Update+
Java-Community

From Legacy to modern Web – ein Reisebericht

Mirko Sertic

Das Internet ist zur Standard-Plattform für Anwendungen aller Art geworden. Eine große Herausforderung ist, bestehende Anwendungen an diesen Standard anzupassen und anpassbar zu halten.

Dieser Artikel zeigt am Beispiel einer Testbed-Anwendung, wie Domain-Driven Design und eventbasierte Architekturen als Grundlage für portierbare Anwendungen verwendet werden können. Ein besonderer Schwerpunkt liegt auf der Nutzung von bestehendem Quellcode und vorhandenen Libraries, Präsentations-Frameworks unterschiedlicher Art (JavaFX / HTML5) und den Möglichkeiten von Transpiler-Technologien, um den Spagat zwischen Wiederverwendung und technologischer Erneuerung zu meistern.

Die Reise

Am Anfang jeder Reise steht eine Idee, vielleicht auch ein gewisses Fernweh. Auch am Anfang meiner Reise war das nicht anders. Der Autor hat nicht von Palmen auf einer Südseeinsel geträumt, sondern sich mit Fragen über System- und Software-Architektur beschäftigt: Unsere (technologische) Welt verändert sich sehr schnell. Wir müssen uns ständig fragen, welche Technologie und welches Framework das Beste für den jeweiligen Anwendungsfall ist. Wir müssen uns fragen, welche Erscheinungen auf dem Mark nur ein kurzfristiger Hype sind und welche einen echten Mehrwert darstellen. Nicht zuletzt müssen wir uns über die wirtschaftlichen Aspekte unserer Lösungen Gedanken machen. Ist das wirtschaftlich? Ist das nachhaltig? Ist das wartbar? Funktioniert das wirklich? Aus all diesen Fragen ist der Wunsch für eine Testbed-Anwendung entstanden, um konkretere Anwendungen und auch Testfälle für bestehende und neue Technologien zu bekommen.

Was für ein Testbed notwendig ist

Für alle Programmiersprachen und Frameworks existiert ein klassisches „Hello World"-Beispiel. Damit erkennt man meistens mit wenigen Zeilen Quellcode die wichtigsten Features einer Programmiersprache. Für eine schnelle Übersicht sind diese Beispiele nicht schlecht. Für eine qualifiziertere Beurteilung sind sie allerdings viel zu klein. In einem Testbed braucht man komplexe Domänen-Logik. Man benötigt synchrone und asynchrone Komponenten sowie performancekritische Anforderungen. Es ist eine auf unterschiedliche Plattformen portierbare Anwendung erforderlich. Bei diesen Überlegungen kam dann dem Autor die ursprüngliche Motivation für seine Reise in den Sinn: Er möchte etwas prüfen und dabei lernen. Er möchte Spaß mit Arbeit kombinieren, was gerne unter dem Stichwort „Gamification" zusammengefasst wird. Und genau dieses Wort lieferte die entscheidenden Hinweise.

Ein Computerspiel erfüllt bei genauerer Betrachtung all seine Anforderungen an eine Testbed-Anwendung. Es besitzt komplexe Domänen-Logik bis hin zu Physik-Simulationen. Es besteht aus synchronen und asynchronen Komponenten. Es ist sehr stark ereignisgetrieben. Es stellt sehr hohe Anforderungen an die Performance. Ein Computerspiel ist idealerweise nicht an eine bestimmte Plattform gebunden. Es sollte auf dem PC genauso laufen wie auf einem Android- oder iOS-Handy oder auch auf einem Tablet. Das Testbed sollte also ein Computerspiel werden.

Koffer packen

Vor jeder Reise müssen Vorbereitungen getroffen werden. In fremden Ländern werden natürlich auch fremde Sprachen gesprochen. Es gibt Unterschiede in der Landschaft und auch in den Bräuchen. Auf diese Unterschiede muss sich der Reisende natürlich auch einlassen.

