Technology Bridge

Object Engineering

Herausgeber: Object Engineering GmbH • www.objeng.ch • Februar 2020 • Nr. 1

Füsse im Beton, Kopf in der Cloud

Dr. Rheinhold Thurner, Andres Koch

Der technische Fortschritt ermöglicht neue innovative Lösungen – aber veraltete Legacy-Systeme hängen oft wie Kletten an den Firmen. Mit traditioneller Wartung oder Weiterentwicklung ist der Sprung aus der Abhängigkeit nicht zu schaffen. Wir zeigen, welche technischen Lösungen es gibt und welche Schwierigkeiten dabei überwunden werden müssen.

Das IT-Management und die Geschäftsleitung wünschen sich, dass ihre IT-Applikationen in der «Cloud» betrieben werden und die Flexibilität und mobile Nutzung «grenzenlos» sind. Währenddessen müht sich die operative IT damit ab, die Legacy aus dem Beton zu sprengen und dabei möglichst wenig Kollateralschaden (Unterbrechungen, Kosten, Zeitbudget) anzurichten.

Die Legacy-Falle

Es ist noch nicht so lange her, dass die «zentrale EDV-Abteilung» mit ihren Anwendungen die gesamten automatisierten Informationsdienstleistungen erbracht hat. Feste Rhythmen bestimmten die Prozesse – die «Tagesverarbeitung», die «Monatsauswertung», den «Jahresabschluss». Ähnlich verhielt es sich mit der Behandlung von neuen Anforderungen an die Systeme und die Warteliste der Anwender wurde länger. Derweil wuchsen meist dezentral und ausserhalb der Kontrolle der «EDV» dezentrale Anwendungen mit Hilfe von Mini-Computern. Regelmässiger Datenaustausch federte das Problem der Datenverfügbarkeit ab. Mit dem Erscheinen von PCs lösten sich die Anwender noch mehr vom «Rechenzentrum» und bauten mit Excel flugs und ohne Zeitverzug ihr oft als «Shadow-IT» apostrophiertes Öko-System. Die Folge – die Anwender lernten, schnell Anpassungen vorzunehmen, währenddessen die zentrale EDV in ihrer Komplexität erstarrte.

Wie die Praxis auch zeigt, liegt das hohe Risiko nicht nur bei den Applikationen, welche mehrmals im Monat die Aufmerksamkeit der System-Administratoren oder sogar der Entwickler erfordern. Nein, die scheinbar problemlos laufenden Applikationen können die höchsten Risiko-Faktoren darstellen.

Verborgen unter der Haube dieser problemlosen Applikationen entwickeln sich Unpässlichkeiten, Inkompatibilitäten, Seiteneffekte usw. und eines Tages schlägt «es» zu. Völlig unvorbereitet ist nach einem Release-Wechsel dann plötzlich «Feuer im Dach». Für «business as usual» ist es dann zu spät, und grosse Ausfalls- und Reparaturkosten treten auf. Die Gründe für das hohe Risiko bilden sich aus Mitarbeiter-Fluktuation (Stellenwechsel intern, extern, Rentenantritt), verflossenem Know-How, nicht mehr aktueller Dokumentation, wandelnden Technologien auf systemnaher Ebene, Ignoranz und fehlender Voraussicht, um nur einige zu nennen. Weiter sind auch Aussagen wie «never change a working system», «es läuft ja», oder Unterschätzung der Wichtigkeit der Applikations-Komponente im Geschäftsprozess Ursachen, dass man sich dann vor einem Scherbenhaufen wiederfindet.

Und aufgepasst, nicht nur Applikationen in COBOL, PL/1, Fortran, Pascal, C oder 4.-GL-Sprachen sind Legacy (Altsysteme), sondern bereits seit Längerem gehören auch Applikationen in C++, Java und der Teen-Sprache C# und weitere «moderne» Sprachen dazu.

Applikationen, in welcher Form auch immer, über mehr als 10 Jahre erfolgreich zu betreiben, heisst, organisiert zu sein und die vorbeugende Wartung (preventive maintenance) der Softwaresysteme zu beherrschen.

