Zufallszahlen mittels Zufallsverteilungen identifizieren. In DESMO-J implementierte Simulationen benötigen oft Zufallswerte, zum Beispiel zur Erzeugung von Ereignissen in zufälligen Zeitabständen oder zufällige Prozesslaufzeiten. DESMO-J bietet für diese Zwecke eine Vielzahl an Zufallsverteilungen, die unterschiedliche Arten von Zufallswerten auf Basis mathematischer Verteilungen erzeugen. Boolesche Verteilungen basieren in DESMO-J auf der Klasse `BoolDist`, wohingegen numerische Verteilungen auf der abstrakten Oberklasse `NumericalDist` im Paket `core.dist` basieren. Diese Klassen erben wiederum von der abstrakten Oberklasse `Distribution`, welche die grundlegende Schnittstelle für Verteilungen liefert. Eine mögliche Teilfunktion wäre die Generierung von zufälligen Proben von Verteilungen mittels der Methode `sample()`. Um eine weitere Unterteilung der Teilfunktionalität zu erreichen können die verschiedenen Arten von Implementierungen dieser Verteilungen betrachtet werden. Es lässt sich erkennen, dass neben booleschen Verteilungen mit zwei konkreten Implementationen auch kontinuierliche numerische (12 Implementationen) und diskrete numerische Verteilungen (7 Implementationen) existieren. Zudem gibt es weitere Arten von Verteilungen die in diesem Beispiel nicht betrachtet werden. Auf Basis dieser Kategorisierung können weitere Teilfunktionen identifiziert werden.

Ein sinnvoller Startpunkt ist die Betrachtung der booleschen Verteilungen als wichtige Basisfunktionalität mit wenigen Abhängigkeiten zu funktional unterschiedlichen Klassen. DESMO-J enthält zwei Arten von booleschen Verteilungen: `BoolDistConstant` und `BoolDistBernoulli`. Erstere gibt bei jeder Probe den bei Erzeugung gewählten booleschen Wert zurück. Letztere erzeugt einen zufälligen booleschen Wert bei einer festgelegten Wahrscheinlichkeit für wahr. Als testbares Kriterium für diese Teilfunktionalität wird das korrekte Verhalten der `sample()`-Methode herangezogen. Dazu gehören das Liefern eines booleschen Wertes anhand der Parameter der Verteilung und übereinstimmende Ergebnisse bei der Wahl desselben Zufalls-Seeds. Die Wiederholbarkeit von Simulationen unter denselben Bedingungen ist gerade bei der Verwendung von Zufallszahlen ein kritischer Faktor. Aus diesem Grund müssen alle Verteilungen in DESMO-J einen numerischen Wert als Seed entgegennehmen können, der dafür sorgt, dass bei der Wahl desselben Seeds dieselben Zufallszahlen generiert werden. Dies ist in der Klasse `Distribution` mittels der Methode `setSeed(long)` möglich, welche den Seed an den Zufallszahlengenerator weitergibt.

Um die booleschen Verteilungen als Teilfunktionalität abzuspalten müssen zunächst die direkt beteiligten Klassen identifiziert werden. Dies sind im Kern die folgenden acht Klassen im Paket `core.dist`:

- `BoolDist`, `BoolDistBernoulli`, `BoolDistConstant`, `Distribution`, `DistributionManager`, `LinearCongruentialRandomGenerator`, `MersenneTwisterRandomGenerator`, `UniformRandomGenerator`

`BoolDistConstant` und `BoolDistBernoulli` sind dabei die Klassen, welche im Rahmen einer Simulation instanziiert werden um boolesche Werte zu erzeugen. `BoolDist` ist deren gemeinsame abstrakte Oberklasse, welche die Schnittstelle für boolesche Verteilungen definiert. Die Verteilungsklassen nutzen Zufallsgeneratoren die das Interface `UniformRandomGenerator` implementieren. DESMO-J enthält den `LinearCongruentialRandomGenerator` auf Basis des JDK-eigenen `Random`-Objekts und den `MersenneTwisterRandomGenerator`, welcher den Mersenne Twister-Algorithmus verwendet. Darüber hinaus soll es möglich sein, eigene benutzerdefinierte Implementationen von `UniformRandomGenerator` zu übergeben, um die Zufallszahlengenerierung anzupassen. Diese Klassen haben Abhängigkeiten zu einer Vielzahl anderer Klassen des Frameworks, welche jedoch nicht direkt in die Funktionalität involviert sind:

- **core.report**: `BoolDistBernoulliReporter`, `BoolDistConstantReporter`, `DistributionReporter`, `Reporter`
- **core.simulator**: `Experiment`, `Model`, `ModelComponent`, `NamedObject`, `Reportable`, `TimeInstant`

Um die Teilfunktionalität herauszulösen müssen diese Klassen entweder in die Teilfunktionalität integriert werden oder durch Dummy-Implementationen ersetzt werden. Dazu wird der Code in den oben genannten Klassen der Teilfunktionalität nicht abgeändert, um die spätere Reintegration nicht zu erschweren. Quelltext 6.1 zeigt beispielhaft den Einsatz von Dummy-Klassen zur Abdeckung von nicht direkt relevanten Aufrufen innerhalb der Teilfunktionalität.

Auf Basis der herausgelösten Klassen kann nun die Isolation des plattformübergreifenden Kerns für die Teilfunktionalität „boolesche Verteilungen“ folgen. Die oben erwähnten numerischen Verteilungen verwenden wie auch die booleschen Verteilungen die genannten Oberklassen und Zufallsgeneratoren. Demnach könnte es sinnvoll sein, weitere Teilfunktionen zu identifizieren, die

```
1  //Aufruf in core.dist.Distribution, owner ist Model
2  owner.getExperiment().getDistributionManager().register(this);
3
4  public class Model { //core.simulator.Model-Dummy
5    private Experiment experiment = new Experiment();
6
7    public Experiment getExperiment() { return experiment; }
8  }
9
10 public class Experiment { //core.simulator.Experiment-Dummy
11   DistributionManager distMan = new DistributionManager();
12
13   public DistributionManager getDistributionManager() { return distMan; }
14 }
15
16 public class DistributionManager { //core.simulator.DistributionManager-Dummy
17   public void register(Distribution dist) { /*register*/ }
18 }
```

Quelltext 6.1: Dummy-Klassen für einen Aufruf in Distribution

auch diese Grundbausteine nutzen. Beispielsweise könnte als zweite Teilfunktion die Gruppe der kontinuierlichen Verteilungen mit der auf `Distribution` und `NumericalDistribution` basierenden Oberklasse `ContDist` betrachtet werden. Auf Seite 108 im Anhang ist zu erkennen, welche Klassen an dieser Teilfunktionalität beteiligt sind.

### 6.3.3. Isolation des plattformübergreifenden Kerns

Nach der Abhängigkeitsanalyse und der Abspaltung einer Teilfunktionalität folgt mit der Bestimmung und Isolation des plattformunabhängigen Kerns der Funktionalität der abschließende Schritt der ersten Phase der Portierungsmethode. Der plattformunabhängige Kern besteht aus allen Codeelementen die automatisch durch das Portierungswerkzeug übersetzt werden können. Dies erfordert in der Regel die Auflösung aller plattformspezifischen Abhängigkeiten. Je nach Konverter können solche Abhängigkeiten jedoch durch Refactorings automatisch übersetzbar gemacht werden. Je umfangreicher der plattformunabhängige Kern definiert werden kann, desto weniger manueller Aufwand fällt im Nachhinein an. JSweet verfügt dank der API des Werkzeugs über einige Hilfsmittel, die die Maximierung des automatisch konvertierbaren Codes unterstützen.

Das JSweet-Plugin für Eclipse gibt bei der Erkennung von für die Übersetzung problematischen Stellen wertvolle Hinweise. Anhand dieser Fehlermeldungen kann entschieden werden, ob die problematische Stelle durch Refactorings abgeändert oder isoliert werden soll. Für das Beispiel der Teilfunktionalität „boolesche Verteilungen“ tritt ein Fehler bei der verwendeten JDK-Klasse `Random` auf. Diese wird von den Zufallszahlengeneratoren `LinearCongruentialRandomGenerator` und `MersenneTwisterRandomGenerator` genutzt, um Zufallszahlen mit einem bestimmten Seed zu erzeugen. Der Typ `Random` kann als nicht übertragbarer Teil des JDK nicht automatisch übersetzt werden. Im Rahmen des gemeinsamen Interfaces `UniformRandomGenerator` soll die Methode `nextDouble()` eine zufällige Gleitkommazahl zwischen 0 und 1 erzeugen. Diese Funktionalität wird auch von dem JDK-Aufruf `Math.random()` abgedeckt, welcher zudem von JSweet automatisch zum äquivalenten `Math.random`-Aufruf in JavaScript übersetzt werden kann. Somit wäre ein Refactoring der problematischen Codeelemente zu dieser kompatiblen Version denkbar. Wie allerdings oben beschrieben, ist

die Erzeugung von Zufallszahlen auf Basis eines Seeds aufgrund der Wiederholbarkeit von Simulationen essentiell. Dem Typ `Random` können Seeds übergeben werden, wohingegen `Math.random()` dies nicht unterstützt. Ein solches Refactoring würde also eine der testbaren Eigenschaften der Teilfunktionalität beeinträchtigen. Aus diesem Grund sollte eine Möglichkeit gefunden werden, sinnvollere Refactorings für den problematischen Typ `Random` zu finden.

Um Abhängigkeiten zur Plattform oder externen Bibliotheken aufzulösen, bietet JSweet die Möglichkeit JavaScript-Bibliotheken aus Java-Code heraus aufzurufen. Dazu werden sogenannte JSweet-Candies verwendet, welche als Java-Adapter für JavaScript-Bibliotheksaufrufe dienen. Konkret bedeutet dies, dass aufgrund von Abhängigkeiten nicht übersetzbare Aufrufe durch Aufrufe auf funktionsgleiche JavaScript-Bibliotheken ersetzt werden. Dadurch wird die ursprüngliche Abhängigkeit aufgelöst und das problematische Codeelement automatisch übersetzbar gemacht. Der generierte JavaScript-Code enthält daraufhin Aufrufe auf die eingebundene JavaScript-Bibliothek. Um diesen Mechanismus nutzen zu können müssen Java-Adapter für Bibliotheken geschrieben werden oder als Maven-Abhängigkeit in das JSweet-Projekt integriert werden. Die Dokumentation von JSweet enthält eine Liste bereits veröffentlichter und importierbarer Candies [Paw16a].

Die benötigte Funktionalität eines Zufallszahlengenerators mit Seed findet sich in der JavaScript-Bibliothek „RandomJS“, welche ein `Random`-Objekt definiert. Dessen Funktionalität deckt sich stark mit der des JDK-`Random`-Typs. Für JSweet ist ein Candy dieser Bibliothek verfügbar, welches genutzt werden kann um ein Refactoring der problematischen Abhängigkeit durchzuführen. Somit muss die Funktionalität nicht manuell neu implementiert werden, sondern wird automatisch auf die plattformspezifische Bibliothek gemappt. Quelltext 9 und 10 ab Seite 109 zeigen das vorgenommene Refactoring mittels Candy am Beispiel des `MersenneTwisterRandomGenerator`. Dabei kommt grundlegend das Adapter-Pattern zum Einsatz, allerdings gibt die Codestruktur mit `UniformRandomGenerator` bereits ein gemeinsames Interface für alle Zufallsgeneratoren vor, sodass in diesem Fall kein Interface zur Umsetzung des Entwurfsmusters extrahiert werden muss. Stattdessen kann die zu portierende Klasse `MersenneTwisterRandomGeneratorJS` das Interface implementieren und somit problemlos mit dem Rest des Codes zusammenwirken. Leider ermöglicht JSweet jedoch nicht automatische Mappings von Typen. Daher ist es nicht möglich die in Java verwendete Klasse während der Konversion zur explizit angelegten JavaScript-Klasse abzuändern. Stattdessen müssen Aufrufe auf `MersenneTwisterRandomGenerator` im Java-Code durch die abgeänderte Implementation ersetzt werden. Hierbei geht jedoch die Ausführbarkeit der Java-Version verloren, da die Candy-Aufrufe im nicht übersetzten Zustand nicht funktional sind. Dies ist allerdings aufgrund der zu Beginn der Portierung auswählten Option einer Portierungsversion akzeptabel.

Der durch das Refactoring angelegte Typ `MersenneTwisterRandomGeneratorJS` erlaubt das Übergeben eines Seeds, auf dessen Basis ein Zufallsgenerator auf Basis des Mersenne Twister-Algorithmus aus der RandomJS-Bibliothek erzeugt wird. Dieser gibt über die vom Interface vorgegebene Methode `nextDouble()` eine Gleitkommazahl zwischen 0 und 1 zurück. In Quelltext 10 ist zu erkennen, dass der Konstruktor ohne Übergabeparameter für den Seed auskommentiert ist. Dies liegt an Limitationen von TypeScript bezüglich überladener Konstruktoren. Dieser Konstruktor erzeugt im Original einen Zufallsgenerator mit dem Standard-Seed 42. Dieses Verhalten muss durch den Wegfall des Konstruktors über die Verwendung des Konstruktors `MersenneTwisterRandomGeneratorJS(long)` abgebildet werden. Im betrachteten Teilfragment erzeugt jedes `Distribution`-Objekt einen eigenen Zufallsgenerator auf Basis eines im `DistributionManager` hinterlegten Standard-Generators. Dieser

Standardgenerator kann gewechselt werden und hat daher die generische Form `Class<? extends UniformRandomGenerator>`. Die Methode `getRandomNumberGenerator()` des `DistributionManager` liefert ohne weitere Konfiguration `MersenneTwisterRandomGenerator.class`, welche in `Distribution` instanziiert wird. Um das Refactoring abzuschließen muss die Methode stattdessen `MersenneTwisterRandomGeneratorJS.class` liefern. Idealerweise ließe sich dieses Mapping automatisch durchführen, allerdings ist dies in JSweet nicht möglich. In diesem Fall muss eine zuvor angelegte Dummy-Klasse die für die korrekte Ausführung der Teilfunktionalität relevante Logik beinhalten. Des Weiteren muss der Instanziierungscode in `Distribution` abgeändert werden, um die automatische Übersetzung von Generics in Jsweet nutzen zu können. Dies ist in Quelltext 11 auf Seite 111 im Anhang zu sehen. Durch diese transformationsvorbereitenden Refactorings kann eine Abhängigkeit zu vom Konverter nicht unterstützten Plattformaufrufen aufgelöst werden. Dies bedeutet, dass die Generierung von Zufallszahlen im automatisch zu konvertierenden Kern verbleiben kann.

Die Isolation des plattformunabhängigen Kerns der zweiten identifizierten Teilfunktionalität birgt einige andere Herausforderungen. Primär sind die Berechnungen für die kontinuierlichen Verteilungen auf Abhängigkeiten zur externen Mathematik-Bibliothek Apache Commons Math angewiesen. Generell lässt sich feststellen, dass Aufrufe dieser Bibliothek in den Implementationen der abstrakten Methode `getInverseOfCumulativeProbabilityFunction(double)` aus `NumericalDist` auftreten. Davon betroffen sind die folgenden Klassen der Teilfunktionalität:

- `ContDistCustom`, `ContDistErlang`, `ContDistGamma`, `ContDistNormal`

Unter den veröffentlichten JSweet-Candies konnte keine Bibliothek gefunden werden, welche die in dieser Methode getätigten Berechnungen ausführen kann. Insofern ist es ohn manuelle Implementierung eines JSweet-Candy nicht möglich, ein Refactoring der externen Abhängigkeit zu einer JavaScript-Bibliothek durchzuführen. Somit muss die dort ausgeführte Funktionalität vom plattformunabhängigen Kern ausgeschlossen und im Anschluss an die Konversion nativ in JavaScript entwickelt werden. Zum Zwecke der Isolation wird der Methodenkörper auskommentiert und durch eine `UnsupportedOperationException` ersetzt. An dieser Stelle muss eine manuelle Anpassung folgen. Alternativ kann die Methode mit einer `@Replace`-Annotation von JSweet annotiert werden. Diese Annotation ersetzt den Körper der annotierten Methode mit dem darin angegebenen TypeScript-Code. So ist es möglich, die manuelle Reimplementation innerhalb der Portierungsversion und vor der Portierung durchzuführen.

## 6.4. Konversion

Nach der Abspaltung einer Teilfunktionalität und der Bestimmung des plattformunabhängigen Kerns kann nun die Konversion des Codes mithilfe der ausgewählten Werkzeugunterstützung folgen. Auf die Portierungsmethode nach Stehle et al. bezogen, wird dieser Arbeitsschritt in die zweite Phase eingeordnet. Diese umfasst neben der Übersetzung des Teilfragments auch die Erzeugung von Traceability zwischen übereinstimmenden Codeelementen im Original und der portierten Version. Abschließender Punkt der zweiten Phase ist eine Evaluation des Generats bezüglich Qualität, Strukturgleichheit und manuellem Portierungsaufwand. Um diese Arbeitsschritte zu beschreiben widmet sich dieses Kapitel zuerst der Konversion problematischer Sprachkonstrukte

und Codeelemente im Beispielprojekt DESMO-J. Anschließend wird der generierte Code analysiert und bezüglich des aufgetretenen manuellen Aufwands evaluiert.

Für die Konversion wird die zur Bearbeitungszeit dieser Arbeit aktuelle Version 2.0.0-rc1 von JSweet verwendet. Die zu portierende Teilfunktionalität liegt als ein JSweet-Projekt in der Entwicklungsumgebung Eclipse vor. JSweet-Projekte nutzen das Build-Management-Tool Maven um den Konverter zu konfigurieren und zusätzliche Konverterartefakte wie Candies einzubinden. In der *pom.xml*-Datei des Projektes wird die verwendete Version des Transpilers und der JSweet-Core-Bibliothek festgelegt. Bei der Verwendung des JSweet-Plugins wird der Übersetzungsvorgang automatisch bei Änderungen des Projektcodes durchgeführt, vorausgesetzt Eclipse meldet keine Fehler im Java-Code und JSweet kann keine inkompatiblen Codeelemente identifizieren. Bei der Prüfung durch JSweet wird analysiert, ob das als intermediäres Medium generierte TypeScript syntaktisch korrekt generiert werden kann. Falls nicht, zeigt das Plugin Fehlermeldungen mit TypeScript-Fehlerbeschreibungen an. Wenn sowohl die Entwicklungsumgebung als auch JSweet keine Fehler feststellen können, wird der Projektcode übersetzt und im konfigurierten Outputverzeichnis ausgegeben.

### 6.4.1. Konversion problematischer Sprachkonstrukte

Aufgrund des oben beschriebenen Verhaltens des Konvertierungswerkzeugs und der vorangegangenen Wahl der Portierungsstrategie, ist ein notwendiger Teil der Konversion von Teilfunktionalitäten die Behebung von bestehenden Fehlern im JSweet-Projekt. Dazu gehören zum einen durch die Abspaltung der Teilfunktionalität entstandene Fehler aufgrund von fehlenden Abhängigkeiten. Zum anderen können Fehler bezüglich der Kompatibilität der verwendeten Codestrukturen mit dem Funktionsumfang des Konverters auftreten. Grundsätzlich erlauben die umfangreiche Konverter-API und JSweet-Candies die Definition eines sehr umfangreichen plattformübergreifenden Kerns, da problematische Abhängigkeiten oftmals ersetzt werden können, statt entfernt werden zu müssen. Auch wenn dies den manuellen Aufwand bei der Erzeugung einer Portierungsversion der Software erhöht, sinkt der nachträgliche Reimplementierungsaufwand in JavaScript deutlich. Dieser Abschnitt zeigt problematische Sprachkonstrukte und Codestrukturen auf, die im Rahmen der durchgeführten Portierung identifiziert wurden. Dabei wird insbesondere betrachtet, wie derartige Probleme bei einer Portierung gelöst werden können bzw. welche Auswirkungen sie auf den gesamten Portierungsprozess haben.

