Projektskizze Hiconic

Kontext

Wir wollen erforschen, welche Auswirkungen ein neuartiges homoikonisches [HIK] Programmierprinzip (vgl. Maschinensprache oder LISP) in einer modernen, typsicheren, objektorientierten, selbst-reflektierten und heterogenen Software-Infrastruktur auf Wiederverwendbarkeit, Stabilität, Flexibilität und Sicherheit von Software-Systemen hat. Die längerfristige Vision unseres Forschungsvorhabens ist es mittels Gödelisierung (d.h. vollständig normalisierte Darstellung von Operationen und Operanden) [GÖD] neuartige, sich selbst reflektierende und über Netzwerke verteilte Software-Systeme zu entwickeln, die sich selbst, z.B. mit Methoden der Modell- und Parameter-Optimierung aus der Physik und Datenanalyse, analysieren, verbessern, anpassen und weiterentwickeln können. Dies schließt KI Methoden mit ein. Eine unserer Grundannahmen ist es, dass Daten und Funktionen gleichartig repräsentiert werden können. Wir nennen dies Modell-Orientiertheit und es entspricht einem ersten notwendigen Schritt hin zu vollständiger Homoikonizität. Eine Konsequenz der Modell-Orientiertheit ist, dass Funktionsaufrufe und Algorithmen, exakt so wie Daten, transportiert und behandelt werden können. Des Weiteren bildet dies eine ideale Voraussetzung für fundamental verstandene Modularität, die direkt auf solchen Modellen aufbaut. In Modulen können auf absolut natürliche und logische Art Separation-of-Concern (SoC) [SOC] und Inversion-of-Control (IoC) abgebildet werden. Wir entwickeln diese Prinzipien konsequent weiter und wollen eine flexible, auch zur Laufzeit mögliche, runtime Injection-of-Concern (RIoC) erreichen. Solcherart implementierte Software zeichnet sich durch minimale konkrete Technologie/Protokoll-Abhängigkeit aus und bleibt ideal erweiterbar – auch über Systemgrenzen hinweg (Cloud-Server-Mobile-Frontend). Wir verfolgen dabei einen bionischen Ansatz, denn die Natur lehrt uns, dass Modularität und die Vorteile der dynamischen Vernetzung allgemeiner und spezifischer Problemlösungskompetenz evolutionär der richtige Ansatz sind; z.B. Großhirn und Kleinhirn. Wir wollen eine ähnliche Plastizität für Software anbieten. Durch eine erweiterte Auffassung von Generizität wollen wir neue (Meta-) Muster (pattern) [PAT] in Software entdecken und optimieren.

Thema und Zielsetzung des Vorhabens

Das Projekt baut auf bereits existierenden selbst erarbeiteten Grundlagen auf (siehe Stand der Wissenschaft und Technik) und kann in drei Schritte unterteilt werden, die strukturell aufeinander aufbauen. Im ersten Schritt soll unser neues Paradigma von der aktuellen Java-Umgebung entkoppelt werden. Das bedeutet, die Modellartigkeit wird eigenständig als Grammatik formalisiert. Damit können Modelle und Modell-Zusammenhänge abgebildet werden, sodass damit normalisiert Programmcode, Datenstrukturen, Verknüpfungen und Abhängigkeiten beschrieben werden können. Die Leistungsfähigkeit der Grammatik kann jederzeit durch das Einbinden neuer, verbesserter und angepasster Modelle erweitert werden. Als Zwischenziel soll so die Notwendigkeit für Java zur Modellierung wegfallen. Allerdings werden vorerst die eigentlichen Algorithmen weiterhin in Java programmiert und angebunden. Im eng damit verbundenen zweiten Schritt werden wir eine erste Verallgemeinerung hin jenseits von Java anbieten. Diese soll in Typescript stattfinden. Da Typescript eine ideale Sprache für Client-Systeme ist, z.B. Web-Browser, wird diese Implementierung eine nahtlose Kommunikation über Netzwerke zwischen diversen Server- und Client-Systemen in einer Mischung aus Java und Typescript ermöglichen. Im Zentrum solcher Systeme stecken dann die Verarbeitung und der Austausch modellierter Daten und Operationen. Alle modellierten Teile dieses Systems werden durch die zuerst entwickelte Grammatik beschrieben, sowohl auf Server- als auch auf Client-Seite inklusive der transportierten Daten. Im dritten Schritt wird ein Asset Marktplatz entwickelt,