Zum Glück ist das Thema „Sprache" bei dieser Reise ein kleineres Problem, denn als plattformunabhängige Programmiersprache war Java sehr schnell das Mittel der Wahl. Viel problematischer ist die angesprochene Komplexität in der Testbed- Anwendung. Für die Modellierung von genau solchen Systemen gibt es zum Glück schon ein Werkzeug in unserem Werkzeugkasten, das natürlich auch in den Reisekoffer gehört: Domain-Driven Design. Darüber gibt es zwei Bücher, die mit ins Reisegepäck kommen: „Domain-Driven Design" von Eric Evans und „Implementing Domain-Driven Design" von Vaugn Vernon. Über die Entwicklung von Computerspielen gibt es ebenfalls zwei Bücher, die dem Autor besonders gut gefallen und die er deshalb auch mit auf die Reise nehmen möchte: „Real-Time Rendering: Third Edition" von Tomas Akenine-Möller, Eric Haines und Naty Hoffmann sowie „Game Coding Complete: Fourth Edition" von Mike McShaffry und David Graham. Diese Bücher lieferten die Grundlagen für diese Reise.

Fremde Sprachen lernen

Auf dieser Reise wird zum Glück überall Java gesprochen. Aus diesem Grund sollte das Thema „Sprache" eigentlich kein großes Problem darstellen. Trotzdem ist das Sprachthema ein Problem, und zwar ein großes. Wie sich herausstellt, werden hier nämlich zwei Sprachen gesprochen. Java ist die technische Sprache für die Formulierung von Quellcode. Es gibt aber noch eine zweite Sprache, die noch viel wichtiger ist und überall gesprochen wird: die Domänensprache, also die Sprache der Nicht-Techniker.

Wie sich als Vorbereitung für die Reise durch intensives Studium der erwähnten Fachliteratur herausstellte, ist die Domänensprache sehr komplex und umfangreich. Hier tauchen Begriffe auf wie „Game Objects", „Instrances", „Behaviors", „Sprites", „Views", „Events", „Templates" und „Sounds" sowie eine ganze Menge von Ereignissen und Beziehungen, die diese Begriffe auslösen und untereinander haben.

Wie passt nun diese Domänensprache mit der Implementierungssprache zusammen? Die Antwort auf diese Frage liefert uns das Domain-Driven Design. Es kennt den Begriff der „ubiquitous language" – der allgegenwärtigen Sprache. Diese Sprache ist spezifisch für eine Fachdomäne und beschreibt diese möglichst genau und widerspruchsfrei. Im weitesten Sinne ist sie also ein Wörterbuch. In diesem Wörterbuch kann man im Lauf der Reise immer wieder nachschlagen. Natürlich lassen sich auch Ergänzungen vornehmen, falls man etwas Neues entdeckt oder ein Eintrag vielleicht doch nicht ganz richtig ist.

Somit haben wir jetzt sowohl die Implementierungs- als auch die Domänensprache sehr stark normiert – aber wie sieht es mit der erwähnten Komplexität aus? Das Testbed soll natürlich kein aus Spaghetti-Code bestehender Monolith werden. Um dies zu verhindern, muss der Quellcode anders strukturiert sein. Damit diese Strukturierung so effizient wie möglich ist, beginnen wir mit der Strukturierung schon auf der Ebene der Domänensprache. Der Umgang mit Komplexität ist also nicht nur ein Teil der technischen Implementierung, sondern ein