Im Laufe der in Unterabschnitt 6.3.2 durchgeführten Auflösung von Abhängigkeiten wurden konkrete Aufrufe zu externen Bibliotheken oder JDK-Klassen durch Aufrufe auf die Konverter-API, äquivalente JSweet-Candies oder Dummy-Aufrufe ersetzt. Für die bisher nicht funktionalen Dummy-Aufrufe muss entschieden werden, ob eine Nachbildung des Aufrufs mit automatisch übersetzbaren Sprachkonstrukten in Java möglich ist. Alternativ wird der Dummy-Aufruf während der Übersetzung beibehalten und anschließend im Generat in JavaScript implementiert. Ersteres Vorgehen verschiebt den manuellen Implementierungsaufwand in die Erzeugung der Portierungsversion vor die automatische Konversion. Dies kann insbesondere in Fällen sinnvoll sein, in denen bei den portierenden Entwicklern eine bessere Kenntnis von Java als von JavaScript vorliegt.

Bei der Portierung konnte festgestellt werden, dass JSweet eine Vielzahl an Klassen des JDK nicht automatisch auf JavaScript abbilden kann. Dies liegt oftmals jedoch nicht an der reinen Portierbarkeit

des darin verwendeten Codes, sondern daran, dass die JDK-Klassen nicht als direkter Teil der Codebasis der zu portierenden Software vorliegen. Beispiele dafür sind unter anderem `java.util.Observable`, `java.util.concurrent.TimeUnit` oder `java.lang.Integer`. Die in DESMO-J genutzten Funktionalitäten dieser Klassen sind in den meisten Fällen in Adapter-Klassen nachzubilden, indem der benötigte Code übernommen wird. Nicht benötigte Methoden werden in den Adaptern dabei entfernt um die Komplexität zu verringern. Die Klasse `TimeUnit` wird in DESMO-J vielfach dazu verwendet, Zeiteinheiten ineinander umzurechnen. Jegliche Aufrufe auf `java.util.concurrent.TimeUnit` werden von JSweet als Fehler erkannt, da der Konverter nicht über die nötigen Makros zur automatischen Übersetzung verfügt. Der für die Nachbildung des Verhaltens dieser Klasse benötigte Code innerhalb eines Adapters lässt sich jedoch übersetzen. Um den Änderungsaufwand im Originalcode zu minimieren, ist es sinnvoll den Adapter mit demselben Bezeichner zu benennen. So reicht die Abänderung der Importanweisung in Klassen, die die problematische JDK-Klasse nutzen, um den Aufruf auf den selbstgeschriebenen Adapter umzulenken.

An zwei Stellen nutzt DESMO-J Sammlungstypen der externen Bibliothek Apache Commons Collections. Der Konverter unterstützt allerdings nur die automatische Übersetzung von Sammlungsstypen des JDKs unter Zuhilfenahme von JavaScript-Arrays (siehe Kapitel 5). Die Nutzung der Sammlungstypen `ReferenceMap` in der Klasse `Model` und `TreeList` in `EventTreeList` dienen grundsätzlich der Performance-Optimierung. Dabei minimieren die in der Originalversion verwendeten Sammlungstypen den bei vielen Schreib- und Lesezugriffen auftretenden Aufwand gegenüber regulären Kollektionstypen des JDK wie `ArrayList` oder `HashMap`. Allerdings muss beachtet werden, dass in der vom Konvertierungswerkzeug erzeugten ECMAScript-Version keine Sammlungstypen vorhanden sind. Somit sind alle Arten von Sammlungen im JavaScript-Generat gleichartig über Arrays realisiert, wobei Unterschiede bei der Performance verloren gehen. Dementsprechend können Kollektionstypen von Apache Commons Collections durch Java-Kollektionen mit äquivalentem Verhalten ersetzt werden. Dies wird über die gemeinsamen Schnittstellen `List` und `Map` aller Kollektionen in Java erleichtert. Durch diese Änderung wird die automatische Übersetzung von Collections aus externen Bibliotheken ohne Veränderung des Verhaltens ermöglicht. Inwiefern die Abbildung dieser Kollectionen über JavaScript-Array Auswirkungen auf die Performance von Simulationen hat, müsste in weiteren Untersuchungen ermittelt werden. Zudem wäre eine Nutzung des in ECMAScript 6 eingeführten `Map`-Typs denkbar, wobei dies einen erhöhten manuellen Aufwand bei der Nachbearbeitung des Generats auf JavaScript-Seite verursachen würde.

Wie bereits in Abschnitt 6.2 angesprochen, stellt die Nutzung von Java Reflection eine Herausforderung bei der Konversion zu JavaScript dar. Reflection bietet die Möglichkeit zur Laufzeit der Software auf Eigenschaften eines Objekts zuzugreifen. DESMO-J nutzt diese Funktionalität an mehreren Stellen, um die Validität von benutzerdefinierten Klassen zu prüfen. Das Framework basiert an vielen Stellen auf der Implementierung benutzerdefinierter Klassen zur Anpassung der Funktionalität bezüglich der zu implementierenden Simulation. DESMO-J bietet in der Regel abstrakte Basisklassen von denen die benutzerdefinierte Implementation erbt. Um Fehler bei der Ausführung der Simulation zu vermeiden, werden die benutzerdefinierten Klassen mittels Reflection auf Validität geprüft. Bei Problemen wird stattdessen eine Standardimplementation des Frameworks genutzt. Ein Beispiel dafür ist die Nutzung benutzerdefinierter `Reporter`. Diese Klassen dienen dazu, den Output im Simulationsbericht für ein `Reportable` zu erzeugen. Um die Ausgabe den

Anforderungen einer bestimmten Simulation anzupassen, kann eine benutzerdefinierte Klasse den abstrakten `Reporter` erweitern und einem Reportable übergeben. Beim Aufruf dieses Reporters wird über Reflection geprüft, ob die benutzerdefinierte Klasse über den benötigten Konstruktor verfügt. Ein weiteres Beispiel ist die Verwendung von unterschiedlichen Implementierungen von Event-Listen (abstrakte Klasse `EventList`) zur Nutzung in einem Experiment. Beim Setzen einer Implementation von `EventList` in `Experiment` wird über Reflection geprüft, ob es sich bei der übergebenen Klasse um eine vollständige Implementation und nicht um ein Interface oder eine abstrakte Klasse handelt. Der Wechsel der Event-Liste wird nur im ersten Fall durchgeführt.

Das Konvertierungswerkzeug JSweet unterstützt die automatische Übersetzung von Java Reflection-Aufrufen in den meisten Fällen nicht. Zudem ist die Abbildung der Konzepte Interface und abstrakte Klasse wie in Abschnitt 5.1 beschrieben in JavaScript nicht möglich. Aus diesem Grund stellt die Prüfung der Validität von übergebenen Objekten für bestimmte Nutzungskontexte eine Herausforderung dar. Theoretisch möglich ist jedoch eine Prüfung der vorhandenen Eigenschaften eines Objekts in JavaScript anhand deren Bezeichner. Diese Prüfung kann entweder in der Java-Portierungsversion mithilfe der JSweet-API oder als nachträgliche Umsetzung in JavaScript realisiert werden. Dabei lässt sich für ein Objekt prüfen, ob dieses Eigenschaften mit einem bestimmten Bezeichner besitzt. Über diesen Mechanismus kann somit erkannt werden, ob das übergebene Objekt die vom Interface oder der abstrakten Klasse in Java vorgegebenen Methoden und Attribute enthält. Allerdings wird dabei lediglich das Vorhandensein des bestimmten Bezeichners geprüft. Eine übereinstimmende Eigenschaft kann also sowohl eine Variable als auch eine Funktion sein. Dies muss über weitere Prüfungen bestimmt werden (z.B. `propertyName instanceof Function`). Dennoch ist eine derartige Prüfung von Objekten bezüglich einer Implementation eines Interfaces oder Vererbung von einer Oberklasse unzuverlässig. Dies liegt primär daran, dass der generierte JavaScript-Code keine Bezüge für die Bindung an einen gegebenen Schnittstellenvertrag enthält. Ein Objekt in JavaScript kann unter Umständen alle für die Validierung notwendigen Eigenschaften enthalten ohne das von den ursprünglichen Oberklassen definierte Verhalten abzubilden. Die mangelnde Typsicherheit in JavaScript erschwert den Validierungsprozess. Ähnliche Limitationen treffen auch auf die in ECMAScript 6 eingeführte Funktionalität des `Reflect`-Objekts zu. Zudem erfordert eine Umsetzung einer derartigen Prüfung, insbesondere angesichts der unzureichenden Funktionsfähigkeit einen großen Mehraufwand.

Aufgrund dieser Aspekte wurde Reflection-Code in den Klassen `Reportable`, `Schedulable` und `Experiment` in eigene Methoden extrahiert, deren Körper durch die JSweet-Annotation `@Replace` durch eine „Noch nicht implementiert“-Warnung ersetzt wird. In allen betrachteten Fällen schränkt dies nicht die Ausführbarkeit von Simulationen ein, stattdessen wird lediglich die Möglichkeit benutzerdefinierte Implementationen zu übergeben entfernt, um die automatische Übersetzung zu ermöglichen. Grundsätzlich ist eine Reimplementation dieser Funktionalität in JavaScript möglich, jedoch stellt die Validierung der übergebenen Objekte eine Herausforderung dar. Probleme bei der Ausführung mit benutzerdefinierten Objekten können unabhängig von einer Validierung im Programm durch sorgfältige Prüfung der Voraussetzungen durch den Entwickler verhindert werden. Dies erhöht allerdings die Komplexität der Implementation benutzerdefinierter Simulationen.

Grafische Benutzeroberflächen stellen bei der Portierung zu JavaScript eine Herausforderung dar. Auf der Java-Seite kommen dabei, wie auch in DESMO-J, die Grafikbibliotheken AWT und Swing

zum Einsatz, welche konfigurierbare GUI-Widgets anbieten, die in Containern wie Programmfenstern angeordnet werden können. Browserbasierte JavaScript-Anwendungen nutzen hingegen in der Regel das Document Object Model (DOM) von HTML-Seiten, um die darauf befindlichen grafischen Elemente zu manipulieren. Diese Konzepte sind aufgrund der unterschiedlichen Paradigmen schwer aufeinander abzubilden, weshalb JSweet keine automatische Übersetzung von GUI-Code durchführen kann. Dies ist insbesondere bei Software mit umfangreicher grafischer Benutzeroberfläche ein Problem, da die notwendige manuelle Portierung deutlichen Mehraufwand erfordert. Der Kern des DESMO-J-Frameworks enthält jedoch lediglich das Anzeigen eines Fortschrittsbalkens für durchgeführte Experimente als grafische Oberfläche. Diese Anzeige ist für die korrekte Ausführung von Simulationen nicht relevant und dient lediglich als Feedback für den Nutzer. Aus diesem Grund ist zu erwägen, das Anzeigen des Fortschrittsbalkens im Rahmen der Erzeugung einer Portierungsversion zu entfernen. Somit können problematische Abhängigkeiten zu den JDK-Komponenten AWT und Swing vermieden werden. Dazu können die während der Abhängigkeitsanalyse gefundenen Klassen mit diesen Abhängigkeiten abgeändert werden. Die grafischen Aspekte des Fortschrittsbalken sind in der Klasse `ExpProgressBar` gekapselt, deren Erzeugung in `Experiment` entfernt werden kann. In der durchgeführten Portierung wurde der GUI-spezifische Code in `Experiment` in eine eigene Methode `showProgressBar()` extrahiert und mittels JSweet-Annotation aus dem Generat entfernt. Grundsätzlich bestehen verschiedene Möglichkeiten die entfernte GUI nachträglich zu reimplementieren. Zum einen verfügt die Konverter-API über Möglichkeiten das DOM einer HTML-Seite durch Java-Code zu manipulieren. So kann in der Portierungsversion eine alternative Implementation innerhalb der extrahierten Methode realisiert werden. Zum anderen kann eine grafische Repräsentation des Fortschritts auch in TypeScript oder JavaScript im Generat umgesetzt werden. Diese Methoden basieren alle auf der Nutzung von HTML-Elementen zur Darstellung der grafischen Oberfläche im Browser. Die hier durchgeführte Portierung profitiert durch die geringe Menge an GUI-spezifischem Code in der Beispielsoftware. Im Hinblick auf eine weitere Portierung von DESMO-J wird insbesondere der als Erweiterung verfügbare grafische Assistent zur Simulationsausführung problematisch sein.

Bei der durchgeführten Abhängigkeitsanalyse konnte festgestellt werden, dass DESMO-J Abhängigkeiten zum JDK-Paket `java.util.concurrent` und zur externen Bibliothek Quasar enthält. Concurrency, also Nebenläufigkeit bezeichnet generell das Programmierkonzept mehrerer Threads zur Laufzeit eines Programms. In Java ist es möglich, mithilfe von Threads die Ausführung eines Programms auf mehrere Prozesse zu verteilen. So können Performancevorteile oder, im Falle von grafischen Anwendungen, nicht blockierende Software erreicht werden. In JavaScript wird Nebenläufigkeit grundsätzlich nicht unterstützt. Bei der Ausführung von JavaScript im Browser wird generell immer nur ein Skript, bzw. eine Abfolge von Anweisungen bearbeitet. Dennoch ist der Aufruf anderer Skripts oder die Reaktion auf Events möglich, allerdings wird in diesen Fällen die Ausführung des Skripts falls nötig unterbrochen und später fortgeführt [Hav14]. Aus diesem Grund stellt Nebenläufigkeit in Java-Programmen ein Problem bei der Portierung zu JavaScript dar, da es für dieses Konzept keine direkte Entsprechung gibt. Basierend auf dieser Einschränkung von JavaScript unterstützt auch das Konvertierungswerkzeug keine automatische Übersetzung von Code mit Abhängigkeiten zu den Teilen des JDK, die Nebenläufigkeit ermöglichen. Darunter fallen zum Beispiel die Klassen `Thread` und `Lock` aus `java.util.concurrent`, welche auch im Framework-Kern von DESMO-J genutzt werden. Um klären zu können, inwiefern dieser Umstand

die Portierung beeinflusst, muss untersucht werden, auf welche Weise Nebenläufigkeit bzw. die damit einhergehenden Klassen genutzt werden.

Wie in Abschnitt 3.5 erwähnt, unterstützt DESMO-J die Implementierung von Simulationen aus event- und prozessbasierter Sicht. Erstere lässt sich als Menge von Ereignissen auf einem Zeitstrahl beschreiben, wobei sich die Programmausführung linear an diesem Zeitstrahl orientiert. Um dies zu realisieren greift DESMO-J nicht auf Nebenläufigkeit zurück. Bei der prozessbasierten Sicht werden nicht Ereignisse betrachtet, sondern Prozesse mit einem eigenen Lebenszyklus, die einen Start- und Endpunkt auf dem Zeitstrahl besitzen. Der Lebenszyklus eines Prozesses bildet dabei den Prozessfortschritt und dabei auftretende Veränderungen ab. Die Ausführung des Lebenszyklus läuft dabei parallel zum fortlaufenden Zeitrahmen des Experiments ab. An dieser Stelle ist eine nebenläufige Ausführung von Prozessen neben dem eigentlichen Experiment denkbar. Tatsächlich sind Prozesse in DESMO-J in der Klasse `SimProcess` mithilfe von Threads des JDK oder Fibers von Quasar umgesetzt. Die Implementation über Fibers ist funktional äquivalent, verspricht aber bei einer hohen Anzahl von Prozessen eine bessere Performance. Obwohl Abhängigkeiten zu den Klassen `Thread` und `Fiber` (bzw. `Strand`) im Code vorhanden sind, ist allerdings festzustellen, dass diese bei der Ausführung der Simulation nicht nebenläufig ausgeführt werden. Tatsächliche Nebenläufigkeit tritt in DESMO-J nicht auf. Stattdessen werden die Prozesse je nach Bedarf gestartet und gestoppt. Ein laufender Prozess läuft allerdings nicht parallel zur Simulation, sondern unterbricht deren Ausführung zeitweilig. Obwohl Prozesse nicht nebenläufig ausgeführt werden, ist die Nutzung von Threads in Java jedoch sinnvoll, da die Ausführung von Threads an einer beliebigen Stelle im Code pausiert und später wiederaufgenommen werden kann. Somit ist eine Unterbrechung eines `SimProcess` durchaus auch während der Ausführung einer Methode möglich. Im Gegensatz zu anderen Methoden der Unterbrechung, wie zum Beispiel über Callbacks, bietet dies den Vorteil die Ausführung von der genauen Stelle (bzw. Codeanweisung) der Unterbrechung an fortzusetzen. Somit dient die Nutzung der Nebenläufigkeitsfunktionen in Java der möglichst präzisen Steuerung von Prozessen.

Obwohl in der Beispielsoftware keine tatsächliche Nebenläufigkeit auftritt, sorgen die Abhängigkeiten zu den genutzten JDK- und Bibliotheksklassen dennoch für Probleme. JSweet bietet keine Möglichkeit der automatischen Übersetzung, weshalb für die Portierung der an der prozessbasierten Simulation beteiligten Klassen ein grundlegendes Refactoring notwendig wäre. Um die Funktionalität zu portieren, müssten alle beteiligten Elemente so abgeändert werden, dass eine Unterbrechung und Wiederaufnahme der Ausführung von Prozessen ohne Concurrency-Klassen möglich wäre. Des Weiteren sollte dabei angestrebt werden, dass der derart überarbeitete Code automatisch von JSweet transpiliert werden kann und anschließend semantisch äquivalent in JavaScript ausführbar ist. Dies erfordert eine umfangreiche und aufwendige Reimplementation der prozessbasierten Simulation in DESMO-J, welche weitreichende Auswirkungen auf das Originalprojekt hat. Aus diesem Grund wurde dieser Ansatz im Rahmen der hier durchgeführten Portierung als nicht sinnvoll erachtet. Grundsätzlich ist es allerdings möglich jede prozessbasierte Simulation auch als eventbasierte Implementierung zu realisieren. Start- und Endpunkte, sowie relevante Meilensteine in Prozessen werden dabei als einzelne Ereignisse modelliert. Aufgrund dessen stellt der Ausschluss der prozessbasierten Sicht für Simulationen von der Portierung eine sinnvolle Lösung des Problems dar. Obwohl das prozessorientierte Paradigma somit für die JavaScript-Version verloren geht, steht dennoch weiterhin derselbe Umfang an modellierbaren Experimenten zur Verfügung. Insgesamt

muss jedoch beachtet werden, dass tatsächliche Nebenläufigkeit in Java bei der Portierung zu JavaScript aufgrund der Limitierung der Zielsprache weitreichende Probleme verursachen kann. Je nach Anwendungskontext muss mit erhöhtem manuellen Portierungsaufwand und Performanceverlust bei der Ausführung gerechnet werden. Um Nebenläufigkeit in JavaScript zu ermöglichen kann eine Reimplementation über Web Worker in Erwägung gezogen werden. Eine automatische Konversion ist jedoch mit den in dieser Arbeit betrachteten Werkzeugen nicht möglich.