Der CIO löst den EDV-Chef ab

Nicht die Programmiersprache oder ein technisches Detail, sondern das Zusammentreffen einer Vielzahl von Verantwortungen, Vorgehensweisen und technischen Lösungen ist das «Legacy-Problem» und damit primär auch ein Organisations-Problem. Der CIO sollte es richten, die Gesamt-Architektur gestalten, (1) obsolete Systeme ausmustern, (2) die Produktion schlank und zuverlässig im Griff behalten und (3) innovative Ansätze kontrolliert in die Produktion überführen. Den technischen Fortschritt beherrscht man nur, wenn man alle drei Bereiche im Griff hat und insbesondere mit Augenmass Obsoletes ausmustert und Neues erprobt.

ITAM (IT Asset Management)-Konzepte geben klare Richtlinien (z.B. COBIT Managing Technological Innovation) dafür, wie eine angemessene Bandbreite der Technologien und Tiefe (Veralterung) verwaltet werden soll.

Legacy-Systeme – besser als ihr Ruf?

Den Legacy-Systemen kann man zweifellos viel vorwerfen, man muss aber auch anerkennen, dass sie meist Ordnung in den Daten und der Datensicherung haben. Sie haben Anwendungen aus Programmen, die ein kompaktes Wissen an Verfahren enthalten. Daten und Prozesse sind minutiös im Programmcode beschrieben. Auf die Frage: «Wie funktioniert bei Ihnen das Umlage-Verfahren» erhielt einer der Autoren einmal als Antwort: «Da müssen Sie den Herrn Halske fragen, der hat das programmiert». Leider war Herr Halske bereits in Pension und nur das Programm konnte Auskunft erteilen. Fazit: In Legacy-Systemen steckt nach dem zweiten Blick mehr, als man auf den ersten Blick meint. Nur zu oft wird der Fehler gemacht, auf den zweiten Blick zu verzichten und damit eine Refakorierung einer Software erst gar nicht in Betracht zu ziehen.

Refaktorierung im Gegensatz zu Software-Wartung

Neuen Anforderungen begegnen wir in der Regel mit «Softwarewartung» – dem klassischen Ansatz dafür, lokale Anpassungen an einzelnen Software-Komponenten vorzunehmen. Dieser versagt jedoch,

Copyright © 2020 by Object Engineering GmbH CH-8142 Uitikon-Waldegg

wenn systematische Änderungen an einer Vielzahl von Komponenten gleichzeitig durchzuführen sind. «Softwarewartung» ändert ein Element nach dem anderen und führt es in die Produktion ein. Refaktorierung hingegen baut eine «Umstellungsmaschine» auf, die in einem Anflauf alle Elemente gleichzeitig ändert – ein entscheidender Vorteil, weil die Produktion und Wartung parallel ungestört weiterlaufen können.

Wie das geschehen soll ist kein Geheimnis: CAAR (Computer Assisted Analysis and Refactoring) zeigt Lösungen auf, wie Systeme und Technologien über Lebenszyklen hinweg professionell zu verwalten sind. Die organisatorischen Konzepte von ITAM und CAAR bilden zusammen die Grundlage für die kontinuierliche Innovation von IT-Systemen.

Die Umstellungsmaschine für Software-Refaktorierung

Eine Umstellungsmaschine kann man sich durchaus als ein riesiges Batch-Programm vorstellen, das die Software einliest, analysiert, transformiert und die korrigierte Software ausspuckt.

Das in **Abbildung 1** gezeigte Vorgehen ist für einfachere Refaktorierungs-Projekte durchaus geeignet. Diese modifizieren den Programm-Code und geben ihn unter Umständen sogar in einer neuen Sprache aus. Der generierte Code ist oft nicht einfach lesbar und möglicherweise gar nicht mehr wartbar. Entscheidend ist jedoch, dass das Design gleich bleibt. Für grundlegende Änderungen und Restrukturierungen ist dieses Verfahren nicht geeignet.

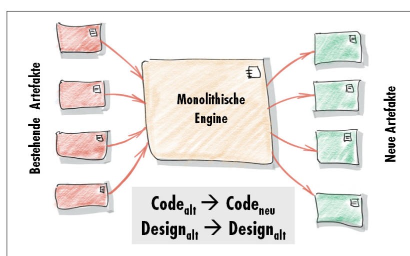

Abb. 1 Monolithische Software-Refaktorierungs-Maschine

Metadaten-zentrierte Software-Refaktorierungs-Maschine

Wir befassen uns mit anspruchsvollen Restrukturierungen und setzen dafür einen strukturierten Ansatz ein: Dedizierte Parser/Analysatoren befüllen ein Refaktorierungs-Repository. Dedizierte Generatoren erzeugen aus dem Repository die Zielkomponenten. Dies ermöglicht es, eine Vielzahl einfacher Werkzeuge zu einer kontrollierten sinnvollen Gesamtheit zu kombinieren.