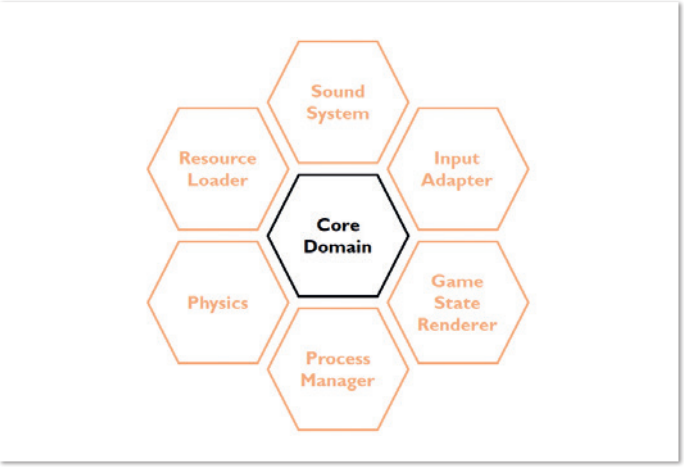

Abbildung 1: Context-Map der Testbed-Anwendung

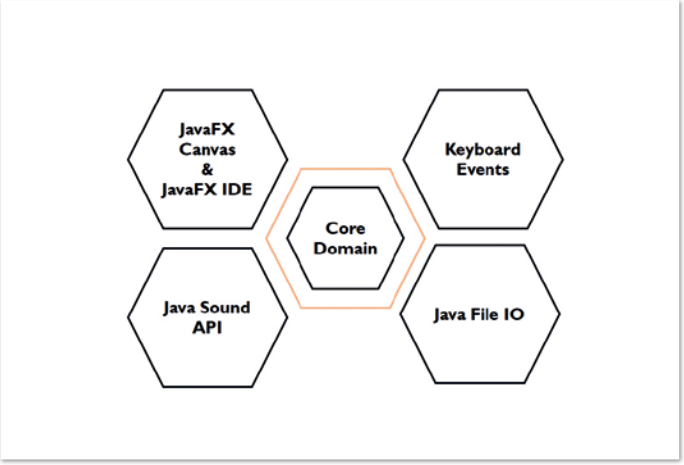

Abbildung 2: Die JavaFX-System-Architektur

wichtiger Teil der gesamten Problem-Modellierung. Domain-Driven Design kennt dafür das Werkzeug der sogenannten „Context-Map". Damit zerlegen wir ein großes System in mehrere Teilsysteme, gewinnen einen besseren Überblick und können damit auch die Komplexität des Gesamtsystems besser im Griff behalten. Die technische Struktur des Quellcodes kann sich dann an der Struktur der Context-Map orientieren beziehungsweise ist sogar identisch damit in Hinblick auf Modul- und Paket-Strukturen sowie deren Abhängigkeiten. *Abbildung 1* zeigt die erarbeitete Context-Map für das Testbed.

In der Mitte steht die Core-Domain. Darin ist der Kern des Systems, also die Essenz, enthalten. Um diese Core-Domain sind Adapter-Domänen angeordnet. Sie stellen entweder Schnittstellen zur Verfügung oder kapseln Funktionalität, die nicht Teil der Core-Domain, aber trotzdem für die Lauffähigkeit der Anwendung notwendig ist. Beispiele für Adapter-Domänen sind das Sound-System oder die Input-Adapter-Domänen.

Durch Anpassung beziehungsweise kompletten Austausch dieser Adapter lässt sich die Core-Domain einfach an andere Umgebungen anpassen. Ein Beispiel für eine eher kapselnde Domäne ist „Physics". Für die Lauffähigkeit des Systems wird in diesem Beispiel zwar eine Physik-Simulation benötigt, der Core-Domain ist es aber egal, welche konkrete Implementierung hier zum Einsatz kommt. Durch dieses Design kann beispielsweise das komplette Physik-Subsystem ausgetauscht werden, falls es zu Laufzeit-Problemen kommen sollte oder wichtige Features fehlen. Die Core-Domain muss dafür nicht angepasst werden.

In der Grafik sind die einzelnen Domänen ganz bewusst als Hexagone dargestellt. Es handelt sich also um eine hexagonale System-Architektur. Diese Architektur-Form bietet die zuvor genannten Vorteile und erleichtert zudem die Modularisierung, damit also auch die Test- und Wartbarkeit des Systems. Ein wesentliches Merkmal ist, dass in der Mitte unserer Hexagon-Struktur ein Rich-Domain-Modell steht, das keine Abhängigkeiten nach außen hat; es stellt lediglich Anforderungen an die umliegenden Adapter in Form von Java-Interfaces zur Verfügung.