In der Code-Basis von DESMO-J lassen sich einige Klassen identifizieren, die nur für die Umsetzung der prozessbasierten Simulation notwendig sind. Darunter fallen unter anderem `SimProcess`, `SimStrandFactory` und deren Unterklassen, `CoRoutineModel` und `ProcessQueue`. Da sie für die Implementation eventbasierter Simulationen nicht benötigt werden, können sie von der Portierung ausgeschlossen werden. Da relevante Klassen wie `Experiment` jedoch Abhängigkeiten zu den ausgeschlossenen Elementen haben, müssen diese Abhängigkeiten durch das Entfernen der entsprechenden Aufrufe, Attribute und Methoden aufgelöst werden. Dabei muss aufgrund der teils engen Verzahnung der Paradigmen besonders darauf geachtet werden, die grundsätzliche Ausführbarkeit der eventbasierten Simulation anhand eines Testfalls zu erhalten. Dabei hilft auch das Schreiben von Unit-Tests zur Kontrolle.

Die im Rahmen dieser Arbeit durchgeführte Portierung beschränkt sich aufgrund der oben genannten Herausforderungen auf die Teilfunktionalität „Eventbasierte Simulation“, wobei als Testkriterium für die Teilfunktionalität die Ausführbarkeit des eventbasierten Tutorialbeispiels von DESMO-J (siehe [DES14]) genutzt wird. Insbesondere durch den Ausschluss der prozessbasierten Simulation kann ein hoher Grad an automatischer Übersetzbarkeit erreicht werden. Als Resultat konnte erreicht werden, dass die ebenfalls mit JSweet übersetzte Beispielsimulation „EventsExample“ in der portierten JavaScript-Version ausgeführt werden kann. Der Simulationsbericht wird dabei nicht wie in Java als HTML-Datei abgelegt, sondern im Browserfenster angezeigt.

Die zweite Phase der Portierungsmethode nach Stehle et al. beinhaltet explizit die Erzeugung von Traceability zwischen korrespondierenden Codeartefakten der Ursprungsversion in Java und der portierten Version in JavaScript. Traceability soll die langfristige Wartbarkeit und Evolution der Software unterstützen. Trace Links zwischen funktional übereinstimmenden Typen ermöglichen eine schnellere Lokalisierung von anzupassenden Stellen in einer Version bei durchgeführten Änderungen in der anderen. Dabei kann es Traceability ermöglichen, selbst bei mangelnder Strukturgleichheit der Versionen dennoch korrespondierende Codeelemente ausfindig zu machen. Abbildung 6.3 zeigt schematisch mögliche Trace Links zwischen Codeelementen in der DESMO-J-Klasse `SingleUnitTimeFormatter` und dem daraus übersetzten JavaScript-Pendant. Das Beispiel zeigt Traceability zwischen Objekttyp, Konstruktor und Methoden auf. Im Falle der überladenen Methode `buildTimeString()` treten Trace Links sowohl zur äquivalenten Funktion, als auch zur ebenfalls aufgerufenen Delegationsmethode auf. Dies ist sinnvoll, da Änderungen in der Java-Version, Anpassungen in der korrespondierenden Funktion oder der Delegationsmethode erfordern können.

Wie bereits in Abschnitt 4.4 erwähnt bestehen verschiedene Möglichkeiten Traceability zu erzeugen. Zum einen kann das genutzte Konvertierungswerkzeug um die Generierung von Trace Links bei der Übersetzung erweitert werden. Dies ist in JSweet unter zusätzlichem Implementierungsaufwand grundsätzlich möglich. Zum anderen kann eine nachträgliche Erzeugung von Trace Links

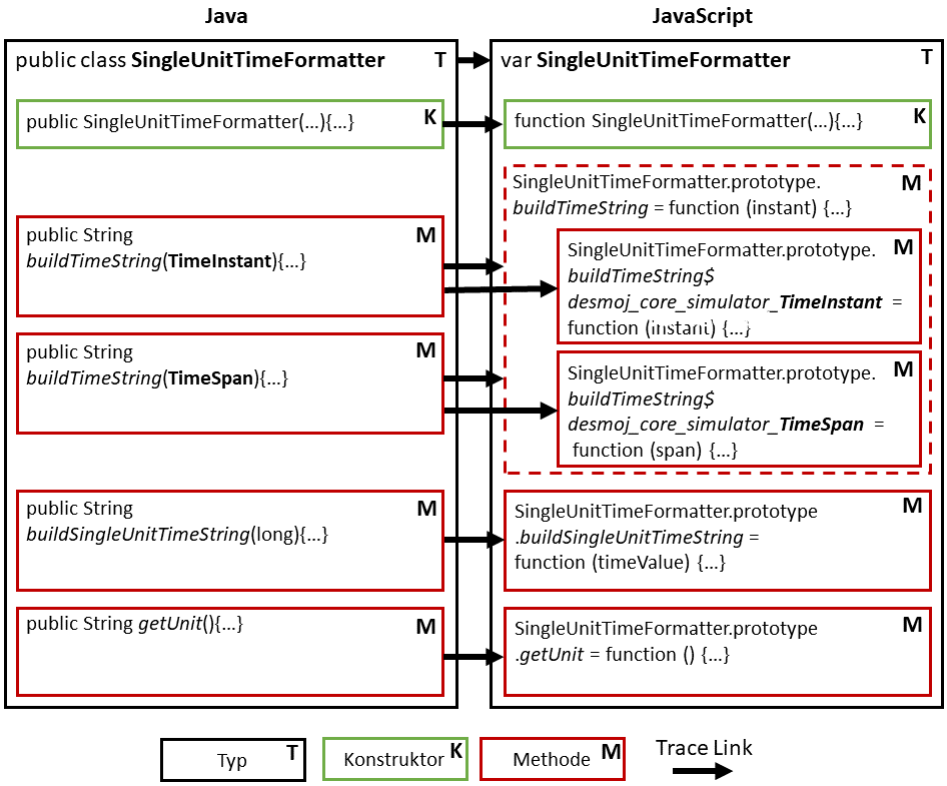


Abbildung 6.3.: Traceability der Klasse `SingleUnitTimeFormatter` zwischen Java und JavaScript

über das vorgestellte Traceability Recovery Tool durchgeführt werden. Die erste Option hat den Vorteil, dass die erzeugten Trace Links definitiv korrekt sind, da korrespondierende Codeelemente direkt bei der Übersetzung verknüpft wurden. Traceability Recovery liefert hingegen lediglich eine Wahrscheinlichkeit für die Übereinstimmung zweier Codeelemente. Allerdings kann Traceability Recovery auch bei einer Weiterentwicklung der Softwareversionen angewandt werden, um Trace Links zu ermitteln. Die bei der Übersetzung erzeugte Traceability verbleibt jedoch ohne manuellem Aktualisierungsaufwand auf dem Stand der ursprünglichen Portierung und bildet durch Weiterentwicklung hinzugekommene oder veränderte Codeelemente nicht ab. Aufgrund der besseren Eignung für eine langfristige Evolution der Softwareversionen wurde für diese Portierung die Nutzung des Traceability Recovery Tools zur Erzeugung der Traceability gewählt. Zudem konnte so ein bestehendes Werkzeug gewählt werden, wodurch der ansonsten nötige Implementierungsaufwand entfällt. Abschnitt 7.3 setzt sich mit der Bewertung der Korrektheit der bestimmten Links auseinander.

### 6.4.2. Manueller Aufwand

Dieser Abschnitt betrachtet den manuellen Aufwand, der für die Portierung der im vorigen Abschnitt erwähnten Teilfunktionalität angefallen ist. Grundsätzlich konnten während des Portierungsprozesses anhand der genutzten Portierungsmethode drei verschiedene Kategorien manuellen Aufwands identifiziert werden:

- Aufwand zur Anpassung und Entwicklung der Werkzeugunterstützung
- Aufwand zur Erzeugung der Portierungsversion
- Aufwand zur Nachbearbeitung des Generats

Die Wahl der Portierungsstrategie aber auch des Konvertierungswerkzeugs hat dabei deutliche Auswirkungen auf das Aufwandsverhältnis zwischen diesen Kategorien. So dient die Erzeugung einer Portierungsversion primär zur Maximierung der automatisch übersetzbaren Anteile, wodurch der Nachbearbeitungsaufwand sinkt. Gleichermaßen kann eine Investition in die Entwicklung zusätzlicher automatischer Werkzeuge später auftretenden, manuellen Aufwand senken. Wie in Abschnitt 6.1 dargelegt, wurde für die hier durchgeführte Portierung die Strategie einer Portierungs mittels Portierungsversion gewählt. Dies ist besonders in Hinblick auf die umfangreiche Konverter-API und Sammlung von Übersetzungs-Makros von JSweet sinnvoll. Durch die Optimierung des Projektcodes für den Konverter in Form einer Portierungsversion lässt sich der automatisch übersetzbare Kern stark maximieren. Für die portierte Teilfunktionalität ergibt sich daher ein sehr geringer Nachbearbeitungsaufwand im JavaScript-Generat. Stattdessen wird der Großteil des manuellen Aufwands vor die automatische Übersetzung zur Erstellung der Portierungsversion verlagert. In Abschnitt D des Anhangs (Seite 111) ist ein Protokoll der durchgeführten manuellen Änderungen zu finden. Der Großteil dieser Änderungen betrifft die Anpassung von Java-Quellcode zur Erhöhung der Kompatibilität mit dem Code-Konverter. Die genutzte Werkzeugumgebung mit der IDE Eclipse und dem JSweet-Plugin gibt dabei über Fehlermeldungen und Warnungen wertvolle Hinweise bezüglich Inkompatibilitäten und Problemstellen. Der portierende Entwickler kann anhand dieser Informationen gezielt nötige Änderungen in Java-Code vornehmen. Bei der reinen Nutzung von JSweet und dem dazugehörigen Plugin, ist der anfallende Aufwand für die Anpassungen der Werkzeugunterstützung aufgrund der niedrigen Konfigurationskomplexität zu vernachlässigen.

Das Vorgehen nach der Portierungsmethode nach Stehle et al. erfordert die manuelle Abspaltung von Teilfunktionalitäten von der Gesamtsoftware. Durch die Aufteilung in kleinere, funktional zusammengehörige Codebasen sinkt die bei der Portierung anfallende Komplexität. Allerdings entsteht zusätzlicher Aufwand bezüglich der Abspaltung und anschließenden Zusammenführung der Teilfunktionalitäten. Dies beinhaltet das in Unterabschnitt 6.3.2 beschriebene Isolieren von Abhängigkeiten und Schreiben von Dummy-Implementationen nicht in der Teilfunktionalität enthaltener Klassen. Insbesondere die Entwicklung der Dummy-Klassen ist in Java-Entwicklungsumgebungen einfach umzusetzen. Durch die in IDEs durchgeführte Überprüfung der Codebasis von Teilfunktionalitäten werden Aufrufe auf fehlende Bestandteile automatisch angezeigt. Aus diesen Fehlermeldungen lassen sich in der Regel mithilfe von integrierten Werkzeugen Dummy-Implementationen erzeugen. Schwieriger gestaltet sich die Auflösung von Abhängigkeiten, wobei vor einem etwaigen Refactoring geprüft werden muss, ob die Abhängigkeit über die Konverter-API umsetzbar ist oder entfernt und später reimplementiert werden muss.

Um die konverterspezifische Portierungsversion zu erzeugen, müssen verschiedene Arten von manuell umzusetzenden Änderungen durchgeführt werden. Eine Kategorie von Änderungen betrifft nicht vom Konvertierungswerkzeug unterstützte Sprachkonstrukte und Syntax. JSweet deckt die Syntax von Java-Code sehr umfangreich ab. Allerdings treten in spezifischen Fällen Probleme bei langen Verkettungen von Methodenaufrufen im Zusammenspiel mit Übersetzungsmakros auf.

So kann ein `newInstance()`-Aufruf an einem Objekt korrekt übersetzt werden. Beim Aufruf von `newInstance()` an einer Methode, die dieses Objekt als Rückgabeparameter liefert treten jedoch für JSweet Probleme auf. Eine Auftrennung der Methodenverkettung in zwei Anweisungen behebt das Problem.

Um Abhängigkeiten zur Sprach-API oder externen Bibliotheken aufzulösen, können, wie bereits erläutert, Reimplementierungen auf Basis der Konverter-API und JSweet-Candies durchgeführt werden. Dabei hängt der anfallende Aufwand stark von der Komplexität der zu ersetzenden Funktionalität ab. Im Portierungsbeispiel wurde die Generierung von Zufallszahlen auf Basis eines Seeds über das RandomJS-Candy umgesetzt, wobei eine alternative Implementation der Klasse `Random` entwickelt wurde. Durch die Kapselung der konverterspezifischen Änderungen in dieser Klasse bleiben die Änderungen im Originalcode minimal. Für die Teilfunktionalität im Beispiel mussten zehn Klassen manuell implementiert werden, um die Kompatibilität mit dem Konverter in der Portierungsversion herzustellen. Diese sind im Paket `def` der Portierungsversion zu finden. Der anfallende Aufwand war dabei eher gering, da oftmals nur einzelne Methoden der zu ersetzenden Klassen reimplementiert werden müssen und zudem vielfach Code aus JDK-Klassen als Basis genutzt werden kann (siehe z.B. `TimeUnit`, `Color`). Um den manuellen Aufwand bei der Nutzung von alternativen Implementierungen zu minimieren, ist zu empfehlen den Bezeichner der ersetzten Klasse beizubehalten. Somit muss lediglich die Importanweisung verändert werden um die Alternativimplementation zu nutzen. Nur in Fällen wo sowohl die alternative, als auch die originale Implementierung genutzt werden, muss aufgrund der gleichen Bezeichner für eine Implementierung der voll qualifizierte Name im Code verwendet werden. Als Beispiel wäre hier eine Klasse die sowohl `java.lang.System.out.println()` als auch `def.System.currentTimeMillis()` nutzt.

Durch die Maximierung der automatischen Übersetzbarkeit mittels der Portierungsversion konnte der nötige Anpassungsaufwand im JavaScript-Generat minimiert werden. Grundsätzlich können drei Aspekte als Gründe für Nachbearbeitungsbedarf erkannt werden:

- Fehler oder Bugs im Konvertierungswerkzeug können für inkorrekt generierten JavaScript-Code sorgen. In der betrachteten Teilfunktionalität tritt dies hauptsächlich bei Aufrufen auf Klassen in JSweet-Candies auf. Dieser Übersetzungsfehler sorgt in den Klassen `def.MersenneTwisterRandomGeneratorJS` und `def.Random` dafür, dass das `Random`-Objekt der RandomJS-Bibliothek nicht aufgerufen werden kann. Aufrufe dieser Klasse müssen manuell korrigiert werden.
- Funktionalitäten, die aufgrund von Inkompatibilitäten mit dem Übersetzungswerkzeug nicht in der Portierungsversion implementiert werden konnten, müssen in JavaScript reimplementiert werden. Im Beispiel betrifft dies unter anderem das Öffnen des Simulationsreports als HTML-Seite im Browser. Diese Funktionalität ist auf diese Weise nicht in der Originalversion enthalten und nur für die portierte Version relevant. Daher ist eine manuelle Implementierung in JavaScript sinnvoll.
- Das Auftreten von Fehlern bei der Ausführung ist zu identifizieren. Diese Fehler können von zuvor unbekannten Übersetzungsfehlern oder semantischen Unterschieden zwischen den beiden Versionen herrühren. Obwohl die semantische Korrektheit des Generats von

JSweet als sehr gut bewertet werden kann, sind problematische Spezialfälle nicht auszuschließen. Debugging-Werkzeuge in gängigen Browsern eignen sich in diesen Fällen für eine Untersuchung des portierten Codes.

Grundsätzlich ist festzustellen, dass bei der durchgeführten Portierung ein Großteil des manuellen Aufwandes bei der Erzeugung der Portierungsversion aufgetreten ist. Die anschließende Nachbearbeitung des Generats beschränkt sich auf geringfügige Korrekturen in drei Klassen, sowie der Implementierung einer neuen Funktion. Die grundlegende Herausforderung bei der gewählten Portierungsstrategie ist die Erzeugung der Portierungsversion aus dem Originalcode, wobei dieser Aufwand bei einer einmaligen Portierung und anschließenden parallelen Evolution beider Versionen nur einmal auftritt.

# 7. Evaluation

Nachdem gezeigt wurde, dass die Portierung von DESMO-J mit den beschriebenen Einschränkungen zu JavaScript durchgeführt werden kann, soll in diesem Kapitel das Vorgehen und die zugrundeliegende Portierungsmethode evaluiert werden. Dabei soll insbesondere die Frage geklärt werden, inwiefern sich das Vorgehen eignet, um eine langfristige, parallele Entwicklung der Softwareversionen im Sinne der Methode nach Stehle et al. durchzuführen. Eine parallele Evolution von Java- und JavaScript-Version ist notwendig, da die Software aktiv entwickelt wird und somit bei Änderungen beide Versionen aktualisiert werden müssen. Eine native Weiterentwicklung kann aufgrund des wegfallenden Aufwands einer erneuten Portierung sinnvoll sein. Zudem ermöglicht eine native Weiterentwicklung eine bessere Anpassung und Ausnutzung auf die Plattform und deren Eigenschaften. Um die Eignung des Outputs der Portierungsmethode für langfristige, parallele Evolution zu evaluieren werden einige Indikatoren untersucht. Dazu gehört zum einen die Strukturgleichheit zwischen Code der Java- und JavaScript-Version. Eine höhere Strukturgleichheit vereinfacht durch besseres Codeverständnis das Umsetzen von Änderungen. Um die Strukturgleichheit zu evaluieren wird Input- und Output-Code miteinander verglichen. Zum anderen dient die Lesbarkeit des generierten Output-Codes als Indikator für die Wartbarkeit der portierten Version. Je einfacher das Generat zu lesen und somit zu verstehen ist, desto einfacher wird eine Weiterentwicklung der Software auf Basis dieses Codes. Als dritter Indikator dient der Beitrag der im Rahmen der Portierungsmethode erzeugten Traceability zwischen Codeartefakten der Java- und JavaScript-Version. Grundsätzlich sollte die Traceability das Codeverständnis über die Strukturgleichheit hinaus unterstützen. Ein zusätzlicher Aspekt der Evaluation betrifft die Richtigkeit der Portierung. Dabei ist die semantische Übereinstimmung der Versionen zu untersuchen. Anhand der verwendeten Beispielsimulation kann geprüft werden, inwiefern sich das Verhalten der portierten Version vom Original unterscheidet und ob diese Unterschiede kritisch sind.

## 7.1. Gegenüberstellung von Input und Output

Um Änderungen an einer Software im Zuge einer parallelen Evolution durchführen zu können, müssen zuerst korrespondierende Stellen in beiden Versionen identifiziert werden. Eine einheitliche Struktur des Codes und der Architektur vereinfacht dabei das Auffinden von Features, an denen Änderungen durchzuführen sind. Möglichst hohe Strukturgleichheit verringert somit den für Codeverständnis zu erbringenden Aufwand. Aus diesem Grund ist zu prüfen, inwiefern die Struktur der ursprünglichen Java-Version in der portierten JavaScript-Version wiederzufinden ist.