Im Folgenden soll dieser Ansatz (**Abbildung 2**) etwas mehr im Detail aufgezeigt werden. Dazu beschreiben wir den typischen Prozess, der unabhängig von der Art von Legacy gültig ist.

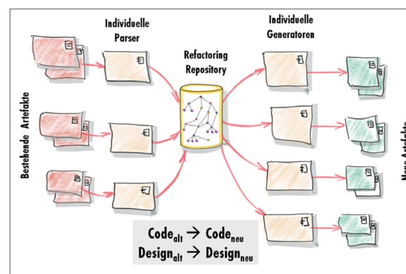

Abb. 2 Repository-zentrierter Ansatz

Refaktorierungs-Prozess

Der in der Praxis bewährte Prozess beinhaltet folgende Schritte:

1. Voranalyse und Machbarkeitsstudie (IT-Inventar, Zeit- und Kostenbudget einschätzen)
2. Test- und Verifikationskonzept für die Applikation vor der Migration (Test-Infrastruktur)
3. Zielarchitektur definieren
4. Spezifikation der automatisierten Migration (Refaktorierung)
5. Umsetzung der Automaten, Durchführung von Testläufen
6. Refaktorierungs-Läufe mit allen Eingangs-Artefakten
7. Integration der generierten Artefakte in Komponenten oder im Gesamtsystem
8. Verifikation der Funktionsweise mit der in (2) erstellten Test-Infrastruktur

Sowohl für die Voranalyse (1) wie auch für die eigentliche Refaktorierung (6) wird ein ähnlicher Prozess (**Abbildung 3**) angewendet:

- Import bestehender Artefakte
- Erkennung relevanter Aspekte (Code-spezifisch, oder System-spezifisch)
- Analyse und Transformation (meist nur bei der Refaktorierung)
- Generierung von Berichten oder neuer Code-Artefakte

Dieser Prozess wird iterativ wiederholt, bis das Repository alle Daten enthält, die zur bestmöglichen Refaktorierung respektive Generierung notwendig sind.

Import

Beim Import verwendet man Sprachen-Parser oder spezifische Filter, welche meist aus Programm-Quellcode (oder rückübersetztem Code), Skripten (Batch-Programmen), Konfigurations- oder Datenbank-Metadaten, strukturierter Dokumentationen oder individuellen manuellen Eingaben bestehen. In Ausnahmefällen, bei denen Programm-Quellencode fehlt, können auch zur Laufzeit aufgezeichnete Daten (z.B. mit Kommunikation zwischen Programmen) verwendet werden. Auch wenn typisch jeweils 80-90% der Programm-Quellen verfügbar sind, ist hier auch Ideenreichtum gefragt, welche Quelle genau einem das letzte Mosaik-Teilchen liefern könnte.

Detailerkennung

Ist der Import in einer ersten Iteration vollzogen, kann man die Daten mittels geeigneter Viewer-Programme inspizieren.

Nach einer ersten Iteration kann auf einer höheren Metadaten-Ebene auf der Basis von Applikation, User-Interface und Datenbank eine Art IT-Inventar erstellt werden. Diese Zusammenhänge können die weitere Analyse- und Refaktorierungs-Strategie unterstützen.

Für eine Refaktorierung sind nicht immer alle Aspekte wichtig. In der Regel weiss man bereits, was an den eingelesenen Metadaten relevant ist. Das können zum Beispiel Code-Stellen sein, die auf eine Datenbank oder auf bestimmte Tabellen zugreifen. Man tut immer gut daran, wenn man mit den Entwicklern (sofern diese nicht bereits in andere Projekte abgezogen sind) spricht und den Tenor untersucht, wie sie über den zu migrierenden Code sprechen. Ebenso ist bereits die Zielarchitektur relevant, welche definiert, wie man sich die migrierten Komponenten vorstellt.

Mit diesen gesammelten Informationen kann man spezielle Filter erstellen, welche genau die gewünschten und relevanten Aspekte in den importierten Artefakten erkennen und entsprechend mit Attributen versehen, welche in einem späteren Schritt für die Analyse, Transformation oder Generierung hilfreich sein können.