Aufgabe der äußeren Schichten ist es nun, diese Schnittstellen zu

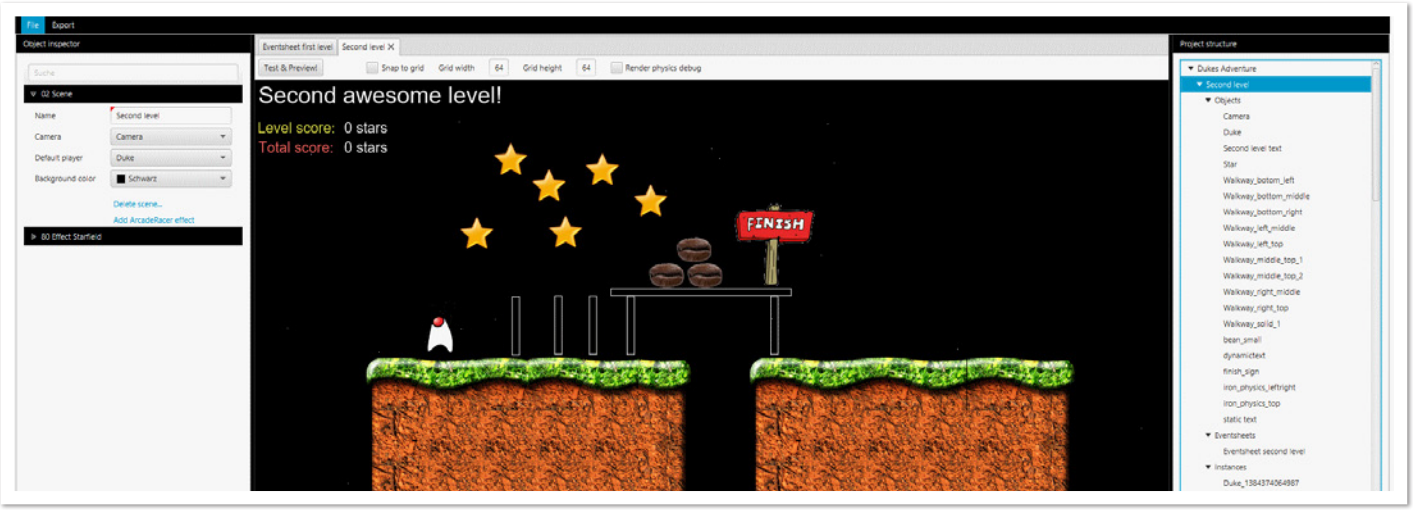

Abbildung 3: JavaFX Editor mit geladenem Level

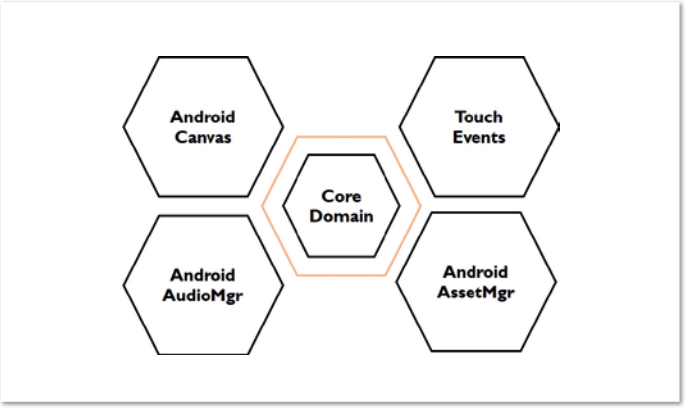

Abbildung 4: Die Android-System-Architektur

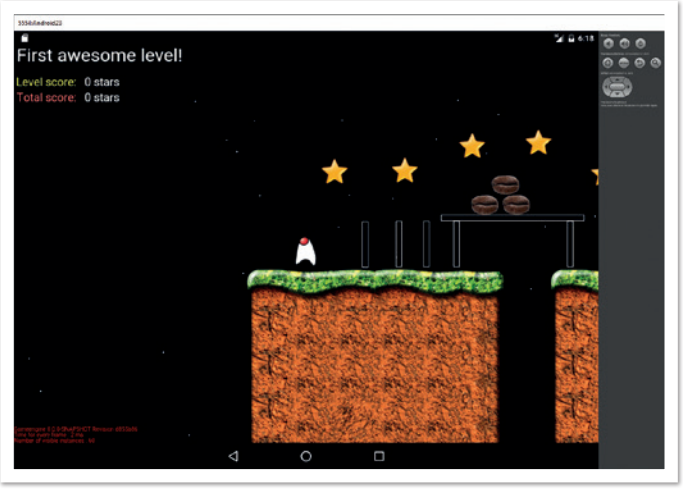

Abbildung 5: Das Spiel auf Android

bedienen. Somit ist die Implementierung der Physik-Simulation durch das Interface zwar abhängig von der Core-Domain, die Core-Domain ist jedoch nicht abhängig von der Implementierung. Beim Starten des Systems werden dann die einzelnen System-Komponenten über Dependency Injection verbunden. Mit dieser System-Architektur machen wir uns nun auf die Reise.