Grundsätzlich ist festzustellen, dass der von JSweet erzeugte JavaScript-Code strukturell ähnlich ist. Das Generat weicht dabei nur von der durch den Java-Input vorgegebenen Struktur ab, wenn diese zu nicht semantisch äquivalentem und unausführbarem Code führt. Gerade in Fällen, in denen Java-spezifische Konzepte in JavaScript nachgebildet werden müssen, entsteht oftmals

zusätzlicher Code, der strukturell nicht auf Entsprechungen im Original zurückzuführen ist. In Java genutzte Bezeichner werden unverändert übernommen, sodass eine strukturelle Zuordnung über die Benennung von Attributen und Methoden möglich ist. JSweet erzeugt aus jeder Java-Datei im Input eine korrespondierende JavaScript-Datei im Output. In Fällen wie komplexen Enumerationstypen, in denen in JavaScript Hilfsobjekte generiert werden, um das Verhalten von Enumerationen in Java nachzubilden (vgl. Unterabschnitt 5.2.1), wird der Code für diese Objekte in derselben Datei erzeugt wie die dazugehörige Enumeration. Dies ist sinnvoll, da dieser Hilfscode eindeutig bestimmten Enumerationstypen zuzuordnen ist und nur im Zusammenspiel damit nützlich ist. Für Interfaces generiert der Konverter eine Datei, welche jedoch keinen Code enthält. Dies liegt daran, dass JavaScript keine Interfaces unterstützt und die in Java durch Interfaces definierten Methoden direkt an der implementierenden Klasse definiert werden. Die eindeutige Bindung an das Interface geht somit in JavaScript verloren, was insbesondere bei einer nativen Weiterentwicklung in JavaScript beachtet werden muss. Bei der Umsetzung neuer Implementationen von Interfaces in der portierten Version, sollte somit die Originalversion referenziert werden, um eine möglichst strukturgleiche Evolution zu erreichen.

Die erreichte Strukturgleichheit kann durch eine Gegenüberstellung von Input und Output ermittelt werden. Ab Seite Seite 116 im Anhang wird der Code der Portierungsversion der DESMO-J-Klasse `core.simulator.ExternalEvent` mit dem Generat von JSweet gegenübergestellt. Die Erzeugung der Portierungsversion aus dem Original hat dabei keine relevanten strukturellen Veränderungen verursacht. Da das Konvertierungswerkzeug keinen Gebrauch von den mit ECMAScript 6 eingeführten Klassen in JavaScript macht, werden die Paketstruktur, Klassen und deren Attribute und Methoden im Generat als Variablen realisiert. Dies stellt einen Unterschied zur Klassendefinition in Java dar, allerdings bleibt die logische Schachtelung der einzelnen Elemente unverändert erhalten. Gleichermaßen stimmt die Klassenstruktur mit ihren Methoden und Konstruktoren mit der Struktur in Java überein. Strukturelle Abweichungen lassen sich bei der überladenen Methode `schedule()` feststellen. Um die Überladung in JavaScript zu realisieren generiert der Konverter Funktionen mit angepassten Bezeichnern, die aus einer Delegationsfunktion mit dem ursprünglichen Bezeichner *schedule* aufgerufen werden. Obwohl die Überladungen strukturell den daraus erzeugten Funktionen zugeordnet werden können, wird die Struktur im Generat durch die zusätzliche Delegationsfunktion erweitert. Dies muss bei einer etwaigen Weiterentwicklung bedacht werden, da bei weiteren Überladungen auch Änderungen in der Delegationsmethode durchgeführt werden müssen. Als abstrakte Klasse definiert `ExternalEvent` die abstrakte Methode `eventRoutine()`, die in Java nicht aufrufbar wäre. Eine derartige Definition abstrakter Methoden ist in JavaScript nicht möglich, daher wird eine Funktion mit Hilfsfunktion erzeugt, die mit einer Fehlermeldung auf eine unerlaubte Ausführung hinweist. Auch hier sorgt die Erzeugung zusätzlicher Funktionen für eine Strukturabweichung. Gleiches gilt für die Funktion `_extends`, welche in jeder Klasse die von einer anderen erbt generiert wird.

Strukturell wird bei der Übersetzung mit JSweet eine hohe Gleichheit erreicht. Die bestehende Projektstruktur mit Paketen, Klassenstrukturen und Architekturen wird während der automatischen Übersetzung beibehalten. Die in JavaScript auftretenden Erweiterungen der Struktur durch zusätzlich generierte Codeartefakte trägt nur geringfügig zu einer Erhöhung der Komplexität bei. Dies liegt vor allem an der konsistenten Verwendung von Bezeichnern und durch den Konverter genutzten Prä- und Suffixen wie „_“ oder „$“, die die Zuordnung solcher Erweiterungen

vereinfachen. Schwerwiegendere Unterschiede der Struktur zur Originalversion rühren demnach eher von notwendigen Veränderungen bei der Erzeugung der Portierungsversion her. Dies ist, abgesehen von der Abspaltung von der Portierung ausgeschlossener Softwarefunktionalitäten, bei der durchgeführten Portierung nicht aufgetreten.

## 7.2. Wartbarkeit des Generats

Eine möglichst hohe Wartbarkeit spielt bei der Eignung einer portierten Software für langfristige Evolution eine wichtige Rolle. Wie bereits im Werkzeugvergleich aufgezeigt, generiert eine Vielzahl der Konvertierungswerkzeuge nur schwer wartbaren Code. JSweet wurde auch insbesondere wegen des im Vergleich gut lesbaren Generats ausgewählt. Eine hohe Lesbarkeit des generierten JavaScript dient dabei als Indikator für eine bessere Wartbarkeit aufgrund von leichter zu erreichendem Codeverständnis. Dieser Abschnitt gibt eine Einschätzung der Wartbarkeit des von JSweet erzeugten JavaScript-Codes bezüglich einer nativen Weiterentwicklung.

Natürlich stellt die Strukturgleichheit zwischen Original und portierter Version einen wertvollen Vorteil für das Codeverständnis des Generats dar. Die Kontextinformationen die aus der Kenntnis der Softwarestruktur gezogen werden können, helfen beim Lesen und Verstehen des Codes. Zu diesen Kontextinformationen tragen auch die JavaDoc-Kommentare an Klassen und Methoden bei, die vom Konverter ins Generat übernommen werden. Diese Aspekte der Softwaredokumentation bleiben beim Portierungsprozess erhalten. Dabei werden die Kommentare zu JSDoc übersetzt. Darin werden zur besseren Nachverfolgbarkeit gegenüber dem Original die in Java definierten Typen von Übergabe- und Rückgabeparametern genannt. Dies ist bei einer Wartung hilfreich, da so im JavaScript-Code erkenntlich ist, welche Objekttypen problemlos als Übergabeparameter genutzt werden können. Die aufgrund der dynamischen Typisierung in JavaScript mögliche Übergabe eines beliebigen Typs kann Probleme in automatisch übersetzten Methoden hervorrufen. Insbesondere bei Methodenüberladungen mittels einer Delegationsmethode in JavaScript kann die Delegation bei nicht vorgesehenen Objekten Probleme verursachen. Daher sind derartige Informationen über die Originalversion bei der Weiterentwicklung des Generats sinnvoll.

Durch die Verwendung des Java-Quellcodes als Input, bleibt dessen Kontrollfluss in der Regel unverändert in JavaScript erhalten. Dies ist gut am Codebeispiel ab Seite 116 zu erkennen. Die starke Ähnlichkeit bei der Syntax von Kontrollstrukturen zwischen den Sprachen ist vorteilhaft. Problematisch ist die Lesbarkeit des Generats in der Regel an den Stellen, an denen Sprachkonstrukte aus Java in JavaScript nachgebildet werden. Um dies zu realisieren wird Hilfscode generiert, welcher mit veränderten Bezeichnern arbeitet und gekürzte Variablennamen nutzt. Zudem ist für diesen Code keine Dokumentation über Kommentare vorhanden. Aufgrund dessen sollten Entwickler mit für den Konverter typischen Lösungen von Kompatibilitätsproblemen zwischen den Sprachen vertraut sein, da dies das Verständnis dieses Hilfscodes erleichtert.

Das Portierungsbeispiel umfasst einige Beispiele an denen die Lesbarkeit des Generats eingeschränkt ist. Grundsätzlich werden Typen im Generat als Eigenschaften von Variablen definiert, die die Paketstruktur des Java-Inputs nachbilden. Daraus ergibt sich der Umstand, dass Aufrufe auf bestimmte Typen immer am voll qualifizierten Namen getätigt werden müssen, anstatt den Typbezeichner wie in Java üblich nach einer Importanweisung abzukürzen. Dies sorgt für eine höhere Komplexität im JavaScript-Code. Bei überladenen Methoden werden in JavaScript mehrere

Funktionen mit unterschiedlichen Bezeichnern erzeugt. Diese Methodenbezeichner setzen sich aus dem ursprünglichen Methodennamen und den Typen der Übergabeparameter zusammen. Auch hierbei werden die voll qualifizierten Name verwendet. Daraus ergeben sich bei einer größeren Anzahl an Übergabeparametern sehr lange Methodenbezeichner, die sich je nach Überladung unter Umständen sehr ähnlich sehen. Obwohl diese Methoden in der Übersetzung nur durch die automatisch erzeugte Delegationsmethode aufgerufen werden, wäre bei einer Weiterentwicklung auch ein direkter Aufruf denkbar und sinnvoll. Lange und komplexe Methodenbezeichner sind dabei hinderlich und unpraktikabel. Zudem wird der Code durch diesem Umstand sehr umfangreich, was grundsätzlich nicht zur Verständlichkeit beiträgt.

Ein weiteres Beispiel ist die Nutzung von Enumerationen. Obwohl JSweet den nötigen Code für die Nachbildung des Verhaltens von Java-typischen Enumerationen erzeugt, sind sowohl dieser, als auch Aufrufe auf den Emulationstyp schwer nachzuvollziehen. Besonders die im Rahmen einer Weiterentwicklung mögliche Erweiterung der Enumeration durch weitere Konstanten stellt aufgrund der Codekomplexität eine Herausforderung dar. Statt einer händischen Implementierung in JavaScript, könnte eine Anpassung der Enumeration in der Java-basierten Portierungsversion und eine anschließende automatische Generierung des JavaScript-Codes sinnvoll sein. Auch die Implementation von Interfaces oder abstrakten Klassen ist im JavaScript-Generat schwer nachvollziehbar. Die nötigen Funktionen sind nach der Übersetzung im Objekt oder dessen Prototypen vorhanden, allerdings ist es schwer zu erkennen, ob sie aus etwaigen Oberklassen oder Interfaces stammen. JSweet erzeugt lediglich eine Eigenschaft, die alle in Java implementierten Interfaces eines Typs enthält. Dies dient allerdings nur bei Kenntnis der Originalversion als Orientierung, da das Konzept von Interfaces in JavaScript nicht widergespiegelt werden kann.

Insgesamt kann festgestellt werden, dass JSweet durchaus lesbaren Code erzeugt, die Lesbarkeit jedoch unter den für die korrekte Ausführung nötigen, zusätzlichen Codeelementen leidet. Diese treten immer dann auf, wenn Konzepte aus Java in JavaScript emuliert werden müssen, um semantisch übereinstimmendes Verhalten bei der Ausführung zu erreichen. Dabei wird komplexerer und teilweise schwer leserlicher Code genutzt, welcher jedoch einen grundlegenden Beitrag zur korrekten Ausführung leistet. Das bedeutet, dass dieser Code für eine sinnvolle Weiterentwicklung verstanden und bei Neuerungen angepasst werden muss. An diesen Stellen ist festzustellen, dass eine native Entwicklung in JavaScript aufwändiger ist, als eine Anpassung der Portierungsversion und anschließende, erneute automatische Konvertierung. Dazu muss weiterhin gesagt werden, dass durch den gewählten Ansatz der Übersetzung aus Java Konzepte übertragen werden, die in nativ entwickelten JavaScript-Programmen in der Regel nicht auftreten würden. So wird das Java-Paradigma so gut wie möglich zu JavaScript übertragen, anstatt JavaScript-eigene Sprachkonzepte und -paradigmen zu nutzen. Dabei kann unnötige Komplexität auftreten, die bei einer grundlegenden Reimplementation in JavaScript durch die Wahl einer geeigneten Architektur und Datenstrukturen vermieden werden könnte. Das Generat ist durchaus verständlich und in JavaScript wartbar. Allerdings sollte Kenntnis der Java-Version und der von JSweet genutzten Emulationsmechanismen bei der Weiterentwicklung vorhanden sein, um die Wartbarkeit durch Kontextinformationen zu verbessern. Alternativ zur Wartung des JavaScript-Generats kann die Wartung der portierten Version im intermediären TypeScript durchgeführt werden. Der von JSweet erzeugte TypeScript-Code ist aufgrund der größeren Ähnlichkeit zum Java-Code einfacher lesbar und bildet die Struktur der Software deutlicher ab. In Java durchgeführte Veränderungen lassen sich daher besser auf TypeScript

abbilden als auf den daraus erzeugten JavaScript-Code. Aus diesem Grund ist die parallele Evolution in Java und TypeScript dem Anpassen des JavaScript-Generats vorzuziehen.

## 7.3. Beitrag der Traceability

Wie in Unterabschnitt 6.4.1 erläutert, wurde bei der hier durchgeführten Portierung die Traceability nicht bei der Übersetzung, sondern nachträglich durch die Nutzung des in Abschnitt 4.4 erwähnten Traceability Recovery Tools erzeugt. Die alternative Erzeugung von Trace Links bei der Übersetzung hätte eine Erweiterung des Konvertierungswerkzeugs erfordert. Dabei würden eindeutige Links zwischen einem Codeartefakt und dem daraus erzeugten Generat erzeugt. Im Gegensatz dazu ermittelt das Traceability Recovery Tool über Information Retrieval-Verfahren mögliche Tracelinks anhand von Artefaktbezeichnern. Als Ergebnis für eine Anfrage auf Tracelinks zu einem bestimmten Codeartefakt, liefert das Tool eine nach einem Übereinstimmungswert geordnete Liste von möglichen Tracelinks. Um den Beitrag der so erhaltenen Traceability einschätzen zu können, muss geprüft werden, wie gut die Qualität der gelieferten Ergebnisse ist. Um Ergebnisse von Information Retrieval-Verfahren zu prüfen, eignet sich die Bestimmung der Mean Average Precision (MAP) nach Manning et al. [MRS+08]. MAP wird mithilfe der folgenden Formel bestimmt, wobei $Q$ die Menge der Informationsbedarfe und $m_j$ die Menge an korrespondierenden JavaScript-Artefakten zu einem Java-Artefakt sind.

$$MAP(Q) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{m_j} \sum_{k=1}^{m_j} Precision(R_{jk})$$

Es wir die durchschnittliche Präzision für die Lieferung des korrekten Zielartefakts berechnet, wobei inkorrekte Links die höher als der Korrekte eingestuft wurde, einbezogen werden. In der Literatur wird empfohlen mehr als 50 Informationsbedarfe zu überprüfen, um Information Retrieval-Verfahren sinnvoll evaluieren zu können [MRS+08] [SR17].

Die in dieser Arbeit portierte Teilfunktionalität von DESMO-J enthält 111 Typen, wobei davon 95 Klassen und 16 Interfaces sind. Um die Qualität der Traceability Recovery zu evaluieren wurde das Tool auf diese Typen angewandt. Eine korrekter Trace Link verknüpft eine Klassendefinition in Java mit der korrespondierenden Objektdefinition im portierten JavaScript-Code. Bei der Anwendung konnte festgestellt werden, dass für Interface-Typen keine direkten Trace Links gefunden werden konnten. Stattdessen zeigten alle gefundenen Trace Links auf Objekte, die das betrachtete Interface nutzen oder implementieren. Dies liegt daran, dass Schnittstellen in JavaScript nicht als eigenständige Typen definiert werden können. JSweet erzeugt aus Interface-Definitionen in Java keinen JavaScript-Code, sondern lediglich leere Dateien. Aus diesem Grund ist eine sinnvolle Nachverfolgung von Interfaces über Traceability zwischen Java und JavaScript nicht möglich. Ähnliches gilt für abstrakte Klassen, welche nur abstrakte Methoden und keine eigenen Implementationen enthalten. Dies tritt mit der Klasse `EventList` einmal innerhalb der Teilfunktionalität auf. Das Generat dieser Klasse enthält für eine Zuordnung durch das Information Retrieval-Verfahren nicht genügend Informationen, da die abstrakten Methoden nur in Unterklassen auftreten. Somit verweisen gefundene Trace Links hauptsächlich auf Unterklassen von `EventList`. Für die restlichen 94 regulären und abstrakten Klassen konnte eine $MAP(Q)$ von $0,9796$ ermittelt werden. Die Tabelle ab Seite 122 im Anhang zeigt die

Auswertung der Typen und die zu erreichenden Trace Links. Das ideale Ergebnis wäre die Ausgabe des korrekten Trace Links für alle Anfragen an erster Stelle der Ergebnisliste für eine *MAP*(*Q*) von 1. Eine derart hohe Präzision zeigt, dass die durch das Traceability Recovery Tool erzeugte Nachverfolgbarkeit in diesem Anwendungsfall mit sehr hoher Wahrscheinlichkeit das korrekte, korrespondierende Codeartefakt zu einer Anfrage liefert. Da das Information Retrieval-Verfahren auf dem Vergleich von Bezeichnern beruht, ist eine hohe Präzision bei automatischer Übersetzung unter weitreichender Beibehaltung von ursprünglichen Bezeichnern zu erwarten.

Der Zugriff auf das verwendete Tool und die dadurch erzeugten Traceability-Informationen können durch das vorhandene Eclipse-Plugin in den Arbeitsprozess während der Portierung und einer anschließenden Weiterentwicklung integriert werden. Während der im Rahmen dieser Arbeit durchgeführten Portierung hat die Traceability dazu beigetragen, die Prüfung des Generats zu bestimmten Codeartefakten in Java zu beschleunigen. Zudem kann von Stellen die in der Portierungsversion abgespalten wurden direkt in den entsprechenden JavaScript-Code gesprungen werden, um dort notwendige Nachbearbeitungsarbeiten oder Reimplementierungen vorzunehmen. So kann Traceability zwischen den Versionen nicht nur eine langfristige Softwareevolution, sondern auch bereits den Portierungsprozess unterstützen.

## 7.4. Evaluation der Methodik

In Kapitel 1 wurde die Frage gestellt, inwiefern sich die Portierungsmethode nach Stehle et al. neben der Portierung von Mobilanwendungen auch für nicht mobile Software und Codebibliotheken wie DESMO-J eignet. Dabei ist besonders interessant, ob die Methode hilfreiche Entscheidungsunterstützung während des Portierungsprozesses liefert. Die Methode definiert entlang ihrer drei Phasen ein klares Rahmenwerk für das Vorgehen bei der Portierung. Die darin definierten Schritte konnten, wie in den vorherigen Kapiteln beschrieben, auch bei dem hier durchgeführten Portierungsprojekt angewandt werden. In der Methode wird Portierung nicht nur als automatische Übersetzung mithilfe von Konvertierungstools beschrieben. Darüber hinaus kann auch eine strukturgleiche Reimplementation im Rahmen der Portierung durchgeführt werden. Allerdings kann festgestellt werden, dass die Durchführbarkeit der vorgegebenen Arbeitschritte, insbesondere bei der automatischen Übersetzung stark von der dafür verfügbaren Werkzeugunterstützung abhängt. Die Werkzeug-Analyse in Kapitel 4 hat gezeigt, dass viele Tools nicht den Anforderungen der Portierungsmethode genügen. Auch die übrigen Werkzeuge wie GWT und JSweet sind nicht explizit als Portierungstools implementiert. Stattdessen dienen sie laut Hersteller der Implementierung von Webanwendungen in der Programmiersprache Java und nicht zur Überführung von Java-Code zu JavaScript für den Zweck einer langfristigen, parallelen Evolution. Aus diesem Grund ist eine möglichst weitreichende automatische Übersetzung von Java-Code nur möglich, wenn die vom Konverter angebotene API für JavaScript-Sprachkonzepte genutzt wird.