Analyse und Transformation

In der Analyse-Phase wird in der Regel eine Vielzahl unterschiedlicher Artefakte eingelesen und im Refaktorierung-Repository abgespeichert. Erst dann können, mit den im vorangehenden Schritt mit Attributen versehenen Aspekten, Komponentenübergrei-

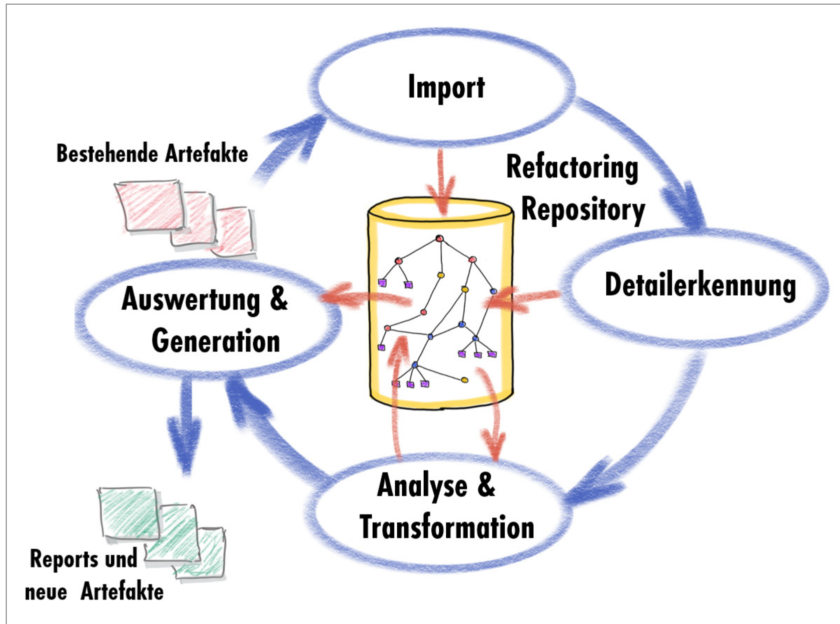

Abb. 3 Prozess für eine systematische Refaktorierung

fende Analysen und Transformationen getätigt werden. Dazu gehören der statische Aufbau beziehungsweise die Nachverfolgung möglicher Aufruf-Graphen, Abhängigkeiten von Modulen, Anomalien wie nicht verwendeter Code, Zugriffe auf bestimmte Module, Benutzereingabe-Masken oder Datenbanktabellen und vieles mehr.

Aus der Analyse gewonnene Erkenntnisse werden in diesem Schritt zusätzlich als Attribute oder Beziehungen ergänzt. Da für das Refaktorierungs-Repository eine Graphen-Datenbank [Rob15] verwendet wird, können die für Graphen bestimmte und im Datenbanksystem integrierte Standard-Algorithmen (z.B. stark verbundene Komponenten, Clusterkoeffizient, Modul-Dekomposition) verwendet werden. Diese sind zwar für Graphen gedacht, aber da die eingelesenen Artefakte nun als Graphen vorhanden sind, liegt es auf der Hand, diese Algorithmen [Nee19] zum Beispiel für die Aufspaltung eines Monolithen zu nutzen.

Nach einem oder mehreren Analyse- und Transformations-Schritten erhalten wir ein angereichertes Metadaten-Repository, welches nun noch mehr über den Aufbau der eingelesenen Komponenten und deren Beziehung untereinander aussagt. Dies kann natürlich mit dem geeigneten Viewer (Neo4J Browser) und geeigneter Abfragensprache (CypherQL) weiter untersucht werden (**Abbildung 4**). Mit jedem Schritt lernt man die zuvor unbekannte Software besser kennen. Das ist ja genau eine der Abhilfen, die nötig sind, wenn die ursprünglichen Know-How-Träger nicht mehr verfügbar sind.

Auswertung und Generierung

Der nächste Schritt setzt nun den Inhalt des Refaktorierungs-Repository auf Berichte oder neue Artefakte um.