Ankunft im JavaFX-Land

Eine erste Etappe der Reise war ein lauffähiges Spiel, das JavaFX für die Visualisierung nutzt. Für die Eingabe sollte die Tastatur zum Einsatz kommen, die Darstellung erfolgt über eine Canvas-Komponente. Für dieses Reiseziel standen gleich mehrere Probleme an. Es musste natürlich die Core-Domain für alles implementiert werden. Zusätzlich waren aber auch die umliegenden Adapter der hexagonalen Architektur für die JavaFX-Umgebung zu implementieren. Außerdem musste natürlich noch das eigentliche Spiel konzipiert werden, das dann natürlich auch gespielt werden kann. Der Autor musste also noch einen Editor für die Core-Domain bauen. Hier zeigte sich ein großer Vorteil der hexagonalen Architektur, denn die Laufzeitumgebung-unabhängige Core-Domain kann sowohl innerhalb eines Editors als auch im tatsächlichen Spiel verwendet werden. *Abbildung 2* zeigt, wie die hexagonale System-Architektur aussieht.

Natürlich musste auch das Spiel konzipiert werden. Hier entschied sich der Autor für ein einfaches Jump-and-Run-Spiel mit dem Java-Duke. *Abbildung 3* zeigt die Ergebnisse der Arbeit – der Editor mit einem geladenen Level des Spiels. Im Spiel soll der Duke, gesteuert durch den Spieler über die Tastatur, Sterne sammeln, auf Bohnen herumhüpfen und über die Ziel-Flagge ins nächste Level springen.

Weiter nach Android-Land

In der nächsten Etappe der Reise soll das zuvor erstellte Spiel auf eine andere Laufzeit-Umgebung portiert werden, in diesem Fall Android. Wie kann das funktionieren? Zum Glück können Java-Programme auch für Android kompiliert werden, die komplette Core-Domain ist also wiederverwendbar. Aber wie sieht es mit den umliegenden Adaptern aus? Ziel der hexagonalen Architektur ist eine hohe Modularisierung und damit auch Adaptierbarkeit. Für Android waren also neue Adapter erforderlich.

Im Beispiel wird der Android Canvas für die Anzeige, der Android AudioManager für die Soundausgabe und der Android AssetManager für das Laden von Game Assets wie PNGs etc. verwendet. Neu für Android ist ebenfalls, dass es keine Tastatur im eigentlichen Sinne mehr gibt; die komplette Benutzer-Interaktion erfolgt über Touch-Events. Wir brauchen also noch einen Adapter, der diese Touch-Events in ein Format übersetzt, das von der Core-Domain verstanden werden kann. Das Diagramm in *Abbildung 4* zeigt die System-Architektur für Android und in *Abbildung 5* ist ein Screenshot für das Spiel zu sehen, das im Android-Emulator läuft.

Eine Zwischenbilanz

An diesem Punkt der Reise zieht der Autor eine kleine Zwischenbilanz. Wie weit war die Planung erfolgreich? Was hat funktioniert? Was nicht? Möchte er die Reise fortsetzen oder doch lieber an diesem Punkt abbrechen?

Als Fazit lässt sich auf jeden Fall sagen, dass die Wahl eines Computerspiels für ein Testbed eine sehr gute Idee ist. Die Portierungsanforderungen erzwingen geradezu eine Architektur im hexagonalen Stil. Ohne diese wäre eine Migration der Anwendung von JavaFX auf Android nicht so einfach möglich gewesen. Auch Java als plattformunabhängige Programmiersprache ermöglicht natürlich eine schnellere Migration auf andere Geräte. Auch hier war die Wahl aus Sicht des Autors richtig. Domain-Driven Design in Verbindung mit hexagonaler Architektur hat sehr positiven Einfluss auf die Gesamt-Architektur des Systems. Er möchte die Reise auf jeden Fall fortsetzen.

Das Internet-Land betreten

Das Computerspiel läuft bereits auf dem Desktop und auf mobilen Android-Geräten. Nun soll es auch als Webseite zur Verfügung stehen. Hier gibt es die erste grundlegende Änderung. Ein Browser versteht nur HTML, CSS und JavaScript. Ohne Erweiterungen lassen sich im Browser keine Java-Programme starten. Gerade diese Erweiterungen stellen ein Problem dar – sie sind entweder nicht installiert, aus Sicherheitsgründen deaktiviert oder schlichtweg veraltet. Es muss also ein anderer Weg gefunden werden, um das Spiel im Browser auszuführen.