welcher nativ mit Hilfe der Grammatik modelliert und dann als Plattform-Projekt aufgebaut wird. Der Marktplatz soll zum Austausch von Modellen und modellierten Zusammenhängen dienen. Benutzer können Assets hochladen, welche selbst modelliert sind, und damit von allen Verallgemeinerungs- und Integrations-Vorteilen profitieren. Zum Beispiel können Assets jederzeit mittels Metadaten ergänzt und erweitert (z.B. Lizenzen, Berechtigungen, Bewertungen, etc.) sowie dann mit anderen Assets zusammen in komplexe Projekte integriert werden. Die wichtigsten Assets werden wir selbst über den Marktplatz frei anbieten. Benutzern können weitere oder bessere Assets frei oder kostenpflichtig über den Marktplatz anbieten. Der Marktplatz wird an ein ganzes Bündel von Entwicklungs-, Integrations- und Laufzeitumgebung angeschlossen sein und wird somit ein starkes Werkzeug für Community Enablement. Als Entwicklungsumgebung werden mindestens Eclipse und IntelliJ inklusive dedizierter Plugins, sowie ein zugrunde liegendes Build- und Test-System angeboten. Die Integration wird typische Entwicklungsprozesse mit git umfassen und darauf aufbauend diverse Continuous Integration und Continuous Deployment Services unterstützen. Als zentraler Speicherort für Code-Artefakte wird der Asset Marktplatz verwendet. Als Laufzeitumgebungen sollen mindestens native Anbindungen an tomcat Server, docker Container, Datenbanken und die wichtigsten Cloud Systeme unterstützt werden. Die Neuartigkeit unserer Vorschläge erfordert, dass Präsentation/Outreach ein Kernelement dieses Vorhabens und unseres zukünftigen Erfolges ist. Unsere Technologie muss als frei zugänglicher (Open Source) evaluierter Prototyp soweit entwickelt werden, dass sie für externe Entwickler hochattraktiv wird und diese beginnen, sich damit produktiv zu beschäftigen. Mit diesem Projekt geben wir neue Impulse und bieten Lösungsansätze für eine Vielzahl der im Feld SWS existierenden Problemkategorien.

Stand der Wissenschaft und Technik

Wir werden in einer Umgebung mit zunehmendem Entwicklungsdruck auf die Software-Industrie mit schnell steigenden Anforderungen, angefacht durch immer schnelleren Technologie/Protokoll-Wechsel, immer verteiltere, parallelere sowie diversere Systeme, immer mehr technologischen sowie staatlichen Compliance Vorgaben neuartige Lösungen aufzeigen. Nur in durchgehend normalisierten Systemen, wie wir sie entwickeln wollen, kann der Zuwachs von Information und Komplexität sogar zur Verringerung von “Speicher” führen, da durch das Erkennen von übergreifenden Zusammenhängen und Mustern vereinfacht und komprimiert werden kann. Wesentliche ungelöste Probleme in komplexen und skalierbaren Systemen betreffen die generelle Wartbarkeit sowie die langfristige Offenheit für Erweiterungen; Viele Sicherheitsprobleme, Kostenfallen und typische unflexible terminale Projekt-Zustände liegen darin begründet. Heutzutage gehen immer mehr Ressourcen der Software-Industrie in der immer wiederkehrenden Neu-Lösung gleicher/ähnlicher Probleme in neuer Problem-spezifischer Form verloren. Es wird versäumt, das Potenzial Domain-unspezifischer (generischer) Lösungen auszuloten, wodurch sich eine neue Meta-Ebene der unspezifischen Gemeinsamkeit scheinbar unzusammenhängender Probleme ergeben wird. Oft wird aus Gründen der Performance auf spezifische Lösungen zurückgegriffen. Die von uns vorgeschlagene modulare Reflektiertheit, von der Ebene der Software-Abhängigkeiten bis hin zu konkreten (generischen) Algorithmen, bietet hier fast schon offensichtliche Lösungsoptionen: Wir stellen die Technologie zur Verfügung, um eine Abwägung von generischen zu spezifischen Teil-Lösungen jederzeit zu überdenken und an die Problemstellung anzupassen. Daher ist es wichtig, die Leistungsfähigkeit generischer Lösungen gezielt zu verbessern. Performance darf nicht der primäre Grund sein, auf Wartbarkeit/Generalität zu verzichten. Dies ist eines unserer zentralen Ziele. Der von uns vorgeschlagene Ansatz ist fundamental unterschiedlich von anderen Strategien und in seiner Klarheit momentan einzigartig. Es ist unser Ziel, unseren Ansatz frei zugänglich anzubieten und wissenschaftlich sowie wirtschaftlich auszuwerten. Wir sind davon überzeugt, dass dadurch neue und