Aus dieser Tatsache können sich Konflikte zwischen Vorgaben bzw. Empfehlungen der Methode und der gegebenen Werkzeugauswahl ergeben. Die Methode empfiehlt generell eine Maximierung der plattformunabhängigen Kerns, um möglichst große Teile der Codebasis automatisiert übersetzen zu können. Dabei sollen plattformspezifische Abhängigkeiten isoliert werden, da diese oftmals für Instabilität in damit verknüpften Codeelementen sorgen [SR17]. Dies ist grundlegend sinnvoll, allerdings bietet JSweet über seine Konverter-API und Candies alternativ die Möglichkeit, Abhängig-

keiten durch JavaScript-Äquivalente zu ersetzen, anstatt sie aus dem automatisch zu übersetzenden Kern herauszulösen. Beides sorgt für einen Mehraufwand bei der Portierung, entweder für die Isolierung und anschließende Reimplementierung oder für die Realisierung der Abhängigkeit über JSweet-eigene Mechanismen im Java-Code. Während der durchgeführten Portierung konnte festgestellt werden, dass die Auflösung derartiger Abhängigkeiten über die API von JSweet grundsätzlich einfach umzusetzen war und zur Definition eines sehr umfangreichen, automatisch übersetzbaren Kerns führte. Voraussetzung dafür ist natürlich die Kenntnis der Konverter-API, was zumindest bei JSweet durch mögliche Veränderungen der API durch die aktive Entwicklung des Konverters erschwert wird. Die Nutzung der Konverter-API erschwert auch die von der Methode empfohlene Abdeckung aller transformationsvorbereitender Refactorings durch Unit-Tests. Dies soll bei notwendigen Veränderungen des Codes vor der Übersetzung die Ausführbarkeit und semantische Gleichheit sicherstellen. Refactorings, die zur besseren Kompatibilität mit JSweet mithilfe der Konverter-API durchgeführt wurden, sorgen allerdings aufgrund der Funktionsweise der API dafür, dass der betrachtete Code in der Regel nicht mehr in Java ausführbar ist. Somit besteht hier ein Konflikt zwischen der Portierungsmethode und der bestmöglichen Nutzung des Konvertierungswerkzeugs.

Die Portierungsmethode beschreibt eine inkrementelle Portierung einer Software anhand von Teilfunktionalitäten. Jede Teilfunktionalität soll dabei einen eigenständigen Nutzen haben und testbar sein. Obwohl es möglich ist, wie in Abschnitt 6.3 dargestellt, Teilfunktionalitäten im DESMO-J-Projekt zu identifizieren und herauszulösen, stellt die enge Verzahnung von Konzepten im Framework dabei eine Herausforderung dar. Verschiedene Bestandteile von zentralen Klassen wie `Experiment` werden zum Beispiel in so gut wie jeder Teilfunktionalität benötigt. Die Erzeugung einer Instanz zu Testzwecken erfordert allerdings weitere Elemente, die oftmals keinen direkten Bezug zur betrachteten Teilfunktionalität haben. Im Portierungsbeispiel ergab sich aus diesem Umstand teilweise ein erhöhter Aufwand für das Schreiben von Dummy-Implementationen. Im Gegensatz dazu, konnten mithilfe der Werkzeugunterstützung in der Entwicklungsumgebung größere Teilfunktionalitäten, wie die in Unterabschnitt 6.4.1 beschriebene Funktionalität „Eventbasierte Simulation“ problemlos portiert werden. Dieses Codefragment enthält weite Teile der für die ereignisbasierte Simulation in DESMO-J benötigten Klassen. Insbesondere die Fehlermeldungen und Hinweise von JSweet und dem dazugehörigen Plugin helfen bei der Identifikation von Problemstellen und problematischen Abhängigkeiten. Dieser weniger fragmentierte Ansatz erschwert natürlich die Prüfung der korrekten Ausführung und semantischen Gleichheit im Gegensatz zur Betrachtung von weniger umfangreichen Teilfunktionalitäten. Im Gegenzug sinkt der Aufwand für die Abspaltung von Teilfunktionalitäten und der anschließenden Zusammenführung.

Zusammenfassend ist dennoch festzustellen, dass die Portierungsmethode ein hilfreiches Rahmenwerk für die Portierung liefert. Die Arbeitsschritte konnten grundsätzlich von Mobilanwendungen auf das vorliegende Framework DESMO-J übertragen werden. Je nach Funktionsumfang des verfügbaren Konvertierungswerkzeugs kann es sinnvoll sein, das Vorgehen den Bedingungen und Möglichkeiten der Werkzeugunterstützung anzupassen. Die von Stehle et al. vorgeschlagene Methodik ermöglicht eine systematische Portierung von Codebibliotheken, wobei das Vorgehen durch die Erzeugung von Traceability zwischen den Versionen, eine langfristige Softwareevolution ermöglicht.

# 8. Fazit und Ausblick

Die Zielsetzung dieser Arbeit ist die Portierung eines funktionalen Teils des Simulationsframeworks DESMO-J entlang der Portierungsmethode von Stehle et al. Dabei werden Herausforderungen und Probleme während des Portierungsprozesses, sowie der Beitrag der Portierungsmethode zu einer systematischen Portierung und Ermöglichung einer langfristigen, parallelen Softwareevolution der Versionen untersucht. Bezogen auf das in der Portierungsmethode beschriebene Vorgehen wurde eine Werkzeuganalyse durchgeführt, welche Java-zu-JavaScript-Codekonverter anhand von Bewertungskriterien vergleicht. Die Analyse zeigt, dass insbesondere für die angestrebte Weiterentwicklung der portierten Version zusätzliche Anforderungen gestellt werden müssen. So erzeugen viele der untersuchten Codekonverter korrekten und performanten JavaScript-Code, welcher jedoch aufgrund von mangelnder Lesbarkeit und Strukturgleichheit zum Input kaum wartbar ist. Besser wartbare Ergebnisse liefern Übersetzungwerkzeuge, welche auf Basis von Transpilern arbeiten und Java-Quellcode statt -Bytecode als Input nutzen. Das Werkzeug JSweet wurde als besonders geeignet identifiziert, da es neben einem hohen Funktionsumfang und Reifegrad einfach in den Arbeitsprozess zu integrieren ist und aktiv vom Hersteller und der Nutzercommunity unterstützt wird. Aus diesem Grund wurde JSweet als Portierungswerkzeug für die Portierung des Fallbeispiels ausgewählt.

Eine grundlegende Herausforderung bei der Codekonversion sind die Unterschiede zwischen Quell- und Zielsprache. Auch zwischen Java und der Zielsprache JavaScript treten konzeptionelle Differenzen auf, die für eine erfolgreiche Portierung überbrückt werden müssen. Um diesen Arbeitsschritt bei der Portierung von Java zu JavaScript zu vereinfachen wurde ein Sprachvergleich durchgeführt und kritische Sprachunterschiede aufgezeigt. Aus diesen Unterschieden ergibt sich für die Portierbarkeit Anpassungsbedarf, der entweder durch Funktionen des Codekonverters oder manuelle Refactorings abgedeckt werden kann. Ziel der Anpassungen ist die Vermeidung von in Java genutzten Sprachkonstrukten, die nicht in JavaScript abbildbar sind. Weitreichende Teile der grundlegenden Sprachsyntax der beiden Sprachen lassen sich problemlos mit Hilfe des Übersetzungwerkzeugs übertragen. Problematisch sind die Abhängigkeiten zu spezifischen Teilen des JDK und zu externen Codebibliotheken, welche sich nicht eins zu eins auf Entsprechungen in JavaScript abbilden lassen. Beispiele dafür sind Funktionen wie Java Reflections, Nebenläufigkeit und grafische Benutzeroberflächen, da deren Abbildung deutlichen manuellen Aufwand hervorruft oder aufgrund von Sprachlimitationen in JavaScript nicht möglich ist. In solchen Fällen ist über einen Ausschluss dieser Elemente von der Portierung oder eine Reimplementation über die Konverter-API oder nativ in JavaScript zu entscheiden.

Den Arbeitsschritten der Portierungsmethode nach Stehle et al. folgend konnte die Portierung eines funktionalen Teils des Simulationsframeworks DESMO-J erreicht werden. Der portierte Code ermöglicht die Implementation und Ausführung von Simulationen auf Basis des ereignisorientierten Paradigmas. Das Fallbeispiel enthält eine Beispielsimulation, welche im Browser ausgeführt werden

kann und dabei semantisch mit der Ursprungsversion in Java übereinstimmt. Eine Weiterentwicklung des portierten Codes ist aufgrund der von JSweet erreichten Lesbarkeit und Strukturgleichheit des Generats möglich. Die im Rahmen der Portierungsmethode über das Traceability Recovery Plugin erzeugte Treaceability zwischen übereinstimmenden Codeartefakten zwischen den Versionen unterstützt dabei die parallele Entwicklung. Ein Problem bei der Portierung weiterer Teile von DESMO-J stellt insbesondere die Nutzung von Nebenläufigkeitsfunktionen des JDK zur Umsetzung von prozessorientierten Simulationen dar, weshalb dieser Aspekt im Fallbeispiel von der Portierung ausgeschlossen wurde. Mögliche Lösungen sind die Reimplemetierung von Prozesslebenszyklen mithilfe von anderen, in JavaScript abbildbaren Mechanismen oder die generelle Umsetzung von Prozessen als eine Folge von Ereignissen.

In Hinblick auf die langfristige Weiterentwicklung der Software für beide Plattformen wäre ausblickend zu klären, welcher Ansatz gewählt werden soll. Möglich wäre bei Änderungen in der Java-Version sowohl die erneute Portierung, als auch die native Umsetzung der Änderungen in JavaScript oder TypeScript. Um eine möglichst hohe automatische Übersetzbarkeit zu erreichen, setzt die erste Option die Erzeugung einer Portierungsversion voraus. So wird der manuelle Aufwand größtenteils auf die Java-Seite verschoben, wohingegen die zweite Option manuellen Aufwand nach der Übersetzung im Generat mit sich bringt. Bei einer Weiterentwicklung des Generats ist in Erwägung zu ziehen, das von JSweet erzeugte intermediäre TypeScript statt des JavaScript-Generats zu nutzen. TypeScript ermöglicht im Gegensatz zum von JSweet erzeugten JavaScript, die Definition von Klassen und Interfaces, wodurch eine Abbildung von Java-Konzepten vereinfacht wird. Noch einfacher ist jedoch oftmals die Übertragung von Änderungen der Java-Version in die ebenfalls in Java vorliegende Portierungsversion. Für eine möglichst effiziente Portierung von Änderungen ist daher eine erneute Übersetzung mit einer angepassten Portierungsversion der nativen Implementierung vorzuziehen. Allerdings können nur durch eine native Umsetzung in JavaScript oder TypeScript besondere Eigenschaften und Vorteile der Zielplattform genutzt werden. Eine Übersetzung aus Java führt bei den genutzten Werkzeugen immer zu einer Umsetzung von plattformuntypischen Java-Paradigmen in JavaScript.

Die hier durchgeführte Portierung basiert grundlegend auf der Erzeugung einer Portierungsversion, welche die Ursprungsversion von DESMO-J gegenüber den Anforderungen des Konverters JSweet optimiert, um die automatische Übersetzbarkeit zu maximieren. Über diese Arbeit hinaus wäre eine Untersuchung der Automatisierbarkeit dieses Prozesses sinnvoll, um den manuellen Aufwand zu senken. Je weiter der manuelle Aufwand zur Erzeugung einer Portierungsversion sinkt, desto eher ist eine erneute Portierung einer nativen Implementierung in der Zielsprache vorzuziehen. Eine Minimierung des Aufwands wäre durch mögliche Automatisierung zu erreichen, wobei der dadurch eingesparte Aufwand gegen den Implementationsaufwand für die spezifische, zu entwickelnde Werkzeugunterstützung aufzuwiegen ist.

Ein weiterer Aspekt wäre die Nutzung des aufgezeigten Vorgehens als Lernhilfe für TypeScript und JavaScript. Eine hohe Strukturgleichheit und zusätzliche Unterstützung durch Traceability vereinfachen das Codeverständnis eines vorliegenden Codes auf Basis einer bekannten Java-Implementation. Die Gegenüberstellung der Versionen legt dem Betrachter Parallelen zwischen der Syntax der Sprachen und deren Funktionsweise dar.

# Literaturverzeichnis


[ABF+17] Andersen, Jakob ; Berger, Nils-Hendrik ; Fischer, Evelyn ; Greiert, Gerrit ; Jährling, Claas ; Khanji, Fares ; Schulz, Maike: *Mobile Anwendungen für mehrere Plattformen – Portierung, Architektur- und Tool-Entwicklung*. April 2017. – Universität Hamburg SWK Masterprojekt 2016/17

[BH06] Bishop, Judith ; Horspool, Nigel: Cross-platform development: Software that lasts. In: *Computer* 39 (2006), Nr. 10, S. 26–35

[Cyb96] Cybulski, Jacob L.: Introduction to software reuse. In: *The University of Melbourne, Department of Information Systems, Melbourne, Research Report* 96 (1996), Nr. 4

[DES14] DESMO-J developers team: *DESMO-J Tutorial - Events Step by Step*. `http://desmoj.sourceforge.net/tutorial/events/0.html`, 2014. – zuletzt abgerufen am 19.03.2018

[Dra17] Dragome: *Dragome Web SDK - Takes your Java code to the web*. `http://www.dragome.com/`, 2017. – zuletzt abgerufen am 07.11.2017

[Ecl18a] Eclipse Foundation: *Eclipse Desktop IDEs*. `https://www.eclipse.org/ide/`, 2018. – zuletzt abgerufen am 31.01.2018

[Ecl18b] Eclipse Foundation: *Eclipse Marketplace: Eclipse Web Developer Tools*. `https://marketplace.eclipse.org/content/eclipse-web-developer-tools-0#group-details`, 2018. – zuletzt abgerufen am 31.01.2018

[ECM17] ECMA European Computer Manufacturer's Association: *ECMAScript® 2017 Language Specification*. `https://www.ecma-international.org/ecma-262/8.0/`, 2017. – zuletzt abgerufen am 22.11.2017

[EF15] Evans, Benjamin J. ; Flanagan, David: *Java in a Nutshell - A desktop quick reference*. O'Reilly Media Inc., 2015

[EKAYW14] El-Kassas, Wafaa S. ; Abdullah, Bassem A. ; Yousef, Ahmed H. ; Wahba, Ayman: ICPMD: integrated cross-platform mobile development solution. In: *Computer Engineering & Systems (ICCES), 2014 9th International Conference on* IEEE, 2014, S. 307–317

[EKAYW16] El-Kassas, Wafaa S. ; Abdullah, Bassem A. ; Yousef, Ahmed H. ; Wahba, Ayman M.: Enhanced Code Conversion Approach for the Integrated Cross-Platform Mobile Development (ICPMD). In: *IEEE Transactions on Software Engineering* 42 (2016), Nr. 11, S. 1036–1053

[EKAYW17] El-Kassas, Wafaa S. ; Abdullah, Bassem A. ; Yousef, Ahmed H. ; Wahba, Ayman M.: Taxonomy of cross-platform mobile applications development approaches. In: *Ain Shams Engineering Journal* 8 (2017), Nr. 2, S. 163–190

[Fla11] Flanagan, David: *JavaScript - The Definitive Guide*. O'Reilly Media Inc., 2011

[FS16] Fischer, Evelyn ; Schulz, Maike: *Portierungsmethoden für Apps*. Juni 2016. – Universität Hamburg SWK Masterprojekt 2016/17

[GCHH+12] Gotel, Orlena ; Cleland-Huang, Jane ; Hayes, Jane H. ; Zisman, Andrea ; Egyed, Alexander ; Grünbacher, Paul ; Dekhtyar, Alex ; Antoniol, Giuliano ; Maletic, Jonathan ; Mäder, Patrick: Traceability fundamentals. In: *Software and Systems Traceability*. Springer, 2012, S. 3–22

[GJKP13] Göbel, Johannes ; Joschko, Philip ; Koors, Arne ; Page, Bernd: The Discrete Event Simulation Framework DESMO-J: Review, Comparison To Other Frameworks And Latest Development. In: *ECMS*, 2013, S. 100–109

[Goo17a] Google: *GWT - Examples and real world projects*. `http://www.gwtproject.org/examples.html`, 2017. – zuletzt abgerufen am 08.12.2017

[Goo17b] Google: *GWT - Productivity for developers, performance for users*. `http://www.gwtproject.org/`, 2017. – zuletzt abgerufen am 07.11.2017

[Goo17c] Google: *GWT Compatibility - Language support*. `http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsCompatibility.html`, 2017. – zuletzt abgerufen am 10.12.2017

[Goo18] Google: *libphonenumber - Google's common library for parsing, formatting, and validating international phone numbers*. `https://github.com/googlei18n/libphonenumber`, 2018. – zuletzt abgerufen am 25.01.2018

[Gre17] Greiert, Gerrit: *Entwicklung eines Plugins in IntelliJ IDEA zum Auffinden von Quellcode-Entsprechungen*. August 2017. – Universität Hamburg Masterstudie 2016/17

[GU10] Guermeur, Daniel ; Unruh, Amy: *Google App Engine Java and GWT Application Development*. Packt Publishing Ltd, 2010

[Hav14] Haverbeke, Marijn: *Eloquent javascript: A modern introduction to programming*. No Starch Press, 2014

[HHM12] Heitkötter, Henning ; Hanschke, Sebastian ; Majchrzak, Tim A.: Evaluating cross-platform development approaches for mobile applications. In: *International Conference on Web Information Systems and Technologies* Springer, 2012, S. 120–138

[ISO11] ISO/IEC: *ISO-IEC 25010: 2011 Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models*. International Organization for Standardization, 2011

[Jet16a] JetBrains: *GWT support in IntelliJ IDEA*. https://www.jetbrains.com/help/idea/2016.2/gwt.html, 2016. – zuletzt abgerufen am 09.12.2017

[Jet16b] JetBrains: *Kotlin to JavaScript Tutorial*. https://kotlinlang.org/docs/tutorials/javascript/kotlin-to-javascript/kotlin-to-javascript.html, 2016. – zuletzt abgerufen am 09.12.2017

[Jet17] JetBrains: *Kotlin - Statically typed programming language for modern multiplatform applications*. https://kotlinlang.org/, 2017. – zuletzt abgerufen am 09.12.2017

[JSw17] JSweet: *JSweet - A transpiler from Java to TypeScript/JavaScript*. http://www.jsweet.org/, 2017. – zuletzt abgerufen am 07.11.2017

[LSWM15] Leopoldseder, David ; Stadler, Lukas ; Wimmer, Christian ; Mössenböck, Hanspeter: Java-to-JavaScript translation via structured control flow reconstruction of compiler IR. In: *ACM SIGPLAN Notices* Bd. 51 ACM, 2015, S. 91–103