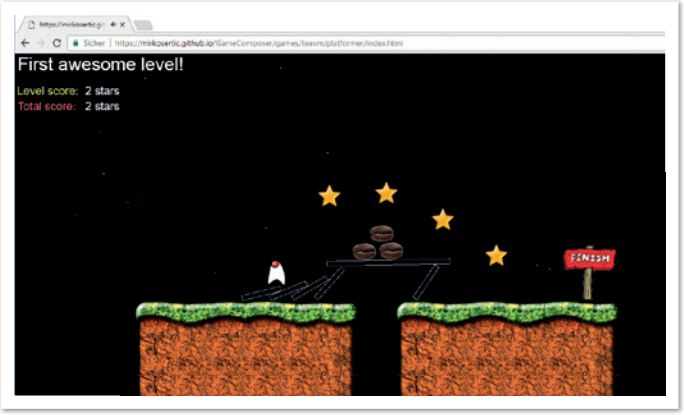

Abbildung 6: Spiel im Browser

Eine Option wäre natürlich, alles in JavaScript komplett neu zu schreiben. Dies ist aus technischer Sicht zwar möglich, aus wirtschaftlicher Sicht allerdings nicht machbar. Es würde redundanter Quellcode entstehen, der getrennt gewartet und weiterentwickelt werden muss, also entsprechender Aufwand. Zudem besteht das Risiko, dass die beiden Codebasen früher oder später auseinanderdriften.

Welche Möglichkeiten gibt es also noch? Hier betreten interessante Technologien die Bühne – sogenannte „Cross-Compiler". Diese können bestehende Programme von einer Sprache in eine andere übersetzen. Wir brauchen also einen Cross-Compiler, der Java nach JavaScript übersetzen kann. Als mögliche Lösungen kann der Autor zwei ganz besonders hervorheben: GWT und TeaVM.

GWT (Google-Web-Toolkit) ist ein Java-zu-JavaScript-Cross-Compiler und kann bestehenden Java-Quellcode nach JavaScript kompilieren. Für unser Computerspiel sind also neue Adapter zu bauen, die HTML5-Canvas für die Darstellung oder das Audio-API für die Soundausgabe nutzen. Diese lassen sich dann mittels GWT-Compiler nach JavaScript übersetzen.

TeaVM ist ein JVM-Bytecode-zu-JavaScript-Cross-Compiler. Anders als GWT ist kein Java-Quellcode für die Übersetzung notwendig, sondern JVM-Bytecode. Mit TeaVM kann man also auch andere Sprachen, die zu Bytecode kompiliert sind, nach JavaScript übersetzen. Genau wie für GWT sind hier die Adapter der hexagonalen Architektur an die HTML5-APIs anzupassen.

Für die weitere Reise entschied sich der Autor nach gründlicher Evaluierung für TeaVM. Gründe dafür waren, dass es für GWT schon länger kein größeres Major-Release mehr gab und dass TeaVM der erste Cross-Compiler war, der auch Java 8 nach JavaScript übersetzen konnte. Ein sehr starkes Argument war auch, dass der TeaVM-Compiler schneller als der GWT-Compiler war, bei teilweise besserem Laufzeitverhalten, Speicherverbrauch und Größe der entstandenen JavaScript-Datei. *Tabelle 1* zeigt die Metriken beim Vergleich der beiden Technologien. Für unsere weitere Reise verwenden wir TeaVM. Als Ergebnis erhalten wir unser Computerspiel lauffähig im Browser *(siehe Abbildung 6)*.

Bei der Evaluation von GWT und TeaVM hat sich eine weitere interessante Erneuerung herausgestellt. Die Implementierung eines HTML5-Canvas-Rendering-Adapters für die hexagonale Architektur war bei beiden Technologien fast identisch. Bei dieser Arbeit hat sich der Autor die Frage gestellt, ob diese Mechanik nicht auch mit WebGL in Verbindung mit GPU-Beschleunigung abbildbar ist. Es wäre natürlich sehr mühsam, die WebGL-Shader und die gesamte Infrastruktur für Textur-Management etc. von Hand zu schreiben, hier gibt es schon fertige Lösungen. Eine davon ist Pixi.js. Damit kann

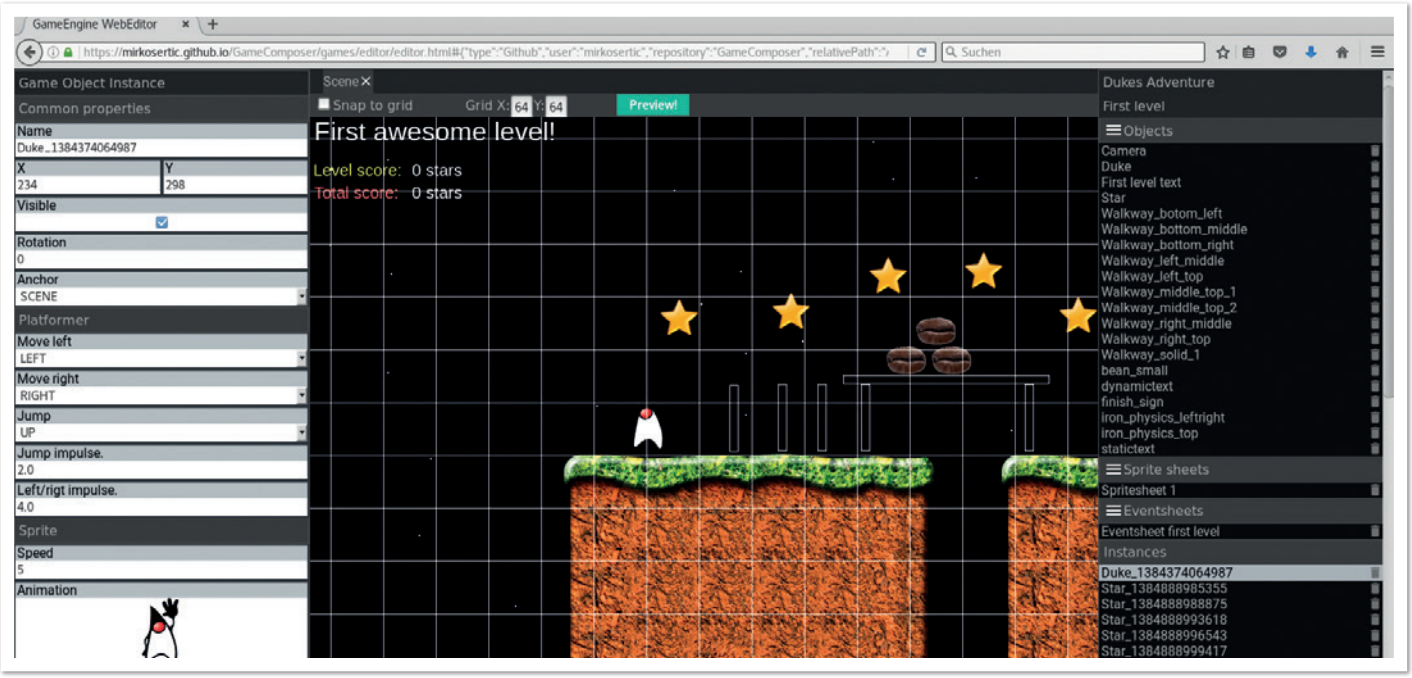

Abbildung 7: Die IDE im Browser

Metrik	GWT 2.8.0	TeaVM 0.4.3
Laufzeit des Compilers	43 Sekunden	12 Sekunden
Größe der JS-Datei	1.540 KB	1.013 KB

Tabelle 1

man einen komplexen Scene-Graphen bauen, der dann entweder WebGL beschleunigt oder über einen Canvas-Fallback ausgegeben wird. Dies ist geradezu ideal für unseren Anwendungsfall. Hier zeigt sich wieder einmal sehr deutlich, wie einfach eine hexagonale System-Architektur die Adaptierung an andere Umgebungen macht.

Fast am Ziel

Auf der bisherigen Reise hat der Autor erlebt und gezeigt, wie eine Java-Anwendung auf dem Desktop, auf mobilen Android-Geräten und als Web-Anwendung zur Verfügung gestellt werden kann, ohne dass die eigentliche Kernlogik neu geschrieben oder angepasst werden musste. Auf dieser Reise ist aber auch ein kleiner Teil leider noch in der alten Welt geblieben, und zwar die JavaFX-basierte IDE für das Spiel. Nun soll auch diese als Web-Anwendung zur Verfügung gestellt werden. Es gibt Projekte wie JPro, mit der JavaFX-Anwendungen ins Web portiert werden können. Diese erfordern aber teilweise Backend-Logik.