In den frühen Iterationen können Reports mit groben Zusammenhängen generiert werden, welche zum Beispiel ein IT-Inventar auf Applikations-Ebene darstellen. Mengenstatistiken helfen, die weiteren Iterationen auf bestimmte Gebiete zu lenken oder die Migrations-Strategie besser auszurichten. Die Auswertungen können einen beliebig hohen Detaillierungsgrad haben, vorausgesetzt, dass dafür die Daten bereits ermittelt wurden.

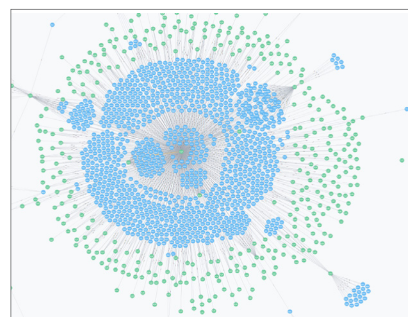

Abb. 4 Graphen Darstellung einer AS400 Batch-Applikation

In späteren Iterationen können dann durch Verfolgen der gespeicherten Graphen und mittels Steuerung durch die in den vorhergehenden Schritten gespeicherten Attribute neue Artefakte generiert werden. Die verschiedenen Generatoren haben einerseits die neue Architektur beziehungsweise den Aufbau der neuen Frameworks gekapselt und generieren aufgrund der Graphen-Information die neuen Artefakte. Zu generieren, ohne eine Zielarchitektur und ein Programm-Design zu haben, ist nicht sinnvoll. Die in der neuen Architektur verwendeten Frameworks (Bibliotheken) würden mit einer Eigenentwicklung abgedeckt oder aber auf andere Weise akquiriert (Einkauf, Open Source u.a.). Weiter ist es auch durchaus sinnvoll, einige wenige, später automatisiert zu generierende Artefakte in einem ersten Schritt manuell zu erstellen und zu testen. Das gibt die erprobten Spezifikationen für die Generatoren. Entwickler scheuen sich in der Regel vor generiertem Programm-Code, weil nach ihrer Erfahrung generierter Code schlecht lesbar, nicht verständlich und somit auch nicht erweiterbar und wartbar ist. Das mag für viel in der Vergangenheit generierten Programm-Code gelten. Heute orientiert sich Code-Generierung an einer klar spezifizierten Zielarchitektur. (**Abbildung 5**)

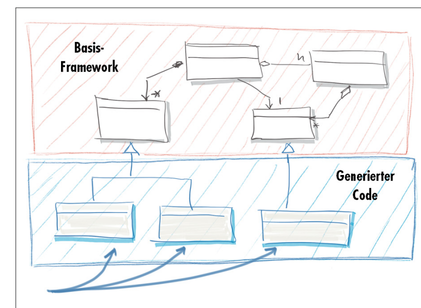

Abb. 5 Generierung in Ziel-Architektur

Beispiel: Von Fat-Clients zu einer Web-basierten Applikation

Bei der Migration einer ***Fat-Client***-Applikation zu einer Web-basierten Technologie wurde das Benutzeroberflächen-Design Technologie unverändert belassen, damit die rund 2000 Benutzer keine Umgewöhnung benötigten. Das technologische Design wurde hingegen unter Einsatz des Angular-Frameworks orthogonal [ThKoKo13].

Mit rund 130 Formularen und rund 10'000 Feldern und entsprechend vielen Feldern für die Messages in den Transaktionen war das Mengengerüst gross genug, um das obige Prinzip anzuwenden. Bei einem Zeitbudget von rund 1900 Stunden konnte nachweislich eine Einsparung von rund 50 Prozent nachkalkuliert werden. Darin eingerechnet war auch der Aufwand für die Erstellung des UI-Frameworks mit Angular, was auch bei einer manuellen Migration hätte aufgewendet werden müssen.

Object Engineering®

Beispiel: Architektur-Migration einer JEE-Applikation [WSRE19-1]
Kai-Uwe Hermann schildert in [WSRE19] eine Architektur-Migration einer Code-Basis von 1.5 Millionen Java-Codezeilen mit rund 6000 Klassen. Bei der spezifischen Detail-Erkennung mussten UI-Attribute und deren Code-Abhängigkeiten erkannt werden. Pro UI-Attribut wurde dann eine neue Klasse (rund 500 Klassen) erstellt und der abhängige Code in diese Klasse migriert. Die neuen Klassen konnten dann statt in einer Vererbungshierarchie in einer Regel-Engine zur Laufzeit verarbeitet werden, was das Ziel der neuen Architektur war. Die neuen Klassen wurden dann manuell integriert, was gleichzeitig auch eine Form von Qualitätskontrolle war.