[Mic17] Microsoft: *TypeScript*. https://github.com/Microsoft/TypeScript/, 2017. – zuletzt abgerufen am 09.12.2017

[MRS+08] Manning, Christopher D. ; Raghavan, Prabhakar ; Schütze, Hinrich u. a.: *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008

[Ora17] Oracle: *Java™ Platform, Standard Edition 8 API Specification*. https://docs.oracle.com/javase/8/docs/api/, 2017. – zuletzt abgerufen am 22.11.2017

[OT12] Ohrt, Julian ; Turau, Volker: Cross-platform development tools for smartphone applications. In: *Computer* 45 (2012), Nr. 9, S. 72–79

[Par18] Parallel Universe: *Quasar - lightweight threads and actors for the JVM*. http://www.paralleluniverse.co/quasar/, 2018. – zuletzt abgerufen am 20.02.2018

[Paw15a] Pawlak, Renaud: JSweet: how does it compare to TypeScript? - A focus on typing and semantics. (2015)

[Paw15b] Pawlak, Renaud: JSweet: insights on motivations and design - A transpiler from Java to JavaScript. (2015)

[Paw16a] Pawlak, Renaud: *JSweet Candies Releases*. http://www.jsweet.org/candies-releases/, 2016. – zuletzt abgerufen am 16.02.2017

[Paw16b] Pawlak, Renaud: *JSweet Language Specifications*. https://github.com/cincheo/jsweet/blob/v1.2.0/doc/jsweet-language-specifications.md#classes, 2016. – zuletzt abgerufen am 08.12.2017

[PDL13] Perchat, Joachim ; Desertot, Mikael ; Lecomte, Sylvain: Component based framework to create mobile cross-platform applications. In: *Procedia Computer Science* 19 (2013), S. 1004–1011

[Pud10] Puder, Arno: Cross-compiling Android applications to the iPhone. In: *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java* ACM, 2010, S. 69–77

[PWZ13] Puder, Arno ; Woeltjen, Victor ; Zakai, Alon: Cross-compiling Java to JavaScript via tool-chaining. In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* ACM, 2013, S. 25–34

[Rim13] Rimmer, Jonathan: *Exploring JavaScript prototypes via TypeScript's class pattern*. `http://blog.brillskills.com/2013/09/exploring-javascript-prototypes-via-typescripts-class-pattern/`, 2013. – zuletzt abgerufen am 01.03.2018

[RS12] Ribeiro, André ; Silva, Alberto R.: Survey on cross-platforms and languages for mobile apps. In: *Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the* IEEE, 2012, S. 255–260

[RT12] Raj, CP R. ; Tolety, Seshu B.: A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In: *India Conference (INDICON), 2012 Annual IEEE* IEEE, 2012, S. 625–629

[SR17] Stehle, T. ; Riebisch, M.: *A Porting Method for Coordinated Multi-Platform Evolution*. 2017. – Preprint

[Tea17] TeaVM: *TeaVM - Java bytecode to JavaScript translator*. `http://teavm.org/`, 2017. – zuletzt abgerufen am 07.11.2017

[Ter01] Terekhov, Andrey A.: Automating language conversion: a case study. In: *Software Maintenance, 2001. Proceedings. IEEE International Conference on* IEEE, 2001, S. 654–658

[TGK+04] Tomer, Amir ; Goldin, Leah ; Kuflik, Tsvi ; Kimchi, Esther ; Schach, Stephen R.: Evaluating software reuse alternatives: a model and its application to an industrial case study. In: *IEEE Transactions on Software Engineering* 30 (2004), Nr. 9, S. 601–612

[THET13] Tacy, Adam ; Hanson, Robert ; Essington, Jason ; Tokke, Anna: *GWT in Action*. Manning Publications Co., 2013

[Tul17] Tulach, Jaroslav: *Bck2Brwsr*. `http://wiki.apidesign.org/wiki/Bck2Brwsr`, 2017. – zuletzt abgerufen am 10.12.2017

[TV00] Terekhov, Andrey A. ; Verhoef, Chris: The realities of language conversions. In: *IEEE Software* 17 (2000), Nr. 6, S. 111–124

[Uda16] Udalov, Alexander: *Issue - Java 8 interoperability*. `https://youtrack.jetbrains.com/issue/KT-4778#tab=Linked%20Issues`, 2016. – zuletzt abgerufen am 09.12.2017

[Ull15] Ullenboom, Christian: *Java ist auch eine Insel: Einführung, Ausbildung, Praxis*. Galileo Press, 2015

[Uni15] Universität Hamburg Arbeitsbereich Modellbildung und Simulation: *Desmo-J: A Framework for Discrete-Event Modelling and Simulation*. `http://desmoj.sourceforge.net/`, 2015. – zuletzt abgerufen am 17.01.2018

[Wag14] Wagner, Christian: *Model-Driven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems*. Springer Verlag, 2014

[XML11] XMLVM: *XMLVM*. `http://xmlvm.org/overview/`, 2011. – zuletzt abgerufen am 10.12.2017

# Anhang

# A. Sprachvergleich und Änderungsbedarf

## A.1. Vererbung

```java
//inheritance.Item.java
package inheritance;

public class Item {

  private long id;
  private String itemName;

  public Item(String name, long id) {
    this.itemName = name;
    this.id = id;
  }

  public long getId() {
    return id;
  }

  public String getItemName() {
    return itemName;
  }
}

//inheritance.Book.java
package inheritance;

public class Book extends Item{

  private String author;

  public Book(String name, long id, String author) {
    super(name, id);
    this.author = author;
  }

  public String getAuthor() {
    return author;
  }
}
```

Quelltext 1: Vererbung in Java

```javascript
//inheritance.Item.js
/* Generated from Java with JSweet 2.0.0-rc1 - http://www.jsweet.org */
var inheritance;
(function (inheritance) {
    var Item = (function () {
        function Item(name, id) {
            this.id = 0;
            this.itemName = null;
            this.itemName = name;
```

```
            this.id = id;
        }
        Item.prototype.getId = function () {
            return this.id;
        };
        Item.prototype.getItemName = function () {
            return this.itemName;
        };
        return Item;
    }());
    inheritance.Item = Item;
    Item["__class"] = "inheritance.Item";
})(inheritance || (inheritance = {}));


//inheritance.Book.js
var __extends = (this && this.__extends) || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
};
/* Generated from Java with JSweet 2.0.0-rc1 - http://www.jsweet.org */
var inheritance;
(function (inheritance) {
    var Book = (function (_super) {
        __extends(Book, _super);
        function Book(name, id, author) {
            var _this = _super.call(this, name, id) || this;
            _this.author = null;
            _this.author = author;
            return _this;
        }
        Book.prototype.getAuthor = function () {
            return this.author;
        };
        return Book;
    }(inheritance.Item));
    inheritance.Book = Book;
    Book["__class"] = "inheritance.Book";
})(inheritance || (inheritance = {}));
```

Quelltext 2: Übersetzung der Vererbung mit JSweet

## A.2. Enumerationen

```
public enum ScreenRatio {

  FREE_RATIO(null),
    RATIO_4_3(4f / 3),
    RATIO_3_2(1.5f),
    RATIO_16_9(16f / 9),
    RATIO_2_1(2f / 1f),
    SQUARE_RATIO(1f);
```

```
9
10      private final Float value;
11
12      private ScreenRatio(Float value) {
13          this.value = value;
14      }
15
16      public Float getValue() {
17          return value;
18      }
19  }
```

Quelltext 3: Eine Enumeration in Java

```
1  /* Generated from Java with JSweet 2.0.0-rc1 - http://www.jsweet.org */
2  var ScreenRatio;
3  (function (ScreenRatio) {
4      ScreenRatio[ScreenRatio["FREE_RATIO"] = 0] = "FREE_RATIO";
5      ScreenRatio[ScreenRatio["RATIO_4_3"] = 1] = "RATIO_4_3";
6      ScreenRatio[ScreenRatio["RATIO_3_2"] = 2] = "RATIO_3_2";
7      ScreenRatio[ScreenRatio["RATIO_16_9"] = 3] = "RATIO_16_9";
8      ScreenRatio[ScreenRatio["RATIO_2_1"] = 4] = "RATIO_2_1";
9      ScreenRatio[ScreenRatio["SQUARE_RATIO"] = 5] = "SQUARE_RATIO";
10 })(ScreenRatio = enums.ScreenRatio || (enums.ScreenRatio = {}));
11
12 /** @ignore */
13 var ScreenRatio_$WRAPPER = (function () {
14     function ScreenRatio_$WRAPPER(_$ordinal, _$name, value) {
15         this._$ordinal = _$ordinal;
16         this._$name = _$name;
17         this.value = null;
18         this.value = value;
19     }
20     ScreenRatio_$WRAPPER.prototype.getValue = function () {
21         return this.value;
22     };
23     ScreenRatio_$WRAPPER.prototype.name = function () { return this._$name; };
24     ScreenRatio_$WRAPPER.prototype.ordinal = function () { return this._$ordinal; };
25     return ScreenRatio_$WRAPPER;
26 }());
27
28 enums.ScreenRatio_$WRAPPER = ScreenRatio_$WRAPPER;
29 ScreenRatio["__class"] = "enums.ScreenRatio";
30 ScreenRatio["__interfaces"] = ["java.lang.Comparable", "java.io.Serializable"];
31 ScreenRatio["_$wrappers"] = [new ScreenRatio_$WRAPPER(0, "FREE_RATIO", null),
32                              new ScreenRatio_$WRAPPER(1, "RATIO_4_3", 4.0 / 3),
33                              new ScreenRatio_$WRAPPER(2, "RATIO_3_2", 1.5),
34                              new ScreenRatio_$WRAPPER(3, "RATIO_16_9", 16.0 / 9),
35                              new ScreenRatio_$WRAPPER(4, "RATIO_2_1", 2.0 / 1.0),
36                              new ScreenRatio_$WRAPPER(5, "SQUARE_RATIO", 1.0)];
```

Quelltext 4: Übersetzung der Enumeration mit JSweet

## A.3. Kontrollstrukturen

```java
int a = 5;
int b = 9;
int[] array = {1, 2, 3, 5, 8};

// If
if(a == b)
  System.out.println("Equal");

// If-else
if (a > b) {
  System.out.println("A");
} else {
  System.out.println("B");
}

//Switch
switch (a) {
  case 5:
    System.out.println("5");
    break;
  case 10:
    System.out.println("9");
    break;
  default:
    System.out.println("Default");
    break;
}

//While
while (a < 10) {
  a++;
}

//Do-While
do {
  a--;
} while (a > 5);

//For
for (int i = 0; i < array.length; i++) {
  System.out.println(array[i]);
}

//For-each
for (int i : array) {
  System.out.println(array[i]);
}
```

Quelltext 5: Kontrollstrukturen in Java

```javascript
var a = 5;
var b = 9;
```

```
var array = [1, 2, 3, 5, 8];

// If
if (a === b)
    console.info("Equal");

// If-else
if (a > b) {
    console.info("A");
} else {
    console.info("B");
}

//Switch
switch ((a)) {
    case 5:
        console.info("5");
        break;
    case 10:
        console.info("9");
        break;
    default:
        console.info("Default");
        break;
}

//While
while ((a < 10)) {
    a++;
};

//Do-While
do {
    a--;
} while ((a > 5));

//For
for (var i = 0; i < array.length; i++) {
    console.info(array[i]);
};

//For-each
for (var index126 = 0; index126 < array.length; index126++) {
    var i = array[index126];
    {
        console.info(array[i]);
    }
}
```

Quelltext 6: Übersetzung der Kontrollstrukturen mit JSweet

## A.4. Fehlerbehandlung

```java
//Class Exceptions
public void tryMethod() {

  try {
    throwException();
    throwFileException();
    throwCustomException();
  } catch (CustomException e) {
    System.out.println("Custom");
  } catch (Exception e) {
    System.out.println(e.getMessage());
  } finally {
    System.out.println("Finally");
  }
}

private void throwException() throws Exception {
  throw new Exception("Message");
}

private void throwFileException() throws FileNotFoundException  {
  throw new FileNotFoundException("Message");
}

private void throwCustomException() throws CustomException  {
  throw new CustomException("Message"); //Benutzerdefinierte Exception
}
```

Quelltext 7: Fehlerbehandlung in Java

```javascript
Exceptions.tryMethod = function () {
  try {
    Exceptions.throwException();
    Exceptions.throwFileException();
    Exceptions.throwCustomException();
  }
  catch (__e) {
    if (__e != null && __e instanceof exceptions.CustomException) {
      var e = __e;
      console.info("Custom");
    }
    if (__e != null && (__e["__classes"] && __e["__classes"].indexOf("java.lang.Exception"
    ) >= 0) || __e != null && __e instanceof Error) {
      var e = __e;
      console.info(e.message);
    }
  }
  finally {
    console.info("Finally");
  };
};

```

```
22 /*private*/ Exceptions.throwException = function () {
23   throw Object.defineProperty(new Error("Message"), '__classes', { configurable: true,
       value: ['java.lang.Throwable', 'java.lang.Object', 'java.lang.Exception'] });
24 };
25 /*private*/ Exceptions.throwFileException = function () {
26   throw Object.defineProperty(new Error("Message"), '__classes', { configurable: true,
       value: ['java.lang.Throwable', 'java.io.IOException', 'java.lang.Object', 'java.io.
       FileNotFoundException', 'java.lang.Exception'] });
27 };
28 /*private*/ Exceptions.throwCustomException = function () {
29   throw new exceptions.CustomException("Message"); //Benutzerdefinierte Exception
30 };
```

Quelltext 8: Übersetzung der Fehlerbehandlung mit JSweet

# B. Abhängigkeitsanalyse

## B.1. Genutzte Klassen des JDK nach Paket

**java.awt**

BorderLayout, Color, Container, Dimension, Frame, Toolkit, Window

**java.awt.event**

ActionEvent, ActionListener, WindowAdapter, WindowEvent

**java.beans**

BeanDescriptor, BeanInfo, EventSetDescriptor, FeatureDescriptor, IntrospectionException, Introspector, MethodDescriptor, ParameterDescriptor, PropertyChangeEvent, PropertyChangeListener, PropertyChangeSupport, PropertyDescriptor, SimpleBeanInfo

**java.io**

BufferedWriter, File, FileWriter, IOException, PrintStream, Writer

**java.lang**

Boolean, Byte, Character, Class, ClassNotFoundException, CloneNotSupportedException, Comparable, Deprecated, Double, Enum, Exception, Float, IllegalAccessException, InstantiationException, Integer, InterruptedException, Iterable, Long, Math, NullPointerException, Number, Object, Override, Runnable, RuntimeException, Short, String, StringBuffer, StringBuilder, SuppressWarnings, System, Thread, ThreadGroup, Throwable, UnsupportedOperationException, Void

**java.lang.reflect**

AccessibleObject, Constructor, Method, Modifier

**java.text**

Collator, DateFormat, DecimalFormat, NumberFormat, SimpleDateFormat

**java.util**

AbstractCollection, AbstractList, AbstractMap, AbstractSet, ArrayList, Arrays, Calendar, Collection, Collections, Date, Enumeration, EnumMap, EnumSet, EventListener, EventObject, GregorianCalendar, HashMap, HashSet, Hashtable, Iterator, LinkedList, List, Locale, Map, Observable, Observer, Properties, Queue, Random, Set, TimeZone, TreeMap, Vector

**java.util.concurrent**

BlockingQueue, ExecutionException, Executor, LinkedBlockingQueue, TimeUnit,

**java.util.concurrent.locks**

Condition, ReentrantLock

**javax.swing**

BorderFactory, JComponent, JFrame, JPanel, JProgressBar, Timer

**javax.swing.border**

Border

## B.2. Genutzte Klassen aus externen Bibliotheken nach Paket

**co.paralleluniverse.common.util**

SameThreadExecutor

**co.paralleluniverse.fibers**

Fiber, FiberExecutorScheduler, SuspendExecution

**co.paralleluniverse.strands**

Strand, SuspendableRunnable

**co.paralleluniverse.strands.concurrent**

ReentrantLock

**org.apache.commons.collections.list**

TreeList

**org.apache.commons.collections.map**

AbstractReferenceMap, ReferenceMap

**org.apache.commons.math**

MathException

**org.apache.commons.math.analysis**

BisectionSolver, UnivariateRealFunction, UnivariateRealSolver

**org.apache.commons.math.distribution**

AbstractIntegerDistribution, BinomialDistribution, BinomialDistributionImpl, ChiSquaredDistributionImpl, ContinuousDistribution, GammaDistribution, GammaDistributionImpl, HypergeometricDistribution, HypergeometricDistributionImpl, IntegerDistribution, NormalDistributionImpl, PoissonDistributionImpl, TDistribution, TDistributionImpl

# C. Teilfunktionalitäten

## C.1. Abspalten der Teilfunktionalität „kontinuierliche Verteilungen“

**An der Teilfunktionalität direkt beteiligte Klassen:**

**core.dist**: ContDist, ContDistAggregate, ContDistBeta, ContDistConstant, ContDistCustom, ContDistEmpirical, ContDistErlang, ContDistExponential, ContDistGamma, ContDistNormal, ContDistTriangular, ContDistUniform, ContDistWeibull, Distribution, DistributionManager, Function, LinearCongruentialRandomGenerator, MersenneTwisterRandomGenerator, NumericalDist, Operator, UniformRandomGenerator

**Vorhandene Abhängigkeiten zu anderen Klassen des Frameworks:**

**core.exception**: DESMOJException, SimAbortedException

**core.report**: ContDistAggregateReporter, ContDistBetaReporter, ContDistConstantReporter, ContDistCustomReporter, ContDistEmpiricalReporter, ContDistErlangReporter, ContDistExponReporter, ContDistGammaReporter, ContDistNormalReporter, ContDistTriangularReporter, ContDistUniformReporter, DistributionReporter, ErrorMessage, Message, Reporter

**core.simulator**: Experiment, Model, ModelComponent, NamedObject, Reportable, SingleUnitTimeFormatter, TimeFormatter, TimeInstant, TimeOperations, TimeSpan

**core.statistics**: StatisticObject

## C.2. Nutzung von JSweet-Candies

```
import java.util.Random;

public class MersenneTwisterRandomGenerator implements desmoj.core.dist.
    UniformRandomGenerator {

  protected Random javaAPIRandomGenerator; // the random generator used
  protected int[] mersenneTwister;
  protected int currentIndex;

  public MersenneTwisterRandomGenerator() {
    this(42);
  }

  public MersenneTwisterRandomGenerator(long seed) {

    mersenneTwister = new int[624];
    javaAPIRandomGenerator = new Random(42);
    this.setSeed(seed);
  }

  public double nextDouble() {

    return (((long) nextInt(26) << 27) + nextInt(27)) / (double) (1L << 53);
  }

  public long nextLong() {
      return ((long) (nextInt(32)) << 32) + nextInt(32);
  }

  public int nextInt(int bits) {

    int y = this.mersenneTwister[currentIndex];
    currentIndex++;

    // re-determine random numbers if pool exhausted
    if (currentIndex > 623)
      this.twistNumbers();