In unserem Anwendungsfall möchten wir allerdings eine Offline-Webanwendung bauen, ohne Verbindung zu irgendeiner Backend-Logik. Zum Glück lässt sich der bestehende Quellcode via TeaVM zu JavaScript kompilieren. Dies geht leider nicht für JavaFX-Komponenten, dieser Teil ist also, der hexagonalen Adapter-Philosophie folgend, neu zu schreiben. Hier stellt sich wieder die Frage: Mit welcher Technologie? Da JavaFX bereits komponentenbasiert ist, möchte man auch für die neue HTML-Anwendung ein komponentenbasiertes System einsetzen. Für den weiteren Weg der Reise entschied sich der Autor für die Evaluierung von HTML5-WebComponents zusammen mit dem Polymer-Framework. Nachdem die entsprechenden Adapter neu geschrieben waren, stand auch die IDE als Web-Anwendung bereit *(siehe Abbildung 7)*.

Hinter den Kulissen arbeitet hier ein interessanter Technologie-Stack. Die komplette Anwendung ist in Java geschrieben und wurde mit TeaVM zu JavaScript übersetzt. Die Anzeige ist mit Pixi.js über WebGL geregelt. Die einzelnen Anzeigebereiche sind über teilweise verschachtelte Web-Components gegliedert. Der aktuelle Zustand der Spielebeschreibung ist in der IndexedDB des Browsers gespeichert und kann bei Bedarf in ein GitHub-Repository übertragen werden.

Die GitHub-Integration war ein notwendiger Schritt für die Game-IDE, da der Zustand des Spiels ja irgendwo gespeichert werden muss und ein JavaScript im Browser bedingt durch die JavaScript-Sandbox nicht auf das lokale Dateisystem zugreifen kann. GitHub stellt ein gutes JavaScript-API zur Verfügung. Die Integration war daher sehr einfach möglich und bietet für unseren Anwendungsfall eine echte Bereicherung.

Am Ziel und darüber hinaus

Die bisherige Reise war eine interessante Erfahrung. Seit Beginn der Reise, die schon mehrere Jahre zurückliegt, benutzt der Autor dieses Testbed immer wieder, um neue Technologien zu prüfen, oder einfach nur, um ein wenig zu spielen und Abwechslung vom Alltag zu bekommen. Er hofft, Interesse für Themen wie Domain-Driven Design und hexagonale Architektur wecken zu können. Natürlich hofft er auch, dass am Beispiel seines Testbed die theoretischen Vorteile auch in der Praxis bewiesen werden konnten.

Was gibt es noch zu sagen? Die Reise wird weitergehen. Am Horizont sind schon neue, interessante Technologien zu erkennen, die geprüft und untersucht werden müssen. Neuronale Netzwerke können beispielsweise für Muster- und Gestenerkennung benutzt werden, mit Erweiterung natürlich auch für komplexe Planungsaufgaben. Die KI von Computerspielen stellt hier ein sehr breites Aufgabenspektrum zur Verfügung.

Eine andere Technologie ist WebAssembly. Es ist neu in Form eines MVP in allen aktuellen Browser-Versionen verfügbar und die Weiterentwicklung von asm.js. WebAssembly ist Bytecode für das Web. Der hier aufgezeigte Transpiler TeaVM unterstützt nicht nur die Kompilierung von JVM-Bytecode nach JavaScript, er bietet auch experimentelle Unterstützung für WebAssembly. WebAssembly bietet neben einem kompakteren Format noch den Vorteil einer effizienteren Laufzeitumgebung und kann als Ergänzung zu bestehenden Web-Anwendungen für komplexere Berechnungen wie Physik-Simulationen genutzt werden.

Wo die Reise enden wird, lässt sich im Moment nicht sagen. Der Autor hofft, auch weiterhin viel Spaß mit seinem Technologie-Testbed zu haben und auch zukünftig viele neue Themen damit ausprobieren und verwirklichen zu können. In diesem Sinne: Schöne Reise.

Mirko Sertic
mirko@mirkosertic.de

Mirko Sertic ist Software Craftsman im Web/eCommerce-Umfeld. In Funktionen als Software-Entwickler, Architekt und Consultant in Projekten in Deutschland und der Schweiz sammelte er Erfahrungen mit einer Vielzahl an Frameworks, Technologien und Methoden. Heute arbeitet er als System-Analyst bei der Thalia Bücher GmbH in Münster mit Schwerpunkt auf Java, eCommerce und Integrations-Technologien. Seine Freizeit verbringt er mit seiner Freundin, seiner Familie und hin- und wieder mit Open-Source-Projekten.