Damit die Entwickler einfach zurück in den alten Code navigieren konnten, wurden Links eingebaut, welche auf Knopfdruck zum Ursprungscode navigierten. Solche einfachen Hilfsmittel haben den integrierenden Entwickler wertvolle Zeit eingespart und Nerven geschont. In diesem Projekt konnten rund 47 Prozent des Aufwand eingespart werden.

Bei den beiden in den Kästen vorgestellten sowie auch bei anderen Projekt-Beispielen wurde erkannt, dass es eine gewisse Menge an Artefakten braucht, damit sich der Aufwand für die Teilautomatisierung einer Refaktorierung lohnt. Der zweite zu berücksichtigende Aspekt ist der, dass Entwickler es nach einer bestimmten Menge an gleichen Refaktorierung-Schritten leid sind, was sich unvermeidlich auf die Qualität des Codes auswirkt. Die Refaktorierungs-Maschine hat als Roboter keine «Gefühle» und macht es immer gleich gut. Entwickler-Ressourcen sollten ja auch für kreative und nicht für repetitive Routine-Arbeiten eingesetzt werden.

Die Krux – die Entscheidungsfindung

Refaktorierung von Legacy-Software hat es oft schwer, einen Projektantrag durchzubringen. Refaktorierung ist ein sehr technisches Thema, oft mit scheinbar beschränkter Dringlichkeit und schwer darstellbarer Wichtigkeit. Funktionale Einzeländerungen besitzen meist einen Sponsor im Fachbereich, welcher Wichtigkeit und Dringlichkeit an den entscheidenden Orten zu vertreten weiss. Für Legacy-Renovation muss der Verantwortliche für «Managing Technological Innovation» das Sponsoring übernehmen. Dieser muss erst gefunden und dann «stufengerecht» informiert werden – sonst wird das Vorhaben aufgeschoben bis möglicherweise auch Refaktorierung nicht mehr helfen kann.

Man sollte sich zu guter Letzt auch die Gewissensfrage stellen: «Wie kann das Legacy-Problem in Zukunft vermieden werden?» Gerade in unserer schnelllebigen Zeit mit vernachlässigter Nachhaltigkeit gibt es kein Wunderrezept. Der «Königsweg» ist klare Verantwortung für Technologie-Entscheidungen und solides Engineering basierend auf einer Software- und Systemarchitektur, die auf lose gekoppelten und modularen Komponenten aufbaut. Mithilfe von Legacy-Engineering gelingt es, auch Legacy-Anwendungen so umzugestalten, dass sie für die Veränderungen der Zukunft gewappnet sind.

Ersterscheinung OBJEKTspektrum 06/2019

Die Autoren

Dr. Reinhold Thurner befasst sich mit Modellierung, Repositories und Generatoren.

Andres Koch El.Ing.HTL, M.Math.(CS) befasst sich seit mehr als 30 Jahren mit Software-Engineering, DSL, Parserbau für Legacy-Migration und -Integration.

Object Engineering GmbH
Birmensdorferstr. 32
CH-8142 Uitikon-Waldegg

Tel: +41 (0) 44 400 47 00

www.object-engineering.ch
kontakt@object-engineering.ch

SNG
Member of the
Solution Network Group

Object Engineering® ist ein eingetragenes Warenzeichen im Besitz der Object Engineering GmbH

Literatur und Links

[Nee19] M. Needham, A.Hodler, Graph Algorithms, O'Reilly, 2019
[ThKoKo13] R. Thurner, A. Koch, R. Koch, Modellbasiertes Software-Reengineering: Teilautomatisiert Migration eines GUI in eine Web-Umgebung in OBJEKTspektrum 06/14, https://www.objeng.ch/ThKoKo13
[Rob15] I. Robinson, J. Webber, E. Eifrem, Graph Databases, O'Reilly, 2015
[WSRE19-1] K.-U. Hermann, Teilautomatisiertes Architektur-Reengineering in einem JavaEE Monolithen, Bison Schweiz AG; Workshop Software-Reengineering & Evolution, Bad Honnef, Mai 2019, https://www.objeng.ch/WSRE19-1

Copyright © 2020 by Object Engineering GmbH CH-8142 Uitikon-Waldegg