dringend benötigte Impulse für die zukünftige Softwareentwicklung augehen können. Das hier beantragte Vorhaben basiert auf vielen Vorarbeiten und einem langen Lernprozess. Die Antragsteller haben im Rahmen kommerzieller Software-Erstellung über einen Zeitraum von über 15 Jahren die Konzepte erarbeitet und immer weiter verfeinert. Die dabei entstandene Kern-Plattform hat in diversen konkreten Projekten erfolgreich demonstriert, dass sie exzellentes Potential für die hier vorgeschlagenen Vorteile bietet. Diese Technologie basiert aktuell auf Java macht vollen Gebrauch von polymorphen und typsicheren Zugriffen. Eine wichtige Grundlage ist das Meta-Modell, welches alle Modelle inklusive sich selbst beschreiben kann. Kontextsensitive und frei modellierbare Zusatzinformationen können auf allen Strukturebenen (Modell, Typ, Eigenschaft, Konstante) eingefügt werden. Das Meta-Model selbst sowie die Zusatzinformationen können jederzeit (d.h. auch "spät" zur Laufzeit) verändert werden. Dabei entsteht kein Performance Nachteil. Der Code ist in jedem Fall Java-Bytecode mit voller Leistungsfähigkeit. Diese Lösung entspricht einer verallgemeinerten Version von "aspect oriented programming" (AOP) [AOP], die mit "just-in-time" Kompilierung [JIT] kombiniert ist. Die so entstehende vollständig modellierte Reflektion von Typen und deren Eigenschaften erlaubt z.B. eine Traversierung auf Instanz- und Modellebene. Zugriffe auf Eigenschaften können abgefangen und beeinflusst werden, um "cross-cutting concerns" [AOP] zu verwirklichen. Dieser Ansatz ist hochgradig Objekt/Typ-orientiert, führt aber in der Entwicklung von Algorithmen direkt zu Mustern des „functional programming" -Paradigmas, in dem Algorithmen untereinander mittels „Request" und „Response" kommunizieren. Ein darauf aufsetzendes Manipulations-Modell erlaubt es inkrementelle Änderungen an Eigenschaften eines jeden Modells zu registrieren, zu speichern, zu wiederholen und zu synchronisieren. Damit entsteht eine Event-Source [EVS] Repräsentation, in der die gesamte Historie gespeichert ist. Wichtige generische Algorithmen für typische Funktionalität, wie Datentransport, Datenbanken/Persistenz (SQL, in-memory), Traversierungsalgorithmen (Cloning, Merging, Synchronizing, Marshalling) existieren schon. Ein Query-Model vereinheitlicht den Zugriff auf verschiedenartig persistierte Daten.

Arbeitsplan, Arbeitspakete (AP)

AP1) Eigenständige Formalisierung des Paradigmas und Entkopplung von Java

Der erste Schritt unseres Projektes ist es, das Modell-Paradigma durch eine Grammatik zu formalisieren und dadurch semantisch von Java zu entkoppeln. Die Grammatik wird mit Hilfe von ANTLR4 entwickelt. Sie soll Modelle, Typen, Eigenschaften, Initialisierung und die Zusammenhänge zwischen Modellen beschreiben können. Darüber hinaus müssen auch Metadaten-Assoziationen abgebildet werden. Von Anfang an wird auch auf die Beschreibung von Software-Abhängigkeiten (dependencies) und Software-Versionierung Wert gelegt. Die Grammatik wird unser Paradigma an seinem zentralen Punkt (Modell) schärfen. Langfristig wird dies zu einer neuen, modell-orientierten Programmiersprache führen. Die Modellgrammatik erfordert zuerst die Entwicklung von Regeln für Lexer und Parser. Darauf folgt die Implementierung des Parsers in Java. Ziel des Parsers wird es sein eine Repräsentation des verarbeiteten Inputs als (natives) Metamodell zu erzeugen. Dieses Metamodell steht im Zentrum des AP1. Das Metamodell kann verschiedenartig weiterverarbeitet werden. Das erste Ziel ist eine Serialisierung mit Ausgabe als Java Sourcecode. Das nächste Ziel ist die direkte Kompilierung zu Java Bytecode. In weiteren Schritten wird die Serialisierung erweitert, damit auch Typescript und später andere Target-Sprachen/Systeme unterstützt werden. Es ist uns besonders wichtig, moderne Entwicklungsmethoden durch Code-Auto-Completion und Syntax-Highlighting anzubieten. In einem ersten Schritt wollen wir dies für mindestens eine der großen IDEs prototypisch anbieten (Eclipse, IntelliJ oder VSCode). Das gilt sowohl für die Programmierung mit der neuen Grammatik, als auch für die Programmierung mit den erzeugten Produkten der Serialisierung der Grammatik im Rahmen weiterführender Projekte.