```

```
38      // determine tempered random number
39      y = y ^ y >>> 11;
40      y = y ^ (y << 7) & 0x9d2c5680;
41      y = y ^ (y << 15) & 0xefc60000;
42      y = y ^ y >>> 18;
43
44      // return number (bit length as desired)
45      return y >>> 32 - bits;
46    }
47
48    public void twistNumbers() {
49
50      for (int i = 0; i < 624; i++) {
51        int y = mersenneTwister[i] & 0x80000000
52            | mersenneTwister[(i + 1) % 624] & 0x7fffffff;
53        if (y % 2 == 0) {
54          mersenneTwister[i] = mersenneTwister[(i + 397) % 624] ^ y >>> 1;
55        } else {
56          mersenneTwister[i] = mersenneTwister[(i + 397) % 624] ^ y >>> 1
57              ^ 0x9908b0df;
58        }
59      }
60      this.currentIndex = 0; // reset index to of number to read next
61    }
62
63    public void setSeed(long newSeed) {
64
65      Random javaAPIRandomGenerator = new Random(newSeed);
66
67      for (int i = 0; i < 624; i++) { // Initialise the generator from seed
68        mersenneTwister[i] = javaAPIRandomGenerator.nextInt();
69      }
70
71      this.twistNumbers(); // Twist them!
72    }
73  }
```

Quelltext 9: Originalimplementation von MersenneTwisterRandomGenerator

```
1   import java.util.function.Supplier;
2   import def.random_js.random.MT19937; // RandomJS-Candy
3   import desmoj.core.dist.UniformRandomGenerator;
4
5   public class MersenneTwisterRandomGeneratorJS implements UniformRandomGenerator{
6
7     private def.random_js.random.Random random;
8
9     /*
10    public MersenneTwisterRandomGeneratorJS() {
11      this(42); //Inkompatibler Aufruf
12    }
13    */
14
15    public MersenneTwisterRandomGeneratorJS(long seed) {
```

```
16      setSeed(seed);
17    }
18
19    @Override
20    public double nextDouble() {
21      return random.real(0, 1, true);
22    }
23
24    @Override
25    public void setSeed(long seed) {
26      Supplier<MT19937> s = def.random_js.random.Random.engines.mt19937;
27      MT19937 mt19937 = s.get();
28      mt19937.seed(42);
29      random = new def.random_js.random.Random(mt19937);
30    }
31  }
```

Quelltext 10: Implementation von MersenneTwisterRandomGenerator mit RandomJS-Candy

```
1  randomGenerator = this.getModel().getExperiment().getDistributionManager().
       getRandomNumberGenerator().newInstance(); // default RandomGenerator
2
3  //Refactoring
4  Class<? extends UniformRandomGenerator> rgclass = this.getModel().getExperiment().
       getDistributionManager().getRandomNumberGenerator();
5  randomGenerator = rgclass.newInstance(); // default RandomGenerator
```

Quelltext 11: Refactoring des newInstance()-Aufrufs in Distribution

# D. Codeanpassungen im Rahmen der Portierung

Die hier aufgezählten Änderungen betreffen die im Rahmen der Portierung der Teilfunktionalität „Eventbasierte Beispielsimulation“ angepassten Klassen zur Erzeugung einer Portierungsversion. Zudem werden die benötigten Nachbearbeitungen im JavaScript-Generat aufgezeigt, die für die Ausführbarkeit der portierten Teilfunktionalität notwendig sind.

## D.1. Import-Mappings

Import-Mappings beziehen sich auf die Abänderung von Importanweisungen von einer im Original verwendeten JDK-Klasse oder externen Bibliothek auf einen im Rahmen der Portierung erstellten Adapter. Diese sind in der Projektstruktur in Bezugnahme auf JSweet-Konventionen im Paket `def` im Quellverzeichnis abgelegt. In Fällen in denen der Adapter nicht alle Aufrufe der ursprünglichen Abhängigkeit abdeckt und diese Klasse weiterhin importiert wird, muss statt einem Austausch von Import-Anweisungen ein Aufruf des Adapters über den voll qualifizierten Namen erfolgen (z.B. `def.System.currentTimeMillis()`).

- java.awt.Color → def.Color
- java.io.File → def.File (für das statische Attribut `separator`)

- java.lang.Integer → def.Integer (für die Methode `toHexString(int)`)
- java.lang.System → def.System (für die Methoden `nanoTime()`, `currentTimeMillis()` und `getProperty("line.separator")`)
- java.util.Observable → def.Observable
- java.util.Observer → def.Observer
- java.util.Random → def.Random
- java.util.concurrent.TimeUnit → def.TimeUnit (auch für statische Imports)

## D.2. Refactorings zur Erzeugung einer übersetzbaren Portierungsversion

Die folgende Aufzählung protokolliert die durchgeführten Änderungen in Klassen die an der zu übersetzenden Teilfunktionalität beteiligt sind. Diese transformationsvorbereitenden Refactorings stellen den manuellen Aufwand dar, der für die Erzeugung der konverterspezifischen Portierungsversion anfällt. Zudem spiegeln sie die Unterschiede zwischen der Portierungsversion und der Originalversion wider. Zur besseren Nachverfolgbarkeit der Änderungen sind die hier aufgeführten Refactorings mit Schlüsselwörtern in eckigen Klammern versehen, die an den veränderten Stellen im beiliegenden Code als Kommentar hinterlegt wurden.

- Abhängigkeiten zu `java.util.concurrent` und zur externen Bibliothek *Quasar* entfernt [PROCESS]
  - `Entity`, `Event`, `EventOf2Entities`, `EventOf3Entities`, `Experiment`, `ExternalEvent`, `ExternalEventStop`, `Model`, `ModelComponent`, `QueueListStandard`, `Schedulable`, `Scheduler`
- def.**BrowserOutputWriter**: Neue Klasse zum Schreiben des Outputs in die Konsole und Öffnen als Pop-Up im Browser
- dist.**Distribution**: `newInstance()`-Aufruf im Konstruktor und Methode `reset()` abgeändert um automatische Übersetzung zu ermöglichen [JSWEET]
- dist.**DistributionManager**: `MersenneTwisterRandomGeneratorJS` verwendet, statt `MersenneTwisterRandomGenerator` [RANDOM]
- dist.**NumericalDist**: `def.NumberValueHelper`-Aufrufe statt `Number.doubleValue()` und `Number.longValue()` [JSWEET]
- report.**FileOutput**: `BufferedWriter` aus Methode `open()` entfernt und `Writer` durch `def.OutputWriter` ersetzt [FILE]
  - Änderung des ursprünglichen Standard-Writers zu `BrowserOutputWriter` in Attribut `NORMAL_FILE_ACCESS`
  - Implementierung des `BrowserOutputWriter` zum Öffnen der HTML-Ausgabe im Browser (und Schreiben in die Konsole)
  - Interface `OutputWriter` als zusätzliche Abstraktionsebene eingeführt. Dies ermöglicht die Verwendung des Outputs ohne Abhängigkeiten zu `Writer`-Klassen des JDK

- report.**FileOutput**: Nutzung von `def.System` für den Aufruf `getProperty("line.separator")` im Attribut `eol` [JSWEET]
- report.**FileSystemAccess**: `OutputWriter` als Rückgabetyp für die Methode `createWriter()` [FILE]
- report.**HtmlTableChartFormatter**: Nutzung von `def.Integer` für den Aufruf `toHexString(int)` in `intToHexString(int)` [JSWEET]
- report.**HtmlTableChartFormatter**: Import von `java.util.Date` statt Nutzung des voll qualifizierten Namens im Code in `close()` [JSWEET]
- report.**HtmlTableFormatter**: Import von `java.util.Date` statt Nutzung des voll qualifizierten Namens im Code in `close()` [JSWEET]
- report.**JavaScriptFormatter**: Nutzung von `def.Integer` für den Aufruf `toHexString(int)` in `intToHexString(int)` [JSWEET]
- report.**ReportMultRowsFileOut**: Abhängigkeiten zu nicht im Teilfragment enthaltenen `Reporter` -Unterklassen entfernt [FRAGMENT]
- report.**SimulationRunReporter**: Nutzung von `java.text.DecimalFormat` entfernt. Gewünschte Formatierung müsste reimplementiert werden. [JSWEET]
- report.**TableOutput**: Nutzung von `def.File` für den Aufruf `File.separator` in `createFileName()` [JSWEET]
- report.**TableOutput**: Aufruf von `System.getProperty("user.dir", ".")` in `createFileName()` durch Standardwert `"."` ersetzt
- report.html5chart.**AbstractChartDataTable**: Aufruf von `Double.POSITIVE_INFINITY` in `getHighestDataValue()` durch `1d/0d` ersetzt (entspricht statischem Wert in `Double`) [JSWEET]
- report.html5chart.**AbstractChartDataTable**: `def.NumberValueHelper`-Aufrufe statt `Number.doubleValue()` in `getHighestDataValue()`| [JSWEET]
- simulator.**Condition**: Aufrufe auf `(PrimitiveWrapper).TYPE` durch `(primitive).class` im Konstruktor und `getType()` ersetzt (z.B. `Integer.TYPE` → `int.class`) [JSWEET]
- simulator.**EventTreeList**: Kollektionstyp `TreeList` von Apache Commons Collections durch `ArrayList` ersetzt [AC_COLLECTION]
- simulator.**Experiment**: Veraltete Methode `randomizeConcurrentEvents()` entfernt, folglich `SortedMapEventList` entfernt [JSWEET]
- simulator.**Experiment**: Erzeugung von `ResourceDB` aus `setupExperiment()` entfernt, zudem das zugehörige Attribut `_resDB` und den Getter `getResourceDB` [PROCESS]
- simulator.**Experiment**: Code für die Anzeige des Fortschrittsbalkens von Experimenten aus `proceed()` in die Methode `showProgressBar()` extrahiert. Aufgrund von Abhängigkeiten zu Swing/AWT mit `@Replace` durch „Noch nicht implementiert“-Warnung ersetzt [GUI]

- simulator.**Experiment**: Validitätscheck von `EventList`-Unterklassen in `setEventList(Class)` in die Methode `checkEventListClassValidity(Class)` extrahiert und mit `@Replace` durch „Noch nicht implementiert"-Warnung ersetzt [REFLECT]
- simulator.**Experiment**: Validitätscheck von `UniformRandomGenerator`-Unterklassen in `setRandomNumberGenerator(Class)` in die Methode `checkRandomGeneratorClassValidity(Class)` extrahiert und mit `@Replace` durch „Noch nicht implementiert"-Warnung ersetzt [REFLECT]
- simulator.**Experiment**: Aufruf von `File` in `getOutputPath` entfernt [JSWEET]
- simulator.**Experiment**: Nutzung von `def.System` für den Aufruf `System.nanoTime()` in `start()` [JSWEET]
- simulator.**Model**: Methode `description()` zu `modelDescription()` umbenannt, um Kollision mit dem Attribut `description` in der Oberklasse `Reportable` zu vermeiden [JSWEET]
- simulator.**Model**: Kollektionstyp `ReferenceMap` von Apache Commons Collections durch `HashMap` ersetzt [AC_COLLECTION]
- simulator.**ModelComponent**: Methode `clone()` hinzugefügt, um den Aufruf zu `Object` weiterzureichen [JSWEET]
- simulator.**ModelCondition**: Aufrufe auf `(PrimitiveWrapper).TYPE` durch `(primitive).class` im Konstruktor und `getType()` ersetzt (z.B. `Integer.TYPE` → `int.class`) [JSWEET]
- simulator.**NamedObject**: Methode `clone()` hinzugefügt, um den Aufruf zu `Object` weiterzureichen [JSWEET]
- simulator.**QueueListRandom**: Aufruf von `nextInt(int)` in `insert(E)` durch `nextIntWithBound(int)` von `def.Random` ersetzt, um Probleme bei der Überladung zu verhindern [RANDOM]
- simulator.**QueueListRandom**: Erzeugung von `Random` den Voraussetzungen von `def.Random` angepasst (Notwendiger Aufruf von `autoSeed()`) [RANDOM]
- simulator.**QueueListStandard**: Kollektionstyp `LinkedList` durch `ArrayList` ersetzt [JSWEET]
  - Anpassungen auch in `QueueListFifo`, `QueueListLifo`
- simulator.**QueueListStandard**: Interface `PropertyChangeListener` und dazugehörige Methode `propertyChange` entfernt [PROCESS]
  - Anpassungen auch in `QueueList`, `QueueListFifo`, `QueueListLifo`, `QueueListRandom`
- simulator.**RandomizingEventTreeList**: Aufruf von `nextInt(int)` in `insert(E)` durch `nextIntWithBound(int)` von `def.Random` ersetzt, um Probleme bei der Überladung zu verhindern [RANDOM]
- simulator.**RandomizingEventTreeList**: Erzeugung von `Random` den Voraussetzungen von `def.Random` angepasst (Notwendiger Aufruf von `autoSeed()`) [RANDOM]
- simulator.**RealTimeEventWrapper**: Nutzung von `def.System` für den Aufruf `System.nanoTime()` in `start()` [JSWEET]

- simulator.**Reportable**: Erzeugungversuch von benutzerdefinierten `Reporter`-Klassen aus `getReporter()` in `tryInstatiatingCustomReporter()` extrahiert und mit `@Replace` durch „Noch nicht implementiert"-Warnung ersetzt. Stattdessen wird Standardklasse genutzt [REFLECT]
- simulator.**Schedulable**.DummyReporter: Erzeugungversuch von benutzerdefinierten `Reporter`-Klassen aus `createDefaultReporter()` entfernt. Stattdessen wird Standardklasse genutzt [REFLECT]
- simulator.**Scheduler**: Abhängigkeiten zu `java.util.concurrent` entfernt (Klassen `ReentrantLock` und `Condition`) [CONCURRENT]
- simulator.**Scheduler**: Kollektionstyp `LinkedBlockingQueue` durch `LinkedList` ersetzt [CONCURRENT]
- simulator.**Scheduler**: Nutzung von `def.System` für den Aufruf `System.nanoTime()` in `start()` [JSWEET]
- simulator.**SingleUnitTimeFormatter**: Nutzung der Methode `append(String)` von `StringBuffer`, da `insert(int, String)` nicht von JSweet unterstützt wird [JSWEET]
- simulator.**SingleUnitTimeFormatter**: `EnumMap`-Konstanten entfernt, da der Kollektionstyp nicht von JSweet unterstützt wird [JSWEET]
- simulator.**SingleUnitTimeFormatter**: Nutzung von `toLowerCase()` statt `toLowerCase(Locale)` in `getUnit()` [JSWEET]
- simulator.**TimeInstant**: Methode `getBeginOf(TimeUnit)` mit `def.js.Date` aus der Konverter-API reimplementiert [TIME]
- simulator.**TimeInstant**: Methode `getTimeAsCalendar()` entfernt, da sie in der JavaScript-Übersetzung mit `getTimeAsDate` identisch ist [TIME]

## D.3. Nachbearbeitungsbedarf im JavaScript-Generat

Idealerweise sollte das JSweet-Generat ohne weitere Anpassungen lauffähig sein, allerdings ist dies aufgrund von Bugs im Konvertierungswerkzeug und nachträglich notwendige Reimplementierung von Funktionen nicht immer gegeben. Daher ist in diesem Abschnitt aufgelistet, welche Nachbearbeitungsbedarfe in der betrachteten Teilfunktionalität auftreten.

- def.MersenneTwisterRandomGeneratorJS: Aufrufe auf das verwendete RandomJS-Candy werden fehlerhaft übersetzt und müssen korrigiert werden (`random.Random -> Random`)
- def.Random: Aufrufe auf das verwendete RandomJS-Candy werden fehlerhaft übersetzt und müssen korrigiert werden (`random.Random -> Random`)
- simulator.EventTreeList: In der Methode `removeFirst` wird ein Array der Länge 1 mit dem gesuchten Objekt statt des Objekts verwendet. Das Objekt muss daher aus dem Array abgerufen werden (`noteObject -> noteObject[0]`)
- def.BrowserOutputWriter: Das Öffnen des generierten HTML-Texts im Browserfenster muss implementiert werden

# E. Gegenüberstellung von Input und Output

Der dargestellte Code wurde für bessere Lesbarkeit an strukturell unbedeutenden Stellen gekürzt. Dies ist mit `/*[...]*/` gekennzeichnet.

```
1  package desmoj.core.simulator;
2
3  import desmoj.core.dist.NumericalDist;
4
5  public abstract class ExternalEvent extends EventAbstract
6  {
7
8    public ExternalEvent(Model owner, String name, boolean showInTrace) {
9      super(owner, name, showInTrace);
10     this.numberOfEntities = 0;
11   }
12
13   public abstract void eventRoutine();
14
15   public void schedule() {
16
17     // generate trace
18     this.generateTraceForScheduling(null, null, null, null, null, presentTime(), null);
19
20     // schedule Event
21     getModel().getExperiment().getScheduler().scheduleNoPreempt(null, this, presentTime())
       ;
22
23     if (currentlySendDebugNotes()) {
24       sendDebugNote("schedules on EventList<br>"
25       + getModel().getExperiment().getScheduler().toString());
26     }
27   }
28
29   public void schedule(TimeSpan dt) {
30     if ((dt == null)) {
31       sendWarning(/*[...]*/);
32       return; // no proper parameter
33     }
34
35     // generate trace
36     this.generateTraceForScheduling(null, null, null, null, null, TimeOperations.add(
       presentTime(), dt), null);
37
38     // schedule Event
39     getModel().getExperiment().getScheduler().scheduleNoPreempt(null, this, dt);
40
41     if (currentlySendDebugNotes()) {
42       sendDebugNote("schedules on EventList<br>"
43       + getModel().getExperiment().getScheduler().toString());
44     }
45   }
46
47   public void schedule(NumericalDist<?> dist) {
```

```
48
49      if ((dist == null)) {
50        sendWarning(/*[...]*/);
51        return; // no proper parameter
52      }
53
54      // determine time span
55      TimeSpan dt = dist.sampleTimeSpan();
56
57      // generate trace
58      this.generateTraceForScheduling(null, null, null, null, null, TimeOperations.add(
        presentTime(), dt), " Sampled from " + dist.getQuotedName() + ".");
59
60      // schedule Event
61      getModel().getExperiment().getScheduler().scheduleNoPreempt(null, this, dt);
62
63      if (currentlySendDebugNotes()) {
64        sendDebugNote("schedules on EventList<br>"
65        + getModel().getExperiment().getScheduler().toString());
66      }
67    }
68
69    public void schedule(TimeInstant when) {
70      if ((when == null)) {
71        sendWarning(/*[...]*/);
72        return; // no proper parameter
73      }
74
75      // generate trace
76      this.generateTraceForScheduling(null, null, null, null, null, when, null);
77
78      // schedule Event
79      getModel().getExperiment().getScheduler().scheduleNoPreempt(null, this, when);
80
81      if (currentlySendDebugNotes()) {
82        sendDebugNote("schedules on EventList<br>"
83        + getModel().getExperiment().getScheduler().toString());
84      }
85    }
86
87    public void scheduleAfter(Schedulable after) {
88
89      if ((after == null)) {
90        sendWarning(/*[...]*/);
91        return; // no proper parameter
92      }
93
94      if (!after.isScheduled()) {
95        sendWarning(/*[...]*/);
96        return; // was not scheduled
97      }
98
99      // generate trace
```

```
100     this.generateTraceForScheduling(null, null, null, after, null, after.getEventNotes().
        get(after.getEventNotes().size()-1).getTime(), null);
101
102     // schedule Event
103     getModel().getExperiment().getScheduler().scheduleAfter(after, null,
104     this);
105
106     if (currentlySendDebugNotes()) {
107       sendDebugNote("scheduleAfter " + after.getQuotedName()
108       + " on EventList<br>"
109       + getModel().getExperiment().getScheduler().toString());
110     }
111
112   }
113
114   public void scheduleBefore(Schedulable before) {
115
116     if ((before == null)) {
117       sendWarning(/*[...]*/);
118       return; // no proper parameter
119     }
120
121     if (!before.isScheduled()) {
122       sendWarning(/*[...]*/);
123       return; // was not scheduled
124     }
125
126     // generate trace
127     this.generateTraceForScheduling(null, null, null, null, before, before.getEventNotes()
        .get(0).getTime(), null);
128
129     // schedule Event
130     getModel().getExperiment().getScheduler().scheduleBefore(before, null,
131     this);
132
133     if (currentlySendDebugNotes()) {
134       sendDebugNote("scheduleBefore " + before.getQuotedName()
135       + " on EventList<br>"
136       + getModel().getExperiment().getScheduler().toString());
137     }
138
139   }
140
141   protected ExternalEvent clone() throws CloneNotSupportedException {
142     return (ExternalEvent) super.clone();
143   }
144 }
```