AP2) Portieren in andere Sprachen und Anwendungen

Analog zu Serverkomponenten im Backend sollen auch Tribefire-Frontends modularisiert und modelliert werden. Dafür muss eine entsprechende UX Trägerplattform für Typescript-Module entwickelt werden. Die Trägerplattform ist ausdrücklich nicht nur für die Anzeige und User-Interaktion, sondern auch für universelle Arbeitsvorgänge auf Client-Geräten gedacht. Dafür wird eine identische Modellierung wie sie auf der Server-Plattform existiert (Service Processing) benutzt. Ein solches Setup kann auch mit Hilfe von node.js direkt als "Lightweight Server" verwendet werden. Längerfristig ist es erwünscht, auch anderer Sprachen/Systeme nativ zu unterstützen. Dadurch können Sub-Probleme jeweils mit optimaler Technologie gelöst und dann mit Hilfe von Tribefire-Plattform-Vernetzung verknüpft werden. Für die UX Plattform sollen dedizierte, optimierte und Domain-agnostische (generische) UI) Komponenten entwickelt werden (z.B. Skalareingabefelder, Layout Panel, Property Panel, Tree/Grid Panel, Thumbnail Panel, Graph Viewer). Weiterhin benötigt die Plattform Komponenten für wesentlichen Kernthemen: Grafischer Modellierer, Metadata Editor. Alle modellierten Komponenten, auch UI Komponenten auf dem Client, sind immer durch einen automatischen Manipulation-Tracking Cross-Cutting Concern verbunden. Ultimativ soll eine generische "Explorer" Komponente erstellt werden. Diese steht dann für Business Modellierung zur Verfügung, kann aber ebenso für die allgemeine Administration der Plattform selbst verwendet werden.

AP3) Portable Entwicklungs-, Integrations- und Laufzeitumgebung, für Open Source Community, Produkte und Community Enablement

Eine benutzerfreundliche, portable Entwicklungsumgebung (SDK) wird entwickelt. Dieses wird mit minimalem Aufwand installierbar sein und bringt alle notwendigen Voraussetzungen für Tribefire-Projekte mit. Diese neue Umgebung wird der Standard für alle Tribefire-Projekte. Ein einheitlicher Standard wird das Onboarding und spätere Hilfestellungen extrem vereinfachen. Ein virtuelles Artefakt-Repository wird als native Tribefire-Plattform-Anwendung erstellt. Ein solches Repository kann Artefakte aus CI Pipelines speichern und wieder zur Verfügung stellen. Das virtuelle Repository erhält ein Explorer-Interface aus dem AP2. Ein großer Vorteil dieser Lösung ist, dass Artefakte auf diese Weise modelliert werden und z.B. durch Metadaten weiter konfiguriert werden können. Im virtuellen Repository können einschränkende Views berücksichtigt werden, wodurch ein einfaches Release und Patch Management möglich wird. Ein virtuelles Repository kann als Backend verschiedene Speichersysteme verwenden. Das zentrales Ziel dieses Projektes ist es, den Prototypen einer als nativen Tribefire-Plattform-Anwendung erstellten Asset Marktplatzes zu implementieren. Ein "Asset" ist dabei die Verallgemeinerung eines Artefakts und kann Modelle sowie weitere Daten und Dateien beinhalten. Ein solcher Marktplatz kann Assets/Artefakte sowohl von CI Pipelines, aber auch von anderen Quellen verarbeiten. Auf einem Marktplatz können Assets durch Benutzer oder automatisiert gelistet, sortiert, gesucht, kombiniert und heruntergeladen werden. Assets können durch Anwender mit Hilfe von Metadaten erweitert werden, um z.B. Bewertungen und Erfahrungen zu speichern. Der Marktplatz ist eine produktartige Weiterentwicklung mit vollständigem Benutzerinterface des einfachen virtuellen Artefakt-Repositories. Über diesen Marktplatz können Entwickler miteinander Modelle austauschen und daraus neue, komplexe Produkte erzeugen. Eine modellierte Abhängigkeits-Auflösung soll über das virtuelle Repository (d.h. auch über den Marktplatz) bereitgestellt werden. Durch eine transitive Serverseitige Auflösung, können Serveranfragen und damit Netzwerk- und Server-Ressourcen eingespart werden. Dies erhöht die Leistung, senkt Kosten und schont die Umwelt.