Quelltext 12: Java-Code der Klasse ExternalEvent in der Portierungsversion

```
1 var __extends = (this && this.__extends) || function (d, b) {
2   for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
3   function __() { this.constructor = d; }
4   d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
```

```
};

/* Generated from Java with JSweet 2.0.0-rc1 - http://www.jsweet.org */
var desmoj;
(function (desmoj) {
  var core;
  (function (core) {
    var simulator;
    (function (simulator) {

      var ExternalEvent = (function (_super) {
        __extends(ExternalEvent, _super);

        function ExternalEvent(owner, name, showInTrace) {
          var _this = _super.call(this, owner, name, showInTrace) || this;
          _this.numberOfEntities = 0;
          return _this;
        }

        ExternalEvent.prototype.eventRoutine = function (who) {
          if (who === undefined) {
            return this.eventRoutine$();
          }
          else
          throw new Error('invalid overload');
        };

        ExternalEvent.prototype.eventRoutine$ = function () { throw new Error('cannot
    invoke abstract overloaded method... check your argument(s) type(s)'); };

        ExternalEvent.prototype.schedule = function (dt) {
          if (((dt != null && dt instanceof desmoj.core.simulator.TimeSpan) || dt === null
    )) {
            return this.schedule$desmoj_core_simulator_TimeSpan(dt);
          }
          else if (((dt != null && dt instanceof desmoj.core.dist.NumericalDist) || dt ===
     null)) {
            return this.schedule$desmoj_core_dist_NumericalDist(dt);
          }
          else if (((dt != null && dt instanceof desmoj.core.simulator.TimeInstant) || dt
    === null)) {
            return this.schedule$desmoj_core_simulator_TimeInstant(dt);
          }
          else if (dt === undefined) {
            return this.schedule$();
          }
          else
          throw new Error('invalid overload');
        };

        ExternalEvent.prototype.schedule$ = function () {
          this.generateTraceForScheduling(null, null, null, null, null, this.presentTime()
    , null);
```

```
53          this.getModel().getExperiment().getScheduler().scheduleNoPreempt$
    desmoj_core_simulator_Entity$desmoj_core_simulator_EventAbstract$
    desmoj_core_simulator_TimeInstant(null, this, this.presentTime());
54          if (this.currentlySendDebugNotes()) {
55            this.sendDebugNote("schedules on EventList<br>" + this.getModel().
    getExperiment().getScheduler().toString());
56          }
57        };
58
59        ExternalEvent.prototype.schedule$desmoj_core_simulator_TimeSpan = function (dt) {
60          if ((dt == null)) {
61            this.sendWarning(/*[...]*/);
62            return;
63          }
64          this.generateTraceForScheduling(null, null, null, null, null, desmoj.core.
    simulator.TimeOperations.
    add$desmoj_core_simulator_TimeInstant$desmoj_core_simulator_TimeSpan(this.presentTime
    (), dt), null);
65          this.getModel().getExperiment().getScheduler().scheduleNoPreempt$
    desmoj_core_simulator_Entity$desmoj_core_simulator_EventAbstract$
    desmoj_core_simulator_TimeSpan(null, this, dt);
66          if (this.currentlySendDebugNotes()) {
67            this.sendDebugNote("schedules on EventList<br>" + this.getModel().
    getExperiment().getScheduler().toString());
68          }
69        };
70
71        ExternalEvent.prototype.schedule$desmoj_core_dist_NumericalDist = function (dist)
    {
72          if ((dist == null)) {
73            this.sendWarning(/*[...]*/);
74            return;
75          }
76          var dt = dist.sampleTimeSpan();
77          this.generateTraceForScheduling(null, null, null, null, null, desmoj.core.
    simulator.TimeOperations.
    add$desmoj_core_simulator_TimeInstant$desmoj_core_simulator_TimeSpan(this.presentTime
    (), dt), " Sampled from " + dist.getQuotedName() + ".");
78          this.getModel().getExperiment().getScheduler().scheduleNoPreempt$
    desmoj_core_simulator_Entity$desmoj_core_simulator_EventAbstract$
    desmoj_core_simulator_TimeSpan(null, this, dt);
79          if (this.currentlySendDebugNotes()) {
80            this.sendDebugNote("schedules on EventList<br>" + this.getModel().
    getExperiment().getScheduler().toString());
81          }
82        };
83
84        ExternalEvent.prototype.schedule$desmoj_core_simulator_TimeInstant = function (
    when) {
85          if ((when == null)) {
86            this.sendWarning(/*[...]*/);
87            return;
88          }
89          this.generateTraceForScheduling(null, null, null, null, null, when, null);
```

```
90          this.getModel().getExperiment().getScheduler().scheduleNoPreempt$
    desmoj_core_simulator_Entity$desmoj_core_simulator_EventAbstract$
    desmoj_core_simulator_TimeInstant(null, this, when);
91          if (this.currentlySendDebugNotes()) {
92            this.sendDebugNote("schedules on EventList<br>" + this.getModel().
    getExperiment().getScheduler().toString());
93          }
94        };
95
96        ExternalEvent.prototype.scheduleAfter = function (after) {
97          if ((after == null)) {
98            this.sendWarning(/* [...] */);
99            return;
100         }
101         if (!after.isScheduled()) {
102           this.sendWarning(/* [...] */);
103           return;
104         }
105         this.generateTraceForScheduling(null, null, null, after, null, /* get */ after.
    getEventNotes()[after.getEventNotes().length - 1].getTime(), null);
106         this.getModel().getExperiment().getScheduler().scheduleAfter$
    desmoj_core_simulator_Schedulable$desmoj_core_simulator_Entity$
    desmoj_core_simulator_EventAbstract(after, null, this);
107         if (this.currentlySendDebugNotes()) {
108           this.sendDebugNote("scheduleAfter " + after.getQuotedName() + " on EventList<
    br>" + this.getModel().getExperiment().getScheduler().toString());
109         }
110       };
111
112       ExternalEvent.prototype.scheduleBefore = function (before) {
113         if ((before == null)) {
114           this.sendWarning(/* [...] */);
115           return;
116         }
117         if (!before.isScheduled()) {
118           this.sendWarning(/* [...] */);
119           return;
120         }
121         this.generateTraceForScheduling(null, null, null, null, before, /* get */ before
    .getEventNotes()[0].getTime(), null);
122         this.getModel().getExperiment().getScheduler().scheduleBefore$
    desmoj_core_simulator_Schedulable$desmoj_core_simulator_Entity$
    desmoj_core_simulator_EventAbstract(before, null, this);
123         if (this.currentlySendDebugNotes()) {
124           this.sendDebugNote("scheduleBefore " + before.getQuotedName() + " on EventList
    <br>" + this.getModel().getExperiment().getScheduler().toString());
125         }
126       };
127
128       ExternalEvent.prototype.clone = function () {
129         var _this = this;
130         return (function (o) { if (_super.prototype.clone != undefined) {
131           return _super.prototype.clone.call(_this);
132         }
```

```
133           else {
134             var clone = Object.create(o);
135             for (var p in o) {
136               if (o.hasOwnProperty(p))
137               clone[p] = o[p];
138             }
139             return clone;
140           } })(this);
141         };
142 
143         return ExternalEvent;
144       }(desmoj.core.simulator.EventAbstract));
145       simulator.ExternalEvent = ExternalEvent;
146       ExternalEvent["__class"] = "desmoj.core.simulator.ExternalEvent";
147     })(simulator = core.simulator || (core.simulator = {}));
148   })(core = desmoj.core || (desmoj.core = {}));
149 })(desmoj || (desmoj = {}));
```

Quelltext 13: Für die Klasse ExternalEvent generierten JavaScript-Code

# F. Evaluation der Traceability Recovery

| Paket/Typ | Interface | Rang des korrekten Links | P | MAP | Korrektes Zielelement |
|---|---|---|---|---|---|
| **Paket: dist** | | | | **94,44%** | |
| dist.ContDist.java | | 1 | 100,00% | | var ContDist |
| dist.ContDistExponential.java | | 1 | 100,00% | | var ContDistExponential |
| dist.ContDistUniform.java | | 1 | 100,00% | | var ContDistUniform |
| dist.ContDistWeibull.java | | 1 | 100,00% | | var ContDistWeibull |
| dist.Distribution.java | | 1 | 100,00% | | var Distribution |
| dist.DistributionManager.java | | 2 | 50,00% | | var DistributionManager |
| dist.LinearCongruentialRandomGenerator.java | | 1 | 100,00% | | var LinearCongruentialRandomGenerator |
| dist.MersenneTwisterRandomGenerator.java | | 1 | 100,00% | | var MersenneTwisterRandomGenerator |
| dist.NumericalDist.java | | 1 | 100,00% | | var NumericalDist |
| dist.UniformRandomGenerator.java | Ja | | #DIV/0! | | |
| **Paket: exception** | | | | **100,00%** | |
| exception.DESMOJException.java | | 1 | 100,00% | | var DESMOJException |
| exception.SimAbortedException.java | | 1 | 100,00% | | var SimAbortedException |
| exception.SimFinishedException.java | | 1 | 100,00% | | var SimFinishedException |
| **Paket: observer** | | | | **100,00%** | |
| observer.Observer.java | Ja | | #DIV/0! | | |
| observer.Subject.java | Ja | | #DIV/0! | | |
| observer.SubjectAdministration.java | | 1 | 100,00% | | var SubjectAdministration |
| **Paket: report** | | | | **100,00%** | |
| report.AbstractTableFormatter.java | | 1 | 100,00% | | var AbstractTableFormatter |
| report.ContDistExponReporter.java | | 1 | 100,00% | | var ContDistExponReporter |
| report.ContDistUniformReporter.java | | 1 | 100,00% | | var ContDistUniformReporter |
| report.DebugFileOut.java | | 1 | 100,00% | | var DebugFileOut |
| report.DebugNote.java | | 1 | 100,00% | | var DebugNote |
| report.DistributionReporter.java | | 1 | 100,00% | | var DistributionReporter |
| report.ErrorFileOut.java | | 1 | 100,00% | | var ErrorFileOut |
| report.ErrorMessage.java | | 1 | 100,00% | | var ErrorMessage |
| report.FileOutput.java | | 1 | 100,00% | | var FileOutput |
| report.FileSystemAccess.java | Ja | | #DIV/0! | | |
| report.HTMLDebugOutput.java | | 1 | 100,00% | | var HTMLDebugOutput |
| report.HTMLErrorOutput.java | | 1 | 100,00% | | var HTMLErrorOutput |
| report.HTMLReportOutput.java | | 1 | 100,00% | | var HTMLReportOutput |
| report.HTMLTableChartFormatter.java | | 1 | 100,00% | | var HTMLTableChartFormatter |
| report.HTMLTableFormatter.java | | 1 | 100,00% | | var HTMLTableFormatter |
| report.HTMLTraceOutput.java | | 1 | 100,00% | | var HTMLTraceOutput |
| report.JavaScriptFormatter.java | | 1 | 100,00% | | var JavaScriptFormatter |
| report.Message.java | | 1 | 100,00% | | var Message |
| report.MessageDistributor.java | | 1 | 100,00% | | var MessageDistributor |
| report.MessageReceiver.java | Ja | | #DIV/0! | | |
| report.ModelReporter.java | | 1 | 100,00% | | var ModelReporter |
| report.OutputType.java | Ja | | #DIV/0! | | |
| report.OutputTypeEndToExport.java | Ja | | #DIV/0! | | |
| report.QueueReporter.java | | 1 | 100,00% | | var QueueReporter |
| report.Reporter.java | | 1 | 100,00% | | var Reporter |
| report.ReportFileOut.java | | 1 | 100,00% | | var ReportFileOut |
| report.ReportManager.java | | 1 | 100,00% | | var ReportManager |
| report.ReportMultRowsFileOut.java | | 1 | 100,00% | | var ReportMultRowsFileOut |
| report.SimulationRunReporter.java | | 1 | 100,00% | | var SimulationRunReporter |
| report.StandardReporter.java | | 1 | 100,00% | | var StandardReporter |
| report.TableFormatter.java | Ja | | #DIV/0! | | |
| report.TableOutput.java | | 1 | 100,00% | | var TableOutput |
| report.TableReporter.java | | 1 | 100,00% | | var TableReporter |
| report.TraceFileOut.java | | 1 | 100,00% | | var TraceFileOut |
| report.TraceNote.java | | 1 | 100,00% | | var TraceNote |
| **Paket: report.html5chart** | | | | **100,00%** | |
| report.html5chart.AbstractChartData.java | Ja | | #DIV/0! | | |
| report.html5chart.AbstractChartDataTable.java | | 1 | 100,00% | | var AbstractChartDataTable |
| report.html5chart.AbstractNumericalChartCanvas.java | | 1 | 100,00% | | var AbstractNumericalChartCanvas |
| report.html5chart.AbstractNumericalCoorChartCanvas.java | | 1 | 100,00% | | var AbstractNumericalChartCanvas |
| report.html5chart.AbstractNumericalCoorChartCanvasDouble.java | | 1 | 100,00% | | var AbstractNumericalCoorChartCanvasDouble |
| report.html5chart.AbstractNumericalCoorChartCanvasLong.java | | 1 | 100,00% | | var AbstractNumericalCoorChartCanvasLong |
| report.html5chart.Canvas.java | Ja | | #DIV/0! | | |
| report.html5chart.CanvasCoordinateChart.java | Ja | | #DIV/0! | | |
| report.html5chart.CanvasCoordinateChartInterval.java | Ja | | #DIV/0! | | |
| report.html5chart.CanvasHistogramDouble.java | | 1 | 100,00% | | var CanvasHistogramDouble |
| report.html5chart.CanvasHistogramLong.java | | 1 | 100,00% | | var CanvasHistogramDoublelong |
| report.html5chart.CanvasTimeSeries.java | | 1 | 100,00% | | var CanvasTimeSeries |
| report.html5chart.ChartDataHistogramDouble.java | | 1 | 100,00% | | var ChartDataHistogramDouble |
| report.html5chart.ChartDataHistogramLong.java | | 1 | 100,00% | | var ChartDataHistogramLong |
| report.html5chart.ChartDataTimeSeries.java | | 1 | 100,00% | | var ChartDataTimeSeries |

| Paket/Typ | Interface | Rang des korrekten Links | P | MAP | Korrektes Zielelement |
|---|---|---|---|---|---|
| **Paket: simulator** | | | | **96,37%** | |
| simulator.Condition.java | | 1 | 100,00% | | var Condition |
| simulator.Entity.java | | 1 | 100,00% | | var Entity |
| simulator.Event.java | | 1 | 100,00% | | var Event |
| simulator.EventAbstract.java | | 1 | 100,00% | | var EventAbstract |
| simulator.EventList.java | (Ja) | | #DIV/0! | | |
| simulator.EventNote.java | | 1 | 100,00% | | var EventNote |
| simulator.EventOf2Entities.java | | 4 | 25,00% | | var EventOfEntities |
| simulator.EventOf3Entities.java | | 3 | 33,33% | | var EventOfEntities |
| simulator.EventTreeList.java | | 1 | 100,00% | | var EventTreeList |
| simulator.Experiment.java | | 1 | 100,00% | | var Experiment |
| simulator.ExperimentParameterManager.java | Ja | | #DIV/0! | | |
| simulator.ExternalEvent.java | | 1 | 100,00% | | var ExternalEvent |
| simulator.ExternalEventDebugOff.java | | 1 | 100,00% | | var ExternalEventDebugOff |
| simulator.ExternalEventDebugOn.java | | 1 | 100,00% | | var ExternalEventDebugOn |
| simulator.ExternalEventStop.java | | 1 | 100,00% | | var ExternalEventStop |
| simulator.ExternalEventTraceOff.java | | 1 | 100,00% | | var ExternalEventTraceOff |
| simulator.ExternalEventTraceOn.java | | 1 | 100,00% | | var ExternalEventTraceOn |
| simulator.Model.java | | 1 | 100,00% | | var Model |
| simulator.ModelComponent.java | | 1 | 100,00% | | var ModelComponent |
| simulator.ModelCondition.java | | 1 | 100,00% | | var ModelCondition |
| simulator.ModelOptions.java | Ja | | #DIV/0! | | |
| simulator.ModelParameterManager.java | Ja | | #DIV/0! | | |
| simulator.NameCatalog.java | | 1 | 100,00% | | var NameCatalog |
| simulator.NamedObject.java | | 1 | 100,00% | | var NamedObject |
| simulator.Parameter.java | | 1 | 100,00% | | var Parameter |
| simulator.ParameterManager.java | | 1 | 100,00% | | var ParameterManager |
| simulator.Queue.java | | 1 | 100,00% | | var Queue |
| simulator.QueueBased.java | | 1 | 100,00% | | var QueueBased |
| simulator.QueueList.java | | 1 | 100,00% | | var QueueList |
| simulator.QueueListFifo.java | | 1 | 100,00% | | var QueueListFifo |
| simulator.QueueListLifo.java | | 1 | 100,00% | | var QueueListLifo |
| simulator.QueueListRandom.java | | 1 | 100,00% | | var QueueListRandom |
| simulator.QueueListStandard.java | | 1 | 100,00% | | var QueueListStandard |
| simulator.RandomizingEventTreeList.java | | 1 | 100,00% | | var RandomizingEventTreeList |
| simulator.RealTimeEventWrapper.java | | 1 | 100,00% | | var RealTimeEventWrapper |
| simulator.Reportable.java | | 1 | 100,00% | | var Reportable |
| simulator.Schedulable.java | | 1 | 100,00% | | var Schedulable |
| simulator.Scheduler.java | | 1 | 100,00% | | var Scheduler |
| simulator.SimClock.java | | 1 | 100,00% | | var SimClock |
| simulator.SingleUnitTimeFormatter.java | | 1 | 100,00% | | var SingleUnitTimeFormatter |
| simulator.TimeFormatter.java | Ja | | #DIV/0! | | |
| simulator.TimeInstant.java | | 1 | 100,00% | | var TimeInstant |
| simulator.TimeOperations.java | | 1 | 100,00% | | var TimeOperations |
| simulator.TimeSpan.java | | 1 | 100,00% | | var TimeSpan |
| **Paket: statistic** | | | | **100,00%** | |
| statistic.StatisticObject.java | | 1 | 100,00% | | var StatisticObject |
| | | | | | |
| **Gesamt** | | | | **97,96%** | |

# Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Zudem versichere ich, dass die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde und die schriftliche Version der auf dem beigelegten elektronischen Speichermedium entspricht.

Datum: Hamburg, den

Unterschrift: