**Technische Universität Braunschweig**
**Institut für Statik**
Prof. Dr.-Ing. Dieter Dinkler

**Masterarbeit**

# Parametrische, statische Dimensionierung von Tragwerken auf Basis von kommerzieller Software

Mathias Worm

25. Januar 2021

Prüfer: Prof. Dr.-Ing. Dieter Dinkler
Zweitprüferin: apl. Prof. Dr.-Ing. Ursula Kowalsky

Betreut durch:
Dr.-Ing. Christian Flack,
Dipl.-Ing.(FH) Lukas Sühnel,
Dipl.-Ing. (FH), M.Sc. Frank Sonntag

Technische Universität Braunschweig

**Institut für Statik**
**Prof. Dr.-Ing. Dieter Dinkler**

Masterarbeit für Herrn Mathias Worm, Matr.-Nr. 4936530

## Parametrische, statische Dimensionierung von Tragwerken auf Basis von kommerzieller Software

In der Entwurfsphase von Gebäuden lassen sich durch die Findung der optimalen Tragwerksform die Qualität und die Baukosten von Bauwerken maßgebend beeinflussen. Im Zuge der Einführung von Building-Information-Modelling (BIM) werden im zunehmenden Maße dreidimensionale Entwurfsmodelle erstellt. Daraus ergeben sich neue Möglichkeiten. Über parametrische und automatisierte Ableitungen von Analysemodellen können in kurzer Zeit unterschiedliche Varianten der Tragstruktur bezüglich Tragsicherheit und Gebrauchstauglichkeit aber auch nach architektonischen Gesichtspunkten beurteilt werden. Die Zielstellung dieser Masterarbeit ist, die grundlegende Vorgehensweise dieses parametrischen BIM-Workflows zu erarbeiten und anhand eines konkreten Beispiels umzusetzen.

Die Arbeit gliedert sich in folgende Teile:

1. Einführung in die parametrische Modellerstellung am Beispiel des Programmsystems Autodesk Revit und Dynamo. Das Programmsystem Autodesk Revit erlaubt über Dynamo eine von Parametern abhängige Modellierung von Gebäuden. Die grundsätzlichen Möglichkeiten dieser Software sind zusammenzufassen und deren Nutzen für die Erstellung von statischen Modellen zu analysieren
2. Erarbeitung und Darstellung eines Grundkonzeptes zur Integration Revit/Dynamo – RFEM/RSTAB Aufgrund eines konkreten Beispiels sind die in der Praxis in Frage kommenden typischen Workflows zu ermitteln und zu beschreiben. Dabei sind die zu berücksichtigenden Objekte (Stäbe, Flächen, Lasten, Lager usw.) zu ermitteln und zu prüfen, ob die Software alle Möglichkeiten bietet.
3. Programmierung eines einfachen Beispiels mittels Dynamo und RFEM/RF-COM. Aufbauend auf die vorher erarbeiteten Anforderungen ist ein Prototyp eines parametrischen Modells zu entwerfen und über eine Schnittstelle mittels Dynamo und RF-COM ein Datenaustausch zwischen Revit und RFEM zu realisieren.
4. Auswertung und Erfahrungsbericht. Anhand eines Fachwerkbinders sind die Möglichkeiten und Grenzen der parametrischen Modellerstellung zu dokumentieren.
5. Mögliche Erweiterungen und Synergie-Potenziale. Als Ergebnis sind mögliche Workflows zu beschreiben, die derzeit noch nicht möglich oder umsetzbar sind. Welche Funktionalitäten fehlen und sollten entwickelt werden?
6. Diskussion und Dokumentation der Ergebnisse

Während der Bearbeitung der Aufgabe ist enger Kontakt mit dem Institut zu halten. Die Betreuung erfolgt durch Herrn Dr.-Ing. Christian Flack, Lukas Sühnel (Dipl.-Ing. (FH)), Betreuer der BIM Schnittstellen bei Dlubal und Frank Sonntag (Dipl.-Ing. (FH), M.Sc.), Support Ingenieur, Coach vor Ort in Leipzig. Zweitprüferin ist Frau apl. Prof. Dr.-Ing. Ursula Kowalsky.
Die Besprechung der Vorgehensweise und des Arbeitsplanes erfolgt ca. zwei Wochen nach Ausgabe der Aufgabe.

| | |
|---|---|
| ausgegeben am: | verlängert bis: |
| abzugeben am: | abgegeben am: |

# Eidesstattliche Erklärung

**Masterarbeit**

**Parametrische, statische Dimensionierung von Tragwerken auf Basis von kommerzieller Software**

von:
Mathias Worm
Matr.Nr: 4936530

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit *Parametrische, statische Dimensionierung von Tragwerken auf Basis von kommerzieller Software* selbstständig verfasst sowie keine anderen als die vollständig angegebenen Quellen und Hilfsmittel benutzt habe und dass die Arbeit nicht bereits als Prüfungsarbeit vorgelegen hat. Mir ist bewusst, dass Täuschungsversuche - insbesondere nachgewiesene Plagiate - nach § 11, Abs. (4), Satz 2 der Allgemeinen Prüfungsordnung zum endgültigen Nichtbestehen einer Prüfung und somit dem Scheitern im Studiengang führen können.

Ort, Datum

Unterschrift

# Inhaltsverzeichnis

# Abkürzungsverzeichnis

**BIM** Building Information Modelling

**FEM** Finite Elemente Methode

**CAD** Computer Aided Design

**OOP** Objektorientierte Programmierung

**GUI** Graphical User Interface

**API** Application Programming Interface

**WPF** Windows Presentation Foundation

**UI** User Interface

**KI** Künstliche Intelligenz

# Abbildungs- und Tabellenverzeichnis

# Codeverzeichnis

# 1. Einleitung

Die Digitalisierung schreitet in jeglichen Bereichen der Wirtschaft immer weiter fort. Auch in der Baubranche ist mit der Idee des Building Information Modelling (BIM) der Grundstein bereits in den 90er Jahren gesetzt worden [8]. Die Idee dahinter ist in einem digitalen Modell die benötigten Informationen eines Bauwerks aller beteiligten Gewerke zu sammeln und somit eine gemeinsame Wissensgrundlage zu schaffen. Dabei spielt das Fachgebiet der Statik eine essenzielle Rolle. Alle Aspekte der Standsicherheit einer Struktur werden in einer Statik ermittelt und resultierend entsteht eine Basis für alle darauffolgenden Prozessschritte. Über die immer weiterentwickelten Computer Aided Design(CAD)-Werkzeuge zur Unterstützung der Modellierung eines BIM-Modells, ist die Möglichkeit zur parametrischen Modellierung entstanden. Grundlage dieser Modellierung ist die Erstellung eines Modells in Abhängigkeit von zuvor definierten Parametern, anhand derer das Modell verändert und konfiguriert werden kann. Der entstehende Mehrwert der gewonnenen Flexibilität in der Modellanpassung soll dazu genutzt werden, in der frühen Planungsphase eines Bauwerks das Optimum zu erzielen. Dabei kann das Optimum von verschiedensten Ausgangspunkten betrachtet werden. Im Rahmen dieser Masterarbeit wird aus der statischen Sicht ein neuer, BIM-orientierten Prozess geschaffen, der durch die technische Entwicklung einer Schnittstelle zwischen einer BIM- und Finite-Elemente-Software ermöglicht wird. Dabei repräsentiert das FEM-Programm, mittels dessen die Berechnung der Struktur durchgeführt wird, das Themengebiet der Statik und die BIM-Software liefert die Grundlage der parametrischen Modellierung.

Einen detaillierten Einstieg in die vorgestellte Thematik bieten die nachfolgenden Abschnitte 2 und 3. Es werden einführend die Vorteile von BIM und dessen Anwendung benannt und das Thema der Masterarbeit genauer eingeordnet. Daraufhin wird die parametrische Modellierung vorgestellt und darauf aufbauend die möglichen Workflows der Thematik betrachtet. Anschließend wird ein praxisnaher Prozess ausgewählt, der mittels der Schnittstelle umgesetzt wird. In den darauffolgenden Kapiteln 4 und 5 werden für die Entwicklung der Schnittstelle benötigte technische Grundlagen aufgeführt und die Entwicklung detailliert dokumentiert. Es werden besondere Meilensteine der Entwicklung genauer beleuchtet, um einen Ansatzpunkt für eine Umsetzung einer eigenen Entwicklung zu bieten. In den beiden letzten Kapiteln 6 und 7 werden die entstandenen Ergebnisse präsentiert und abschließend eine potentielle Weiterführung der Entwicklung diskutiert.

# 2. Building Information Modelling

Grundlage der Forschungsthematik dieser Arbeit ist das Konzept des Building Information Modelling oder auch kurz BIM genannt. Mit der Umstrukturierung der Baubranche in diese Richtung resultieren neue Möglichkeiten und Optionen. In diesem Kapitel werden diesbezüglich grundlegende Kenntnisse vermittelt und der Ansatzpunkt der vorliegenden Arbeit eingeordnet.

## 2.1. Begriffsdefinition

Das BIM-Konzept ist keine neue Erfindung der Bauwirtschaft. Schon in den 1970er Jahren wurden Forschungsarbeiten veröffentlicht, die den Einsatz von virtuellen Gebäudemodellen behandeln. Eine Begrifflichkeit wie BIM wurde zu dieser Zeit jedoch noch nicht verwendet. Diese Beschreibung wurde erstmals in einem Paper der Wissenschaftler van Needervan und Telman im Jahre 1992 erwähnt und begann sich mittels einer Veröffentlichung des Unternehmens Autodesk im Jahre 2003 zu verbreiten. [8] Dennoch gibt es bis heute keine einheitliche Definition für den BIM-Begriff. Abhängig vom Kontext und den darin enthaltenen Akteuren können unterschiedliche Sachverhalte mittels BIM beschrieben werden. Somit kann mit dem Kurzwort BIM ein Building Information Model gemeint sein, was die Sicht auf die reinen Gebäudedaten fokussiert, wohingegen Building Information Modelling sich eher auf die Planungsphase bezieht. Ebenfalls kann von Building Information Management die Rede sein, sofern der Managementaspekt hervorgehoben werden soll. [20] In dieser Arbeit liegt der Fokus auf der Phase der Planung und somit ist mit der Abkürzung BIM immer das Building Information Modelling gemeint. Dies definiert eine objektorientierte Repräsentation eines Bauwerks oder bebauter Umgebung. [20] Dabei wird bei der Modellierung eine große Informationstiefe forciert, die sich in nicht-geometrischen Zusatzinformationen widerspiegelt. Verwendete Objekte besitzen beispielsweise Typinformationen, technische Eigenschaften oder auch Kosten. BIM beschreibt dabei nicht nur den Erschaffungsprozess eines digitalen Bauwerkmodells, sondern auch deren Änderung und Verwaltung über den ganzen Lebenszyklus. [8]

## 2.2. Warum Building Information Modelling?

Die Digitalisierung der Wirtschaftszweige wird unter dem Stichwort „Industrie 4.0“ schon seit mehreren Jahren vorangetrieben. Über kurze Zeit konnte in großen Bereichen der weltweiten Industrie ein erheblicher Anstieg der Produktivität verzeichnet werden. Auch in der Planung, Ausführung und Nutzung von Gebäuden ist im Bauwesen eine vermehrte Nutzung von digitalen Werkzeugen zu erkennen. Dennoch besitzt die Bauindustrie einen sehr großen Nachholbedarf in Bezug auf die Digitalisierung im Vergleich zur standortgebundenen Industrie [8][20].
Somit fehlt an gewissen Stellen Informationstechnik, die Prozessschritte erleichtert beziehungsweise erst ermöglicht. Dies resultiert in einem erheblichen Verlust an wertvollen

Informationen. Ausgelost wird dies durch die Übermittlung der notwendigen Informationen anhand von beschränkenden Dateiformaten oder gedruckten Bauplänen. Dabei begrenzt sich der Verlust nicht auf einzelne Phasen, sondern beeinflusst alle fünf: Entwurf, Planung, Ausführung, Bewirtschaftung und Umbau [8]. In der nachfolgenden Abbildung 2.1 ist der Informationsverlust schematisch dargestellt. Über die Dauer eines Projekts steigt das Wissen der Gesamtbeteiligten über alle Phasen an. Die Schwierigkeit dabei ist den Mitwirkenden alle nötigen Informationen in gewünschter Menge, Qualität und Geschwindigkeit bereitzustellen. Wie in Abbildung 2.1 deutlich zu erkennen, findet ein großer Verlust an Informationen an den Phasenübergängen statt. Diese sind meist gleichbedeutend mit einem Wechsel der Verantwortlichkeiten und dem bearbeitenden Team.

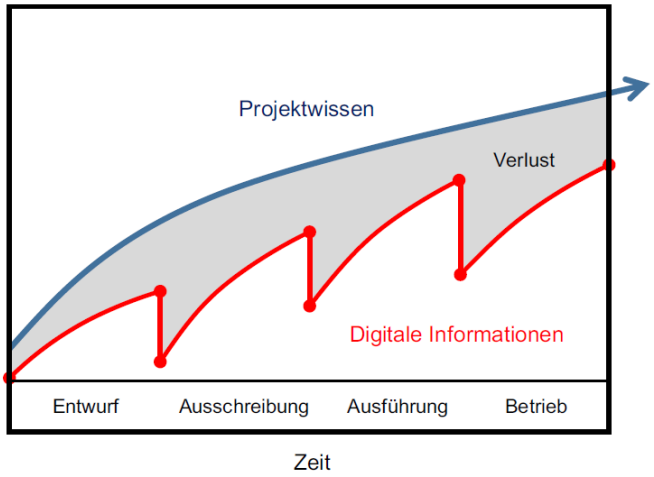


Abbildung 2.1.: Informationsverlust zwischen den einzelnen Phasen bei der Übergabe [8]

Die Realisierung eines Bauvorhabens ist ein komplexer Prozess mit einer Vielzahl an Akteuren aus verschiedenen Fachdisziplinen. Ein reibungsloser Ablauf kann nur gewährleistet werden, sofern ein intensiver Informationsaustausch stattfindet. Generell geschieht dies in Form von technischen Zeichnungen und Gebäudeinformationen als Schnitte, Grundrisse und Detailzeichnungen. Diese enthalten meist keine weiteren Hintergrundinformationen, da sie nur Striche auf einem Papier sind. Jegliche Zusatzinformationen, welche dieser Zeichnung zugrunde liegen, gehen verloren. Das Potential der Informationstechnologie bleibt somit unangetastet und die Prozesse können sich nicht weiterentwickeln. Ein großes Problem stellt die Überprüfbarkeit dieser Pläne dar. Sie können meist nur manuell gesichtet und überprüft werden, was eine große Fehlerquelle darstellt. Weiterhin werden viele einzelne Zeichnungen von verschiedensten Fachplanern dezentral abgelegt weshalb ein Informationsverlust vorprogrammiert ist. [8]
Wie wichtig die Kooperation in der Planungsphase ist, ist zu erkennen, sofern die daraus resultierenden Bauschäden und Fehler an Bauwerken betrachtet werden. So ergibt sich nach [1], dass allein 90 % aller Baumängel durch die Planung und Ausführung verursacht werden. Von der Planung allein werden 37 % verursacht. Mit BIM können diese Prozesse effizienter gestaltet werden. Dies beweist beispielsweise die Automobilindustrie. Sie nutzt schon seit einer längeren Zeit eine durchgängige modellgestützte Produktentwicklung und -fertigung und konnte resultierend prägnante Effizienzsteigerungen verzeichnen. [8]
Dies ist eine der Beispielbranchen, welche eine schnelle Digitalisierung umsetzen konnte. Die Baubranche im Ganzen kann einen solch schnellen Wandel nicht vollziehen, da sie

mit anderen Randbedingungen zu kämpfen hat. Zum Beispiel ist die ganze Wertschöpfungskette generell nicht durch einen Unternehmer abgedeckt, sondern wird mittels einer Vielzahl zusammengestellt. Dabei ist die Zusammenarbeit in der Regel auf wenige Projekte begrenzt und der Informationsaustausch findet über vielfältige Kanäle statt, was wiederum zu Verlusten führt. [8] Ein einheitlicher, standardisierter Datenaustausch ist noch nicht realisiert, obwohl es mit dem IFC-Dateiformat Bestrebungen in diese Richtung gibt. Für weiterführende Informationen ist auf Quelle [8] verwiesen. Jedoch ist der Wandel in der Bauindustrie von vielen einzelnen, kleinen Unternehmen abhängig und nicht von wenigen großen, wie es beispielsweise bei der Automobilindustrie ist.
Dabei ist der Fortschritt der Digitalisierung in der Bauwirtschaft von großer Wichtigkeit, da sie am Anfang der industriellen Wertschöpfung steht. Ohne Sie würden sämtliche Bauwerke und deren Infrastruktur fehlen. Aus diesem Grund ist besonders der Fortschritt in diesem Bereich für eine gesamtwirtschaftliche Entwicklung notwendig [20].

## 2.3. Was ist Building Information Modelling?

Die Idee von Building Information Modelling versucht genau den zuvor angesprochenen Informationsverlust zu beherrschen und mittels eines konsistenten 3D-Modells zu überbrücken. In erster Linie soll mit diesem Modell eine Grundlage der Kommunikation entstehen und ein hoher hinterlegter Informationsgrad erreicht werden. Der Begriff Buildung Information Modelling beschreibt somit den Prozess der Erstellung, Änderung und Pflege dieser Modelldaten. Das Ziel ist es, über die einzelnen Phasen des Lebenszyklus eines Bauwerks hinweg, von der Planung bis zum Rückbau (siehe Abbildung 2.2), Modelldaten zu nutzen und somit den großen Mehrwert aller spezifischen Gebäudeinformationen zu besitzen. [8]


Abbildung 2.2.: Lebenszyklus eines Bauwerks im Sinne von BIM und die enthaltenen Informationen der Phasen [8]

In Abbildung 2.2 ist der angesprochene Zyklus eines Building Information Model, also

dem eigentlichen Modell, abgebildet. Weiterhin sind die Bereiche aufgeführt, welche im Modell eine Berücksichtigung finden und somit als Information enthalten ist.
Zur Erstellung eines solchen Modells sind entsprechende digitale Werkzeuge notwendig. Dabei handelt es sich nicht um eine reine 3D-CAD Software, sondern vielmehr um eine 3D-Modellierungsumgebung, in der jegliches Element einem Katalog an Bauteilen entnommen wird. Es wird eine Wand, Stütze, Fenster, Tür auch als diese erkannt. So können zum Beispiel im Sinne der Statik relevante Informationen, wie die Systemlinie bzw. -fläche, hinterlegt werden, um ein statisches Modell zu erstellen [1]. Weiterhin können benötigte Pläne oder Schnitte vom Modell abgeleitet werden. Dies führt dazu, dass alle Pläne die gleiche Basis haben und somit konsistent sind. Außerdem können Projektschritte und deren Abläufe genauer geplant und mit in das Model integriert werden. Durch die Sammlung aller benötigten Informationen an einem Ort, müssen keine repetitiven Arbeiten durchgeführt werden, was zur Steigerung der Arbeitsleistung führt und die Fehleranfälligkeit im Allgemeinen senkt. [8]

## 2.4. BIM in der Anwendung

Mit der Anwendung von Building Information Modelling ergibt sich eine erhebliche Änderung in der Planungsphase. Diese ist grafisch in Abbildung 2.3 dargestellt. Mittels dieser Grafik kann der Vorteil vom BIM bestens beschrieben werden.

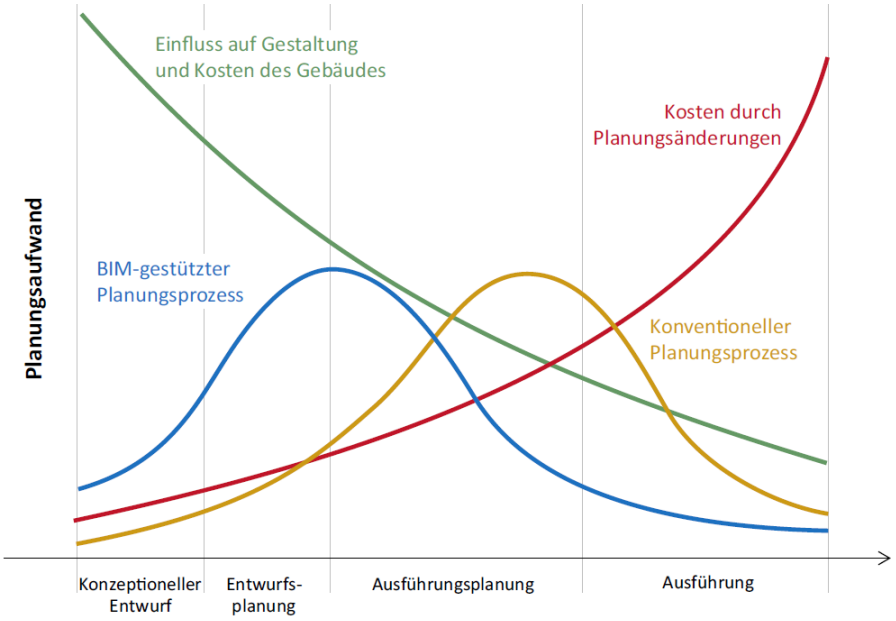


Abbildung 2.3.: Phasenverschiebung der Planung mit BIM in Gegensatz zur konventionellen Planung [8]

Die Abszisse zeigt die einzelnen Phasen vom Entwurf über die Planung bis hin zur Ausführung. Die Ordinate beschreibt das Maß an Planungsaufwand. Mittels des in Blau dargestellten Graphen ist der deutlich phasenverschobene, BIM-gestützte Planungsprozess gegenüber der konventionellen Planung (in Gelb) zu erkennen. Letztere zeichnet sich dadurch aus, dass erst in späteren Phasen der eigentliche Hauptaufwand geleistet wird. Somit ist es auch erst zu einem späteren Zeitpunkt im Projekt möglich eine umfassende

Bewertung des Entwurfs durchzuführen [8]. Dies kann zu Änderungen führen, die sich dann stärker auf das Budget ausschlagen können. Das Änderungspotential und die damit verbundenen Kosten werden über die Graphen in Grün und Rot illustriert.
Der BIM-gestützte Prozess benötigt dahingegen den größten Anteil an Planungsaufwand in den ersten Phasen und bleibt somit sehr viel flexibler in der Umplanung und Bewertung von Entwürfen, da es ein komplexes 3D-Modell gibt. Somit werden die Kosten minimiert und die Flexibilität und Qualität der Arbeit maximiert.
Die Vorteile in der Planungsphase sind somit deutlich zu erkennen. Doch nicht nur die Planung profitiert von der Anwendung von BIM. Auch in der Bauausführung und in der späteren Nutzung entstehen Vorteile. Diese werden im Rahmen der Arbeit jedoch nicht weiter beleuchtet. Für weiterführendes Material ist auf die Quellen [8] und [20] verwiesen.

## 2.5. Statisch Parametrische Modellierung im BIM Prozess

Der größte Vorteil, der sich durch eine BIM-gestützte Planung ergibt, ist die gewonnene Flexibilität der Entwurfsänderung in der frühen Projektphase. Da mit hoher Sicherheit davon ausgegangen werden kann, dass sich Änderungen ergeben werden, sollte der Planungsprozess dies berücksichtigen und bestmöglich darauf vorbereitet sein. Mittels einer parametrischen Modellierung kann dies erreicht werden. Das Modell kann in gewünschter Art und Weise von zuvor ermittelten Parametern abhängig gestaltet werden, die das größte Änderungspotenzial darstellen. Änderungen am Bauwerk resultieren meist in geometrischen Anpassungen. Mitwirkende Gewerke müssen sich den Änderungen annehmen und gegebenenfalls neu planen. Aus tragwerksplanerischer Sicht ist anzustreben, dass sich das Tragwerk den neuen Randbedingungen automatisch anpasst und neu berechnet wird, sofern sich beispielsweise im Laufe des Projekts die Anzahl oder Größe der Fenster ändert. Um dies umzusetzen, ist ein hoher Grad an Automatisierung und die Entwicklung von ausgereiften Algorithmen erforderlich. Des Weiteren sind intelligente Softwaresysteme notwendig, die als Basis dieser Entwicklung dienen können.

Mittels dieser Arbeit wird ein erster Schritt in diese Richtung erarbeitet und der momentane Stand der Technik diesbezüglich analysiert. Es wird versucht eine mögliche Entwicklungsrichtung anhand von Beispielen zu betrachten und anschließend zu bewerten. Im nachfolgenden Kapitel werden erste theoretische Untersuchungen durchgeführt.

# 3. Statisch parametrische Dimensionierung von Tragwerken

Für die Lösung von Ingenieuraufgaben werden meist Modelle erstellt oder vorhandene herangezogen. In Bezug auf das Bauwesen werden für ein Planungsprojekt viele verschiedene Modelle verwendet, die unterschiedlichste Ziele verfolgen können. So existieren in erster Linie geometrischen Modelle, welche zur Darstellung und Berechnung dienen. Weitere Betrachtungen widmen sich z.B. den baubetrieblichen sowie zeitlichen Aspekten. Die Art und Weise der Repräsentation kann von einem vollständigen dreidimensionalen Modell einer Baustelle bis hin zur Abbildung der Ablaufplanung in einem Zeitstrahl dargestellt werden. Gemein haben all diese Betrachtungen, dass sie die zukünftige Realität in einem kleineren Maßstab wiedergeben und als Ausführungshilfe dienen. Unter Berücksichtigung des BIM-Konzepts sollten alle Modelle zu einem fusionieren und somit nur noch eine Quelle an Information darstellen. Logischer Weise sind viele Prozesse voneinander abhängig und bauen sequenziell aufeinander auf. Diese Abhängigkeit führt dazu, dass nachgelagerte Prozessschritte sich an die vorgegebenen Randbedingungen anpassen und bei Änderung dieser, erneut ausgearbeitet und überdacht werden müssen, sofern diese schon geplant wurden. Das entstehende Änderungspotential der vorgelagerten Schritte sollte bei der Planung als eine Art Konstante angesehen werden, um auf Anpassungen vorbereitet zu sein. Mittels des Konzepts der parametrischen Modellierung kann dieses Ziel erreicht werden. Dabei ist jedoch zu beachten, dass die Parameter abhängig von der jeweiligen Betrachtung sind. Generell kann eine Fragestellung oder These dabei helfen Parameter zu definieren. Um dies einmal zu verdeutlichen wird beispielhaft ein Bauprojekt aus einer wirtschaftlichen Sicht begutachtet.

These: „*Wirtschaftlich ist ein Projekt, sofern es seine Baukosten erwirtschaftete und darüber hinaus noch weitere Einnahmen generiert.*“

Mit dieser Leitthese könnte geschlussfolgert werden, dass eine Möglichkeit zur Erlangung von Wirtschaftlichkeit die Senkung der Baukosten ist. Generell können diese über die erstelle Gebäudefläche oder das Volumen des Bauwerks bestimmt werden, sofern wir von einem Wohn- oder Bürogebäude ausgehen. Alternativ könnte auch eine Betrachtung über das Gewicht eines Bauwerks stattfinden. Allein durch den Unterschied des Blickwinkels auf die Ermittlung der Wirtschaftlichkeit eines Gebäudes können verschiedenste Variablen oder Parameter berücksichtigt werden. Somit können je nach Aufsteller des Modells und dessen Motivation, beziehungsweise Aufgabenstellung, die Parameter variieren und von unterschiedlichster Natur sein.

## 3.1. Statisch parametrische Modellierung

Dreht sich der Blickwinkel eines Modells auf die statische Sicht, so sind meist geometrische Eigenschaften von vorrangigem Interesse. Die Höhe, Länge und Breite eines Objekts sind die offensichtlichsten Parameter. Vorgegeben werden diese Dimensionen meist durch den Entwurf eines Architekten, mit dem dann der Tragwerksplaner die Statik des Bauwerks aufstellt. Um ein generelles Verständnis für die Herangehensweise der parametrischen Modellierung zu bekommen, wird nachfolgend als Grundlage der Statik ein parametrisch architektonisches Modell erstellt und das Potenzial dieser Modellierung aufgezeigt.

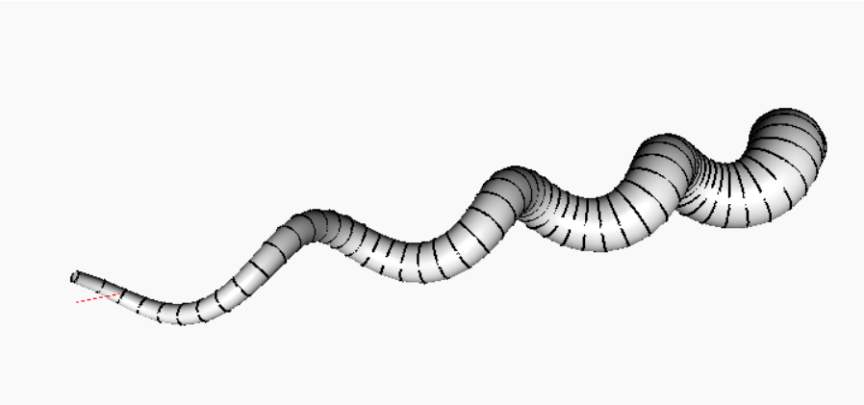

Abbildung 3.1.: Beispielstruktur eines architektonischen parametrischen Modells

Das in Abbildung 3.1 dargestellte Modell stellt die Geometrie eines futuristischen Gebäudes dar. Dabei ist der kreative Aspekt in den Vordergrund gestellt. Im ersten Schritt der Modellierung muss ermittelt werden, wie die gewünschten Formen und Strukturen beschrieben werden können. Der dargestellten Struktur liegen z.B. mathematische Funktionen zu Grunde, welche als Ausgangssituation für eine parametrische Gestaltung dienen sollen. Die entsprechende Logik ist die Folgende:

```
1
2 x = 0..360*rotation..5;
3 y = Math.Sin(x + phase) * amplitude;
4 z = Math.Cos(x + phase) * amplitude;
5
```

Code 3.1: Definition der Punktkoordinaten der Mantelfläche

Mittels der definierten Koordinaten $X, Y$ und $Z$ werden die Punkte der Mantelfläche der Geometrie erstellt. Die drei enthaltenen Variablen *rotation*, *phase* und *amplitude* sind die ausgewählten Parameter, da sie das Erscheinungsbild in gewünschter Art ändern können. Mittels des Parameters *rotation* kann die Ausdehnung der Struktur in $X$-Richtung variiert werden (siehe Zeile 2 Code 3.1). Die Winkelfunktionen Sinus und Cosinus dienen zur Ermittlung der Koordinaten für $Y$ und $Z$. In Zeile 3 und 4 in Code 3.1 ist die Nutzung der anderen Parameter abgebildet. Dem Namen entsprechend können die Winkelfunktionen einer Phasenverschiebung über *phase* unterzogen werden. Durch *amplitude* kann der Ausschlag, also die Amplitude, der Funktionen verändert werden, was im mathematischen Sinne den Grad der Steigung beschreibt. Ein abschließender Parameter $d$ wurde zur Änderung des Durchmessers der einzelnen Kreise hinzugefügt, um mehr Diversität

zu erzielen. Die Kreise sind in Abbildung 3.1 schwarz dargestellt.
Die Motivation dieses Modells ist die Erstellung einer korkenzieherähnlichen Struktur, die aus architektonischen Gesichtspunkten bewertet wird. Das Modell ermöglicht es über das Ändern der Parameter weitere komplexe Geometrien zu erstellt, ohne die vorgelagerte Arbeit zu verlieren, wie es in der üblichen Planung der Fall wäre. Ein weiterer, planerischer Vorteil, der durch das Parametrisieren von Modellen entstehen kann, ist die Möglichkeit, den Bauherren in die architektonische Gestaltung mit einzubeziehen. In kürzester Zeit können unterschiedlichste Geometrien erstellt werden. Das Potential dieser Modellierung ist in der nachfolgenden Abbildung 3.2 visualisiert.

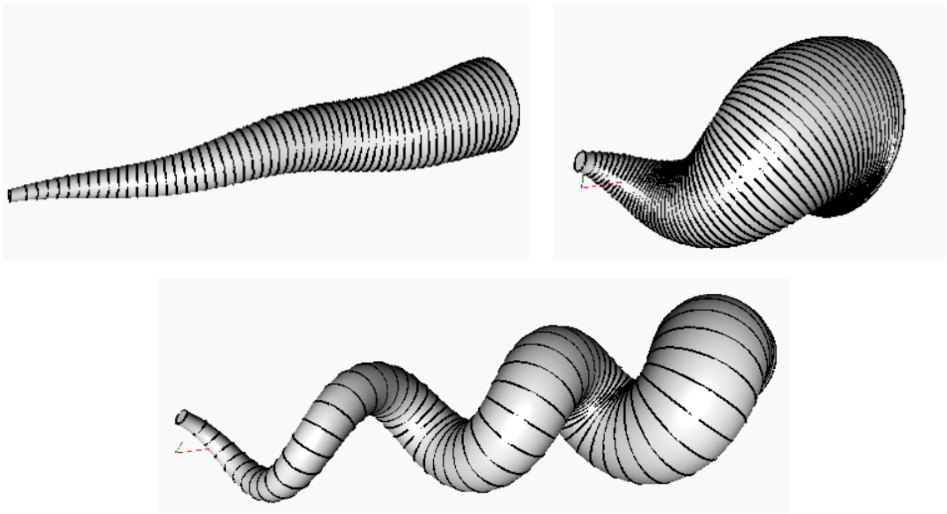

Abbildung 3.2.: Diversität des parametrischen Beispielmodells

Wie in der Architektur ist es auch in der Statik möglich, ein parametrisches Modell zu erstellen. Dort ergibt sich der Unterschied zum zuvor gezeigten Modell, dass der Anspruch und der Zweck ein anderer sind. Dies resultiert darin, dass die Ansätze der Parametrisierung überdacht werden müssen.
Die Statik als Disziplin hat sich zur Aufgabe gesetzt, die Tragfähigkeit von diversen Strukturen zu ermitteln. Dafür betrachtet man das Tragsystem und nicht das Erscheinungsbild. Das Tragwerk ist unter der gestalterischen Hülle verborgen und zeigt sich dem Betrachter nicht augenblicklich. Der Ingenieur hat die Aufgabe das statische Modell zu designen und kann verschiedenste Systeme aufbauen. Um die verschiedenen Ansätze der Modellierung besser zu verstehen, findet nachfolgend ein Modellvergleich statt.
In der Abbildung 3.3 ist der visuelle Unterschied eines geometrischen und statischen Modells gezeigt. Auf der linken Seite ist das geometrische Modell eines vierstöckigen Rohbaus abgebildet, welches alle Elemente in seiner tatsächlichen Ausführung zeigt. Hierbei handelt es sich um einen klassischen Massivbau eines Bürogebäudes. Die enthaltenen Stützen werden auf Einzelfundamenten gegründet, wohingegen die tragenden Wände mittels Streifenfundamenten in den Untergrund eingebunden werden. Des Weiteren enthalten die Bodenplatten vom 1. bis zum 3. OG eine kreisrunde Öffnung. Diese werden über ein Stützensystem in Kombination mit aussteifenden Wänden getragen. Auf dem Dach ist eine Terrasse vorgesehen, die mittels einer Stahlkonstruktion überdacht werden soll. Ein zusätzlich aussteifendes Treppenhaus ist auf der rechten Seite an das Bauwerk angeschlossen. Diese Beispielkonstruktion dient nur dem Vorführungszweck und enthält daher nur durch die Statik berücksichtigte Elemente. Aus solch einem geometrischen

Entwurf wird dann das Tragsystem extrahiert, welches auf der rechten Seite der Abbildung 3.3 dargestellt ist. In Türkis sind alle Mittellinien der Stützen sowie Mittelflächen der Scheibentragwerke dargestellt. Die Plattentragwerke sind hier in Ocker abgebildet wohingegen alle Lager rot sind. Dieses Modell definiert alle tragsystemrelevanten Bauteile. Dabei werden diese Elemente auf ihre Systemachse beziehungsweise -fläche begrenzt, da die Dimensionierung dieser das Berechnungsziel ist. Für ein vollständiges Berechnungsmodell müssten zusätzlich die entsprechenden Lasten aufgebraucht werden.

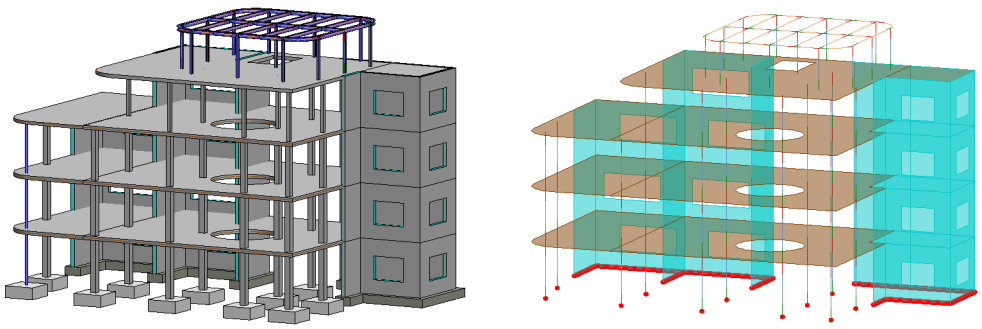

Abbildung 3.3.: Unterschied zwischen geometrischem und statischem Modell

Es ist nicht immer von Vorteil ein statisches Modell im Dreidimensionalen zu entwerfen, da die Berechnungsmodelle um ein Vielfaches komplexer werden und resultierend der zeitliche Aufwand steigt. Der Vorteil dahingegen ist die Genauigkeit der Berechnung und somit auch die bessere Dimensionierung der Bauteile, da dreidimensionale Effekte berücksichtigt werden können. Dieser Sachverhalt und viele weitere dieser Art müssen vom Ingenieur bewertet und analysiert werden. So bestimmt der Tragwerksplaner das Design der Tragstruktur und kann Einfluss auf die Berechnung nehmen. Je nach Planer kann somit ein leicht anderes System entstehen, das lediglich in den Detailpunkten Unterschiede aufweist. Die grobe Geometrie wird nicht geändert, da die äußere Gestalt meist durch einen Entwurf vorgegeben ist und dieser nur in letzter Instanz angepasst werden sollte. Der Bauingenieur in der Rolle als Tragwerksplaner ist nur dann an der Gestaltung eines Bauwerks maßgebend beteiligt, sofern es sich um Ingenieurbauten handelt, da diese oft Statik-getrieben sind. Ingenieurbauten sind beispielsweise Stützbauwerke, Tunnel oder Brücken. Am Beispiel einer Brücke kann der Begriff „Statik-getrieben“ gut verdeutlicht werden. Das Tragwerk einer Brücke ist meist gleichzusetzen mit dem Erscheinungsbild. Also bestimmt das Tragwerk und die Umsetzbarkeit dessen die Gestalt.
Es entstehen zwei unterschiedliche Aufgabenbereiche, die die Statik im Bauwesen einnehmen kann. Zum einen ist die reine Dimensionierung einer vorgegebenen Struktur gefragt, wohingegen auf der anderen Seite zusätzlich die Gestaltung gewünscht ist. Verknüpft man nun diese Erkenntnis mit der parametrischen Modellierung, wird deutlich, dass die statisch parametrische Modellierung in die jeweils speziellen Workflows eingegliedert werden muss und deren Anwendbarkeit sichergestellt wird. Ziel sollte es sein, einen einzigen Prozess zu kreieren, der alle Bedürfnisse abdeckt.

## 3.2. Praxisrelevante Workflows zur statisch parametrischen Modellierung

Wie im vorherigen Abschnitt erläutert, ist ein wichtiger Aspekt bei der Betrachtung von statisch parametrischer Modellierung die Einbindung und Analyse der praxisrelevanten Workflows. Es muss sichergestellt werden, dass sich die neue Modellierung dem Arbeitsablauf anpasst und damit eine Sinnhaftigkeit der Anwendung gegeben ist. So soll sich als Ziel dieses Abschnitts ein Anwendungsgebiet beziehungsweise ein Anwendungsfall herauskristallisieren, der als Grundlage des konzeptionellen Entwurfs der Schnittstelle dient. Ohne diese vorherigen Überlegungen könnte die zu entwickelnde Schnittstelle einen Prozess unterstützen, der in sich schlüssig und sinnvoll ist, jedoch keine Anwendung in der Praxis findet. Um dies zu vermeiden, ist in der ersten Entwicklungsphase über das Ziel und den Mehrwert der Anwendung nachzudenken und diese anhand der praxisorientierten Prozesse der Tragwerksplanung zu analysieren. Ein weiterer wichtiger Punkt, der bei der Umsetzung nicht aus den Augen verloren werden sollte, ist die Effizienzsteigerung. Nur sofern diese gegeben ist, sollte der Anwendung und ihrer Weiterentwicklung nichts im Wege stehen. Das für den Prozess beschränkende Maß stellt die technische Umsetzbarkeit dar. Durch vorhandene Softwaresysteme und Entwicklungsmöglichkeiten sind die Randbedingungen vorgegeben. Somit muss sich eine Symbiose aus den drei Faktoren Umsetzbarkeit, Anwendbarkeit und Effizienz ergeben.
Als Ausgangssituation dient der schon in Abschnitt 3.1 erwähnte Prozess, dass für ein durch den Planer vorgegebenes Modell die Statik aufgestellt wird. Es kann angenommen werden, dass dieser Workflow in der Praxis generell am häufigsten auftritt. Dieser

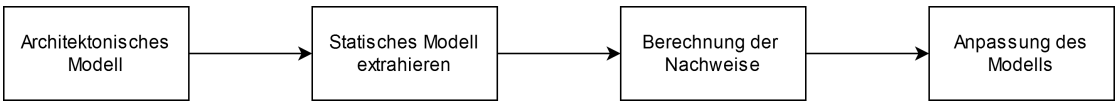


Abbildung 3.4.: Genereller Prozess der Tragwerksplanung

Prozess ist in der Abbildung 3.4 dargestellt. Auf Basis eines architektonischen Entwurfs beziehungsweise Modells beginnt die Tragwerksplanung das statische Modell zu generieren. Anhand dessen werden dann die zu erbringenden Nachweise geführt und die Struktur entsprechend angepasst oder beibehalten, sofern die Nachweise es erfordern. Dieser Prozess beschreibt den generellen Arbeitsablauf aus Sicht der Tragwerksplanung. Um die bestmögliche Anwendbarkeit einer parametrischen Modellierung in der Statik zu gewährleisten, ist es wichtig, sich an diesem Prozess zu orientieren.
In strikter Anlehnung an die Ausgangssituation ist der Prozess in Abbildung 3.5 entstanden. Es dient ein architektonisches Modell als Grundlage der Modellierung des statisch parametrischen Modells. Der zweite Schritt wird dahingehend geändert, dass nun ein parametrisches Modell abgeleitet wird und dieses leicht änderbar ist. Die nachgelagerten Prozessschritte bleiben äquivalent.

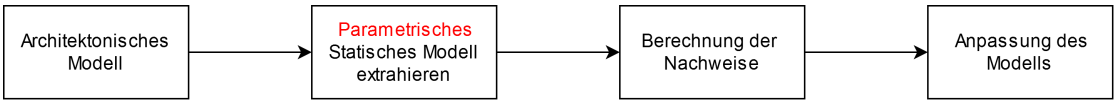


Abbildung 3.5.: Dimensionierender praxisnaher Workflow zur statisch parametrischen Modellierung

Der interessanteste Punkt im Workflow, der eingehend genauer betrachtet wird, ist die Erstellung des parametrischen Modells. Dabei ist vorstellbar, dass dieser Schritt manuell oder automatisiert stattfinden könnte. Optimaler Weise würde das statische Modell automatisch aus dem architektonischen Modell abgeleitet werden und lediglich Detailanpassungen müssten stattfinden. Dies würde eine erhebliche Effizienzsteigerung nach sich ziehen. Das automatische Ableiten einer Struktur und Berücksichtigung aller Eventualitäten erfordert jedoch einen erheblichen Arbeitsaufwand im Design und der Entwicklung. Außerdem ist kritisch zu analysieren, in welchem Teilschritt die Parameter definiert werden sollen und wie sie in das Modell eingebunden werden. Das Prinzip der parametrischen Modellierung ist die Erstellung eines Modells, welches der Abhängigkeit vorher definierter Parameter unterliegt. In diesem Kontext ist unklar, wie zuvor definierte Parameter dem Automatismus übergeben werden könnten. Es ist zweifelhaft, dass dieser Schritt in den Prozess mit eingebunden werden kann und in welcher Form dieser umgesetzt werden könnte. Um dies zu klären, muss die Fragestellung beantwortet werden, ob das automatische Ableiten eines parametrischen Modells generell möglich ist und welche technischen Voraussetzungen benötigt werden. Dennoch ist davon auszugehen, dass die technische Umsetzbarkeit einer Teilautomatisierung des Prozessschrittes gegeben sein sollte. Dies sollte jedoch mit zusätzlichen Analysen genauer überprüft werden. Resultierend übersteigt diese Entwicklung den zeitlichen Rahmen der vorliegenden Arbeit und könnte daher als Thematik weiterführender Untersuchungen dienen. Daher wird vorausgesetzt, dass der Teilschritt „Ableitung eines statisch parametrischen Modells“ manuell ausgeführt wird. Dieser Schritt wird generell ohnehin vom Tragwerksplaner manuell ausgeführt, was somit keinen Mehraufwand bedeutet. Daher ist der zuvor in Abschnitt 3.1 genannte Aufgabenbereich der Statik, *die reine Dimensionierung*, mit einem Prozess versehen und der gestalterische Aspekt wird anschließend mit in den Prozess einbezogen. *Die Gestaltung* mittels eines parametrischen Modells setzt voraus, dass die Erstellung dessen im Fokus und an die erste Stelle des Workflows rutscht und zusätzlich auch eine essenzielle Rolle einnimmt. Der Workflow ändert sich wie folgt:

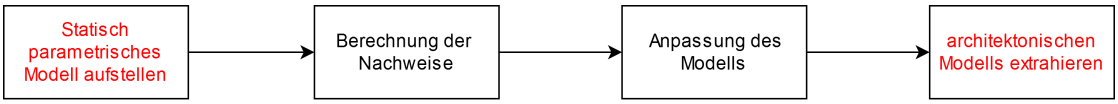


Abbildung 3.6.: Gestalterischer praxisnaher Workflow zur statisch parametrischen Modellierung

In diesem Workflow ist das parametrische Modell in den Fokus gerückt und wird initial aufgestellt. An dieser Stelle fließen die zuvor erarbeiteten Parameter in die Modelldaten ein. Somit stehen diese im Mittelpunkt und dienen als Ausgangspunkt für eine mögliche Weiterverarbeitung. Anschließend werden die statischen Berechnungen durchgeführt und deren Ergebnisse untersucht. Daraufhin werden die Anpassungen aufgrund der Nachweislage durchgeführt. Der Schritt zur Erstellung eines architektonischen Modells findet abschließend statt. Dies gewährleistet, dass die Informationen der Berechnungen gleich mit einfließen. Der Vorteil dieser Prozessabfolge ist, dass das architektonische Modell als Resultat des Workflows entsteht, gefordert Gebrauchstauglichkeits- beziehungsweise Tragfähigkeitsnachweise schon erfüllt sind und keine weiteren Berechnungen durchgeführt werden müssen. Dieser Prozess bezieht nun auch das gestalterische Arbeitsfeld der Tragwerksplanung ein. Eine technische Umsetzbarkeit dieses Workflows sollte ebenfalls gegeben sein.

Die Abbildung 3.5 und Abbildung 3.6 beschreiben die angestrebten Prozesse, anhand derer die Umsetzung der Schnittstelle ausgerichtet wird. Somit wäre ein erster wichtiger Schritt für die Nutzung einer parametrischen Modellierung im Bereich der Statik getätigt.
Auf dieser Basis sind weiterführende Entwicklungen denkbar. Mit dem Ziel das perfekte Bauwerk aus verschiedensten Blickwinkeln erzeugen zu können, wäre eine Optimierung die Lösung. Es könnte mittels der Parameter ein Modell erstellt werden, das je nach Anwender mit den aus seiner Sicht wichtigsten Parametern versehen und über diese optimiert wird. Diese Funktionalität würde einen großen Vorteil gegenüber der konventionellen Bemessung von Tragwerken darstellen. Durch eine computerseitige Optimierung wird es dem Anwender ermöglicht in kürzester Zeit komplexe Strukturen bezüglich ihrer Gestalt und dessen Tragwerk zu untersuchen. Resultierend könnten über Fallstudien schon im frühen Stadium eines Projekts das optimale Bauwerk designet werden, was sich bestens an BIM ausrichten lässt. Der entsprechende Workflow gestaltet sich diesbezüglich wie folgt:

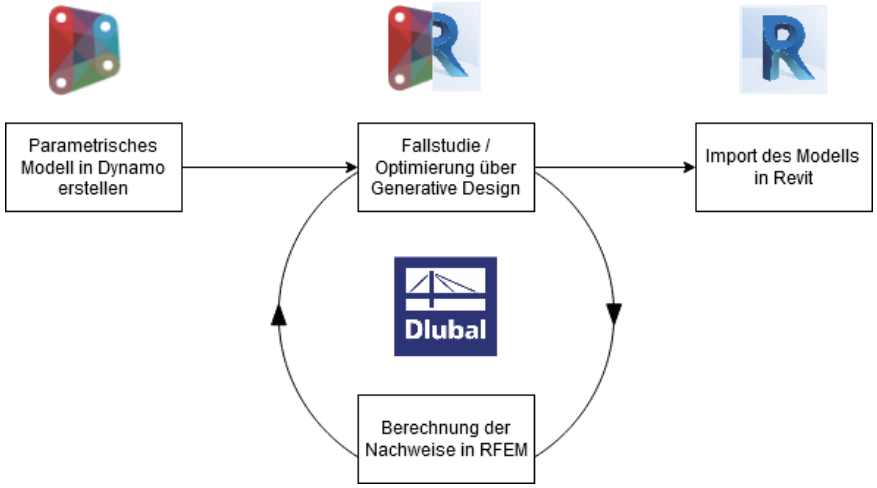


Abbildung 3.7.: resultierender Workflow mit Zuordnung der verwendeten Softwarekomponenten (Logos:[5][7][9])

In der Abbildung 3.7 ist der entstehende Workflow dargestellt und mit den vorgesehenen Softwarekomponenten versehen. Das parametrische Modell wird initial in der Software Dynamo aufgebaut. Diese bietet das Framework zur parametrischen Modellierung. Daraufhin wird die Optimierung über das in Dynamo enthaltene Modul namens Generative Design durchgeführt. Während der Optimierung werden die Berechnungen in RFEM durchgeführt und die Ergebnisse in Generative Design zusammengetragen und analysiert. Nachdem die Optimierung abgeschlossen ist, kann das optimierte architektonische Modell in Revit eingefügt werden.
Der in Abbildung 3.7 resultierende Workflow dient als Grundlage der nachfolgenden Entwicklung. Gelingt es diesen als eine Prototyp-Entwicklung umzusetzen, ist die zuvor angestrebte Symbiose aus den drei Faktoren Umsetzbarkeit, Anwendbarkeit und Effizienz möglich und das Potenzial einer Weiterentwicklung ist bewiesen.

## 3.3. Verwendete Software

In diesem Abschnitt werden alle Programme vorgestellt, die für die Umsetzung dieser Arbeit benötigt wurden.

### 3.3.1. Autodesk Revit

Die Software Revit 2021 ist eine Planungssoftware aus dem Hause Autodesk. Dabei konzentriert sich der integrierte Planungsprozess am Konzept von Building Information Modelling. Die dazugehörige Benutzeroberfläche ist in der Abbildung 3.8 dargestellt. Das Modellieren eines Bauwerks in einer 3D CAD Umgebung ermöglicht die Erstellung eines detaillierten Modells. Dieses kann dann so erweitert werden, dass für alle im BIM-Prozess beteiligten Gewerke entsprechende Informationen enthalten sind. Aus der statischen Sicht kann ein analytisches Modell als Hintergrundinformation hinterlegt werden, siehe Abbildung 3.3. Revit enthält eine Vielzahl von verschiedensten Funktio-

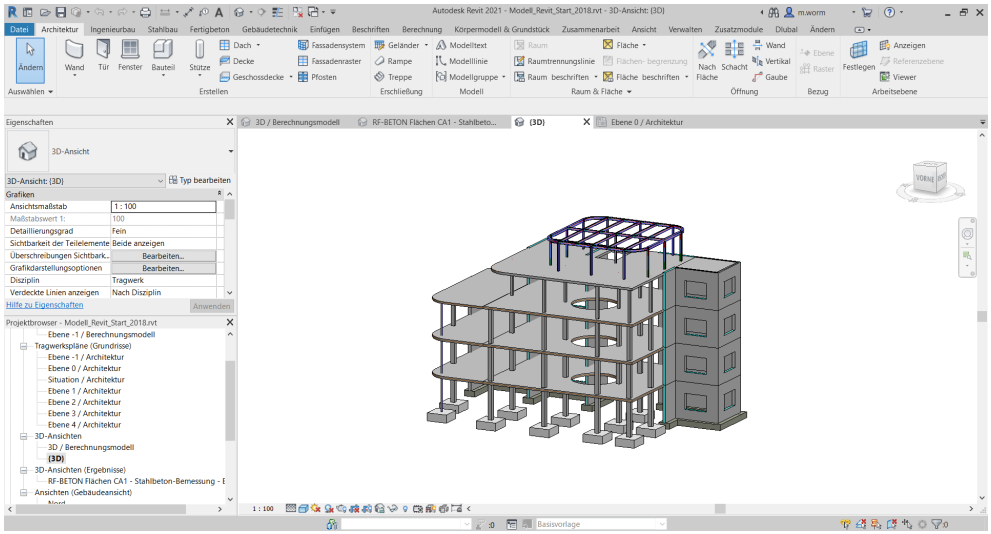

Abbildung 3.8.: Die Revit Benutzeroberfläche

nen. So könnten in einem *Architektur*-Menü grundlegende Bestandteile wie beispielsweise Wände, Dächer, Geschosse oder Fassadensysteme erstellt werden. Über die spezifischeren Reiter Ingenieurbau, Stahlbau, Fertigbeton, Gebäudetechnik können detailliertere Konstruktionen in das Bauwerk integriert werden. All diese Module bieten eine solide Basis an Elementen zur Konstruktion von Bauwerken. Über die sogenannten *Familien* erhält der Anwender die Möglichkeit, diese Elemente individuell zu erweitern und diverse Objekte selbst zu erstellen. Diese Funktion ermöglicht dem Anwender alle Freiheiten der Modellierung und der Bedarf an Modifikation kann gedeckt werden. Mittels einer Integration von Dynamo in Revit ist es möglich die Modellerstellung in Revit über eine Logik zu realisieren. Dynamo bildet dabei die Grundlage einer parametrisierten Modellierung wohingegen Revit als ein Drehpunkt für den ganzheitlichen BIM-Prozess angesehen werden kann.

### 3.3.2. Dynamo

Die Software Dynamo ist eine Plattform zur visuellen Programmierung und in verschiedensten Versionen erhältlich. Für die vorliegende Arbeit wurde Dynamo for Revit 2.5 verwendet. Diese Version ist seit Revit 2020 mit in Revit integriert und benötigt keine zusätzliche Installation. Unter dem Namen Dynamo Sandbox ist eine weitere open-source Variante erhältlich, die alle Kernfunktionen enthält und die jeweils neuesten Features integriert. Mittels des open-source Konzepts ist der Quellcode der Applikation zur Weiterentwicklung durch die Dynamo-Community offengelegt [5]. Diese Version ist interessant, sofern eine nicht-kommerzielle Alternative gefragt ist oder ein autarkes Framework zur visuellen Programmierung benötigt wird. Da die gewünschte Schnittstelle an den BIM-Prozess angeschlossen sein wird, ist eine Anbindung an Revit als Basis notwendig. Im weiteren Kontext wird die Version Dynamo for Revit der Einfachheit halber nur noch als Dynamo bezeichnet. Wie oben genannt verwendet Dynamo als Grundlage

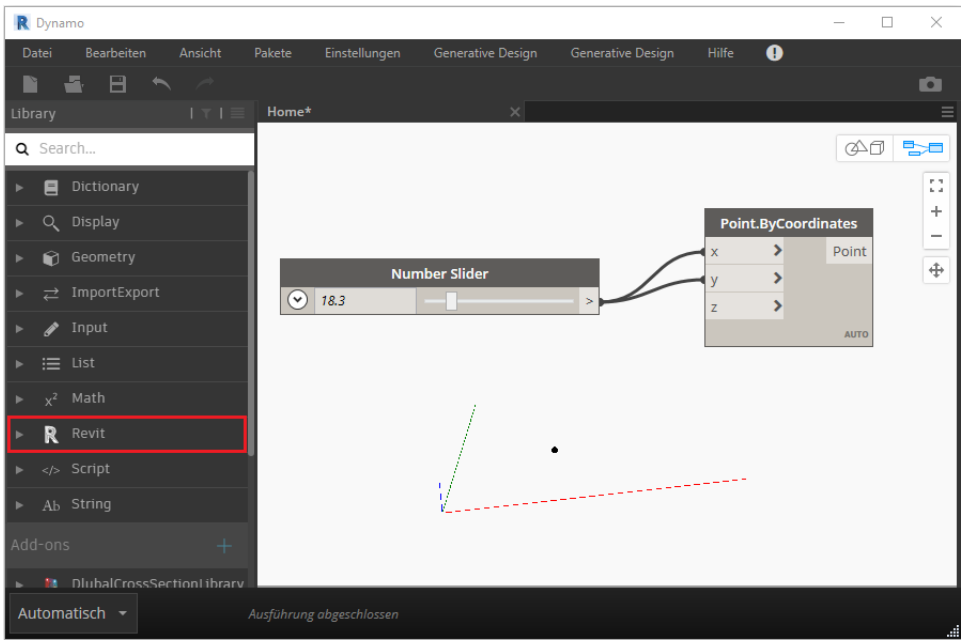


Abbildung 3.9.: Vorstellung Dynamo als Plattform zur visuellen Programmierung - *graph*: Anlegen eines Punktes in Dynamo

der parametrischen Modellierung verwendet das Konzept der visuellen Programmierung. Dies bedeutet, dass über sogenannte Knoten ein Netzwerk aus Anweisungen erstellt werden kann. Diese wird dem Dynamo-Standard entsprechend nachfolgend nur noch als *graph* (eng.) bezeichnet. Die visuelle Programmierung zielt darauf ab, auch „nicht-Programmierer“ in die Erstellung von Algorithmen und Logik mit einzubeziehen. Es setzt keine Vorkenntnisse der klassischen textbasierten Programmierung voraus und ist somit schneller durch eine breite Masse an Nutzern erlernbar [6]. Als Produkt entsteht eine sequenzielle Struktur von Anweisungen, die entweder automatisch beziehungsweise manuell ausgeführt werden kann. In der Abbildung 3.9 ist ein einfaches Beispiel eines *graphs* abgebildet. In ihm sind zwei Konten enthalten, die einen Punkt im lokalen Koordinatensystem anlegen, der über einen Slider parametrisch verändert wird. Durch den abgebildeten „Number Slider“- Knoten kann ein gewünschtes Zahlenintervall beschrieben werden, das an bestimmte Inputs (hier die Koordinaten X und Y) übergeben werden

kann und somit der Output gewünscht angepasst wird. Mittels dieser Slider wird eine parametrische Modellierung ermöglicht und kann durch den Anwender hinsichtlich individueller Ziele verwendet werden.
Zur Realisierung der Schnittstelle mit Revit bietet Dynamo ein vorgefertigtes Paket (siehe Abbildung 3.9 linke Sidebar), welches Knoten bereitstellt, die die Interaktion mit Revit erlaubt. Zusätzlich bietet Dynamo die Möglichkeit eigene Pakete einzubinden, welche anschließend an selber Stelle unter dem Punkte *Add-ons* zu finden sind. Dieser Mechanik wird sich im Rahmen dieser Arbeit bedient.

### 3.3.3. Autodesk Generativ Design

Mit der Version Revit 2021 hält die neue Funktionalität namens Generativ Design Einzug in die Software. In einer früheren standalone-Version des Moduls wurde es unter dem Namen *Refinery* geführt und befand sich noch im Beta-Stadium [5]. Es befähigt den Anwender über Dynamo und Revit einen *graph* einer sogenannten „Studie“ zu unterziehen. Um dies besser zu verstehen, muss das Designkonzept dahinter genauer beleuchtet werden. Allgemein betrachtet lässt sich das Generative Entwerfen den Computergestützten Designs unterordnen. Darunter versteht man einen Ansatz, bei dem ein Konstrukteur eine Reihe von Anweisungen, Regeln und Beziehungen definiert, die genau die Schritte identifizieren, die notwendig werden, um das vorgestellten Design zu erreichen [6]. Somit wird der Computer dazu verwendet, um rechen- und zeitintensive Schritte zu übernehmen und somit Aufwand zu sparen. Generative Design geht noch einen Schritt weiter, indem es Parameter und eine Zielstellung verwendet. Diese dienen dem Computer dazu, alle möglichen Designs zu erforschen und Designvariationen zu generieren. Durch die zu definierenden Parameter kann der Computer konkurrierende Zielstellungen abzuwägen und hinsichtlich eines Optimums entscheiden [6]. Dieser zielorientierte Designansatz ermöglicht es dem Konstrukteur, einen besseren Einblick in das Design selbst zu enthalten, da mehrere Alternativen miteinander verglichen werden und somit eine Aussagekraft bezüglich deren Wertigkeit entsteht. Zusätzlich können durch die beschleunigte Betrachtung schneller fundierte Designentscheidungen getroffen werden [6]. Im Allgemeinen kann gesagt werden, dass schneller qualitativ hochwertigere Ergebnisse erzielt werden können. Für weiterführende Betrachtungen dieser Thematik und den dahinterliegenden Mechanismen, sei auf die Quelle [6] verwiesen. Diese Betrachtung wird mittels der zuvor angesprochenen Studie auf einen *graph* angewendet. Es stehen mehrere Zielstellungen zur Auswahl.

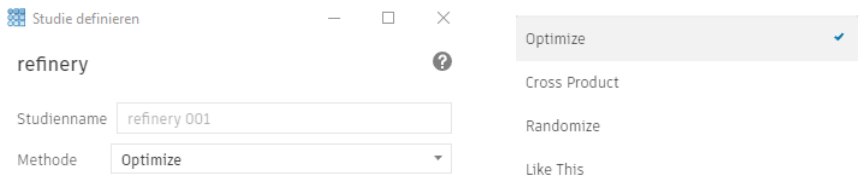


Abbildung 3.10.: Benutzeroberfläche des generativen Designs und deren Konfigurationsmöglichkeiten

In der Abbildung 3.10 (rechts) sind diese aufgeführt. Bezüglich der statischen Betrachtungen soll eine Optimierung des erstellten Tragwerks durchgeführt werden. Dabei bietet sich die Möglichkeit die zuvor als Output deklarierten Ergebnisse hinsichtlich eines Maximums oder Minimums zu optimieren. Dabei kann bestimmt werden über wie viele

Generationen (Iterationen) die Optimierung stattfinden soll. Es dient jeweils die vorherige Generation als Input der Nächsten, womit sich eine Art Evolution einstellt [6]. Mittels dieser Methode wird versucht eine Fallstudie/Optimierung entsprechend dem in Abbildung 3.7 dargestellten Workflows durchzuführen.

### 3.3.4. RFEM

Zur Berechnung des Tragwerks wird ein FEM-Programm verwendet. Dabei steht die Abkürzung FEM für die Finite Elemente Methode, die das Tragwerk eines Bauwerks in ein Netz aus kleinen endlichen Elementen zerlegt. Jedem Element liegen physikalische Modelle zugrunde, die das Verhalten unter einer Belastung abhängig vom betrachteten Werkstoff beschreiben. Mit zusätzlich festgelegten Randbedingungen kann dann die Berechnung der Verformung dieses Elementnetzes bestimmt werden und anschließend auf die internen Kraftgrößen geschlossen werden. Abschließend kann dann ermittelt werden, wie das Tragwerk dimensioniert werden muss, um der gegebenen Belastung Widerstand zu leisten. Die für diese Untersuchung kommerziell verwendete Software des Unternehmens Dlubal GmbH nennt sich RFEM 5.23. Die Software stellt dabei die reine FE-Kernapplikation dar, die mittels einer modular aufgebauten Programmfamilie erweitert werden kann. Über ein wahlweise 2D beziehungsweise 3D CAD System können Strukturen definiert und deren Material und Einwirkung modelliert werden. Der Anwender ist in der Lage ebene und räumliche Platten-, Scheiben-, Schalen- und Stabtragwerke zu kreieren. Die Realisierung von Mischsystemen, Volumen- und Kontaktelemente ist ebenfalls gegeben. Über die große Anzahl an Zusatzmodulen können weiter Struktur-Analysen entsprechender Normung integriert werden. [9]
Eines der für diese Untersuchungen wichtigsten Module ist RF-COM. Dabei handelt es sich um eine programmierbare Schnittstelle (API), die auf der von Microsoft Corporation basierenden COM-Technologie basiert. Über RF-COM wird dem Nutzer ein Befehlssatz an die Hand gegeben, der die Steuerung der Applikation ohne Nutzeroberfläche freigibt. Es können Geometrien erstellt und die für eine Berechnung wichtigen Informationen und Elemente angelegt werden. Für den in Abbildung 3.7 dargestellten Workflow dient RFEM als Rechenkern.

### 3.3.5. Visual Studio Community 2019

Visual Studio Community 2019 ist eine kostenlose integrierte Entwicklungsumgebung (IDE) von Microsoft zur Erstellung von Anwendungen für Android, iOS und Windows [17]. Visual Studio bietet eine generelle Programmierumgebung, in die nachträglich die für das gewünschte Projekt benötigten Pakete integriert werden können. Zusätzlich bietet sich der große Vorteil, dass in Visual Studio ein Debugger integriert ist, der es erlaubt in die laufende Anwendung einzugreifen.
Die IDE ist somit der Ort an dem die Schnittstelle entwickelt wird und alle Verknüpfungen geschaffen werden. Für eine detailliertere Darstellung der Software und deren Anwendung ist auf den Unterabschnitt 4.3.1 verwiesen.

# 4. Schnittstelle Dlubal RFEM - Autodesk Revit/Dynamo

## 4.1. Technische Grundlagen

In diesem Abschnitt werden die für diese Arbeit wichtigen technischen Bereiche genauer ausgeführt, da so ein gewisses Grundverständnis der komplexen Sachverhalte vermittelt wird. Im Fokus der Betrachtung stehen die wichtigsten Themen und Konzepte.

### 4.1.1. Objektorientierte Programmierung

Für die Entwicklung der Schnittstelle ist eingehend das Konzept der objektorientierten Programmierung, abgekürzt OOP, zu verstehen, da es das Fundament der ganzen Entwicklung ist. Wie der Name des Programmierungskonzepts schon vermuten lässt, dreht sich alles um Objekte. Die Begrifflichkeit objektorientierte Programmierung wurde durch *Alan Kay* definiert und erstmalig in seiner Publikation [12] im Kontext der Programmiersprache *Smalltalk* erwähnt. Bis Mitte der Neunzigerjahre wurde die OOP zur Standardmethode der Softwareentwicklung [13]. Um einen guten Einstieg in diese Thematik zu erhalten, hat sich bewiesen, dass die enthaltenen Prinzipien anhand eines greifbaren Beispiels leichter vermittelt werden können. Somit wird nachfolgend ein realistisches Szenario betrachtet, mittels dessen eine Anwendung konzipiert wird. Dafür versetzt man sich nun in die Lage eines Fahrradladeneigentümers, der an Kunden Fahrräder vermietet. Ein Managementsystem wird benötigt, das den Vermietungsprozess digitalisiert und gleichzeitig dokumentiert.
Die essenzielle Aufgabenstellung einer jeden Softwareentwicklung ist das Managen der vorhandenen Komplexität [10]. Dabei ist die Begrifflichkeit „Komplexität“ anfangs schwer zu definieren. Beschreiben kann man sie als eine abstrakte Definition, die versucht den bevorstehenden Entwicklungsaufwand abzuschätzen. Die objektorientierte Programmierung beschreibt die reale Welt mittels Klassen und Objekten, um die vorhandene Komplexität in kleine verständliche Sinneinheiten einzuteilen, die für den Menschen greifbar sind. Die essenziellen Bausteine der objektorientierten Programmierung, die verstanden werden müssen, sind *Klassen*, *Objekte*, *Attribute* und *Methoden*. Dafür werden diese Bestandteile auf das zuvor beschriebene Praxisbeispiel des Fahrradladens bezogen.

Initial sollte sich die Frage gestellt werden, welche Akteure am umzusetzenden Prozess beteiligt und mit in das Programm aufgenommen werden sollten. Für die Vermietung eines Fahrrads wird das Rad selbst, ein Kunde und die Vermietung benötigt. Diese realen Objekte können jeweils mittels einer Klasse beschrieben werden. Eine Klasse kann als eine Standarddefinition beziehungsweise ein Bauplan für ein Objekt angesehen werden. In diesem Bauplan sind Attribute enthalten, die als Eigenschaften angesehen werden können. Außerdem werden Methoden definiert, die das Verhalten dieser Klasse beschreiben. Dabei werden nur zielrelevante Attribute und Methoden einer Klasse designet, die dem Zwecke der Softwarefunktionalität dienen. Die Klasse eines Kunden könnte beispielsweise wie folgt definiert werden:

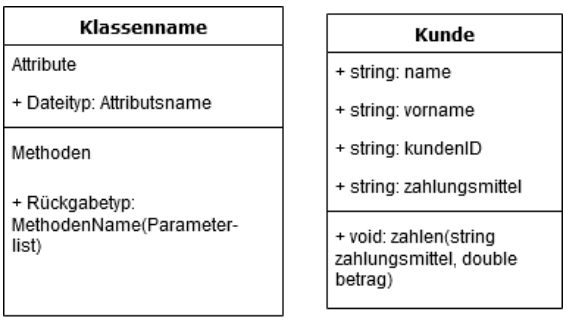


Abbildung 4.1.: Allgemeiner Aufbau eines Klassendiagramms (links) und der spezifische Aufbau der Beispielklasse Kunde (rechts)

In der oberen Abbildung ist der allgemeine Aufbau einer Klasse als Unified Modelling Language Diagramm (UML) sowie das darauf aufbauende Klassendiagramm *Kunde* aufgeführt. Die Attribute *name*, *vorname*, *kundenID* und *zahlungsmittel* sind die benötigten Eigenschaften, die für das Vermietungssystem benötigt werden. Kommt nun ein neuer Kunde in den Laden, wird dieser im Programm angelegt. Jeder Kunde wird als ein eindeutiges *Objekt* mit seinen spezifischen *Attributen* ins System eingepflegt. Dieses *Objekt* wiederum wird von der *Klasse Kunde* abgeleitet. Allgemein kann ein *Objekt* als eine Instanz einer *Klasse* beschrieben werden [12]. Über die Methode zahlen() ist das *Objekt* in der Lage einen Geldtransfer für das Fahrrad durchzuführen.
Somit wird für jeden Kunden ein Objekt dieser Klasse erzeugt und die kundenspezifischen Attribute hinterlegt. Objekte sind in der Lage miteinander zu kommunizieren und interagieren, was das objektorientierte Programmdesign grundlegend ermöglicht [12][13]. Im gleichen Stil müssen dann Klassen für die Vermietung und das zu vermietende Fahrrad erstellt werden. Daraufhin wird die Kommunikation mittels der in den Klassen enthaltenen Methoden realisiert. Somit ist ein Design in den ersten Zügen für diese Anwendung fertiggestellt.
Jedoch sollte nicht mit der Implementation begonnen werden, ohne die drei Grundprinzipien der objektorientierten Entwicklung zu berücksichtigen, da erst diese ein vollständiges Design ermöglichen. Definiert wurden sie erstmals durch *Alan Kay* und sehen wie folgt aus: [12]

- Datenkapselung
- Vererbung
- Polymorphie

Das Prinzip der *Datenkapselung* befasst sich mit objektspezifischen Daten und deren Zugehörigkeit sowie Zugriffsrechte. Durch die Einführung von Objekten werden Daten in diesen gespeichert und sollten auch nur durch sie selbst gelesen beziehungsweise geändert werden können. Hintergrund ist das geregelte Ändern beziehungsweise Übergeben von objektspezifischen Daten, um zu gewährleisten, dass diese nicht in einem invaliden Zustand geraten. Darunter wird verstanden, dass es zum Beispiel nicht möglich sein darf, einem Fahrradrahmen eine negative Höhe zuzuweisen. Der Zugriff kann über Methoden der Klasse geregelt werden. [13]
Das Prinzip der *Vererbung* kann als eine hierarchische Gliederung von Klassen angesehen werden. Bei der vererbenden Klasse spricht man von einer Elternklasse, wohingegen die erbende Klasse als Kindklasse bezeichnet wird. Vererbung wird verwendet um Redundanzen im Quellcode zu vermeiden und eine Art Spezialisierung einzuführen, die die Kindklasse annimmt. Dabei werden die Eigenschaften und Methoden von der Elternklasse an die Kindklasse übergeben beziehungsweise vererbt. Sollte die Funktionalität einer Methode für die erbende Klasse angepasst werden, kann dies innerhalb der Kindklasse durchgeführt werden. [13]
Das Prinzip der *Polymorphie* beschreibt eine Vielgestaltigkeit von Objekten einer Art. So kann am Beispiel von Fahrrädern Polymorphie anschaulich erklärt werden. Spezifische Formen wie das Rennrad, E-Bike oder Trekking-Rad können gefunden werden. Dennoch besitzen alle Fahrräder die gleichen grundlegenden Eigenschaften. Sie haben alle zwei Reifen, einen Lenker und einen Sattel. Außerdem sind sie alle auf den Menschen zugeschnitten und können durch ihn gefahren werden. Man erhält eine Vielgestaltigkeit, die mit Hilfe einer Elternklasse namens Fahrrad hierarchisch unterteilt werden kann. [13]

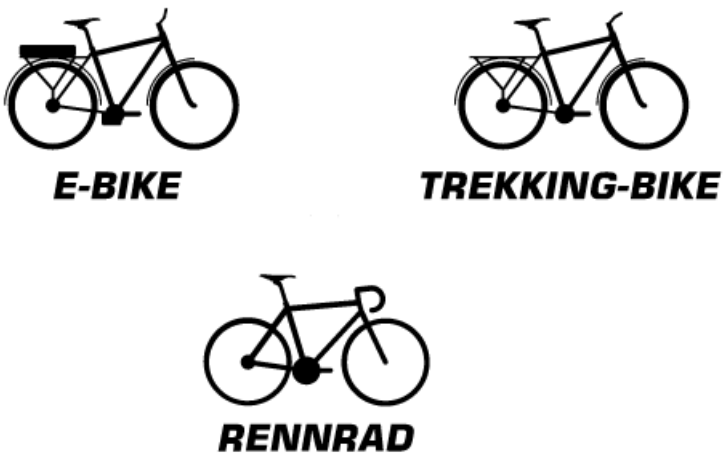


Abbildung 4.2.: Polymorphie Beispiel des Fahrrads [14]

Eine letzte Begrifflichkeit, die im Sinne der objektorientierten Programmierung betrachtet werden muss, ist das Interface. Aus dem Englischen übersetzt bedeutet dies „Schnittstelle“, was auch treffend die Aufgabe dieser bezeichnet. Das Interface stellt eine verallgemeinerte Kommunikationsschnittstelle zwischen programmtechnischen Einheiten dar. So kann zum Beispiel eine Schnittstelle die Kommunikation zwischen Klassen beziehungsweise Modulen oder sogar ganzen Programmen managen. Genau dieser Technologie wird sich bedient, sofern ein Programm eine API verwendet.

### 4.1.2. Programmiersprache C#

C# ist eine Programmiersprache, die als eine Entwicklung von Anders Hejlsberg, Scott Wiltamuth und Perter Golde gilt. Im Jahre 2000 wurde sie durch Microsoft als Teil des .Net Frameworks erstmals vorgestellt. Entwickelt wurde sie unter den Aspekten, dass sie einfach, modern und objektorientiert sein sollte, um für allgemeine Zwecke verwendet werden zu können. Dabei sollen implementierte Funktionen wie das *strong type checking*, *array bounds checking*, *uninitialized variable checking* und *automativ garbage collection* die Softwareentwicklung unterstützen, um einen einheitlichen Standard zu etablieren [11]. All diese Funktionen machen C# zu einer zeitgemäßen und gut anwendbaren Programmiersprache, die vom .Net Framework und dessen zahlreichen Funktionen stark profitiert.

### 4.1.3. .Net Framework

Das .Net Framework ist eine Plattform zur Entwicklung von Software. Vor seiner Veröffentlichung von Microsoft in Verbindung mit C# wurden viele Applikationen auf Basis der COM-Technologie aufgebaut [2]. Wie in Unterabschnitt 3.3.4 erwähnt, baut auch die Schnittstelle RF-COM von RFEM darauf auf. Vorteilhaft an der COM-Technologie ist die Fähigkeit, verschiedenste Programmiersprachen vereinen zu können. Zum Beispiel kann eine in *C++* geschriebene Bibliothek auch unter *Visual Basic* verwendet werden [2]. Als eine Art Weiterentwicklung dieser Technologie wurde daraufhin das .Net Framework für die Windows-Plattform eingeführt. Diese soll ein flexibleres und stärkeres Fundament bieten als COM [2]. Die neueste Entwicklung von Microsoft ist die .Net Core Plattform, welche die Bindung an das Windows-Betriebssystem aufhebt und ebenfalls Entwicklungen auf z.B. macOS, iOS, Android und verschiedensten Unix/Linux Distributionen ermöglicht [2]. Das für diese Arbeit verwendete .Net Framework ist sehr stark mit der Programmiersprache C# verknüpft und in Kombination bieten diese eine Anbindung an die veraltete COM-Technologie. Dies sind dennoch nicht alle unterstützten Entwicklungsmöglichkeiten. Nachfolgend werden weitere Funktionen von .Net aufgeführt [2]:

- Verschiedenste Programmiersprachen werden unterstützt (C#, Visual Basic, F#, etc.).
- Eine geteilte runtime-Umgebung wird von allen .Net-fähigen Programmiersprachen verwendet.
- Unterstützt Vererbung, Errorhandling, Debugging auch über die Grenzen einer Programmiersprache hinaus.
- Umfassende Bibliothek von tausenden vordefinierten Klassen für unterschiedlichste Anwendungen vorhanden.

- Bibliotheken werden nicht mehr an das System gebunden (Registrierungseintrag) und können mehrfach nebeneinander existieren.

### 4.1.4. Dynamic Link Library

Eine Dynamic Link Library, auch *dll* abgekürzt, ist in erster Linie ein Dateiformat, das unter den Windows-Betriebssystemen verwendet wird. Dabei handelt es sich um eine kompilierte Bibliothek, in der Funktionen hinterlegt sind, die zuvor unter Verwendung verschiedener Programmiersprachen erstellt wurden. Die Besonderheit dieses Datenformats ist, dass mehr als ein Programm gleichzeitig auf dieses zugreifen kann und die enthaltenen Funktionen nutzt. Ein gutes Beispiel dafür ist die Comdlg32.dll, die alle allgemeinen Dialogfeldfunktionen enthält. Jedes Programm, das diese verwendet, ist in der Lage ein Dialogfeld zu öffnen. Durch das *.dll*-Format wird die Wiederverwendung von Code gefördert und eine effiziente Speichernutzung hervorgerufen, aufgrund der geringen Größe der Datei. Unter Windows ist dies die übliche Praxis um Funktionalität zu kapseln. Ein Programm kann somit verschiedenste .dll-Dateien implementieren und benötigte Funktionen aus diesen übernehmen. Somit entsteht ein Katalog aus modularem Code, der aufgrund seiner geringen Größe schnell geladen wird und die Performance von Programmen steigert. Es ist bei der Verwendung sicherzustellen, dass die durch die implementierte *dll* eingebrachte Abhängigkeit nicht von anderen Programmen überschrieben oder geblockt wird, da dies häufig Fehler aufwirft [17].
Dieses Dateiformat wird für die Entwicklung benötigt und beherbergt den Quellcode der Schnittstelle.

## 4.2. Architektur der Schnittstelle

In den vorherigen Kapiteln wurden alle Voraussetzungen und Grundlagen vermittelt, um sich nun dem Design und der eigentlichen Funktionalität zu widmen. Die in Abschnitt 3.2 beschriebenen Workflows sind dabei immer im Blick zu halten. Als Einstiegspunkt muss geklärt werden, was umgesetzt werden soll und wie genau dies geschieht. In erster Linie ist zu zeigen, dass es möglich ist, den angestrebten Workflow umzusetzen und einen Prototyp zu erstellen. Detailliert beschrieben sollte die Schnittstelle in der Lage sein, die in Dynamo erstellten tragwerksspezifischen Daten zu übertragen. Dabei ist noch zu nennen, dass der Prototyp nur für die Übertragung von Stabtragwerken ausgelegt wird. Diese bilden den optimalen Ausgangspunkt, da sie aus geometrischer Sichtweise nur an Punkten miteinander verbunden sind und somit weniger Randbedingungen auftreten als bei Scheibentragwerken. Sie können als die einfachsten statischen Elemente angesehen werden.
Als Ziel soll ein Fachwerkträger als parametrisches Modell in Dynamo erstellt werden, anschließend an RFEM übergeben und dort berechnet werden. Diese Aufgabe kann in mehrere Schritte unterteilt werden. Der erste ist die parametrische Modellierung. Dieser Schritt wird über Dynamo realisiert und benötigt keiner weiteren Entwicklung, da Dynamo alle Funktionalitäten bereitstellt. Im zweiten Schritt muss die Geometrie aus Dynamo zu RFEM übertragen werden und in einem dritten, findet die Berechnung statt. Der zweite Schritt ist der Punkt an dem die Schnittstelle zum Einsatz kommt und die Geometrie übersetzt wird. Dafür wird das Anlegen folgender Elemente benötigt:

| **geometrische Elemente** | **statische Elemente** |
|---|---|
| • Knoten bzw. Punkte | • Stäbe |
| • Linien | • Lager |
| | • Lasten |
| | • Lastfälle |
| | • Lastfallkombinationen |
| | • Material |
| | • Querschnitt |

Es kann dabei nach zwei verschiedenen Elementtypen unterschieden werden. In Dynamo werden aus Modellsicht nur rein geometrische Informationen erzeugt (Punkte und Linien). Diese reichen nicht aus, um ein vollständiges, statisches Modell zu erstellen. So muss beispielsweise aus einer Linie ein Stab generiert und an diversen Knoten Lager angebracht werde. Zusätzlich ist es nötig Querschnitte zu erstellen und diesen ein Material zuzuweisen. Diese und weitere Aufgaben müssen durch die Schnittstelle ermöglicht werden. Dafür ist es notwendig herauszufinden, wie diese Funktionalität implementiert werden kann.
Für die Umsetzung einer Anbindung der beiden Programme sind softwareseitig Voraussetzungen notwendig, welche sowohl von Dynamo 2.5 als auch von RFEM 5.6 erfüllt werden müssen. In erster Linie ist eine programmierbare Schnittstelle (eng. API) notwendig, da diese erst einen Datenaustausch ermöglicht. Eine API kann grundsätzlich als eine Möglichkeit angesehen werden softwareinterne Logik zu verwenden, die standardisiert über die API angesprochen werden kann, ohne den ursprünglichen Quellcode offen zu legen. Zusätzlich ist der Umfang an freigegebener Funktionalität essenziell. Meist ist es nicht möglich alle programminternen Funktionen mittels der API zu steuern. Sowohl Dynamo als auch RFEM bieten eine API, die für die Entwicklung der Schnittstelle verwendet wird.

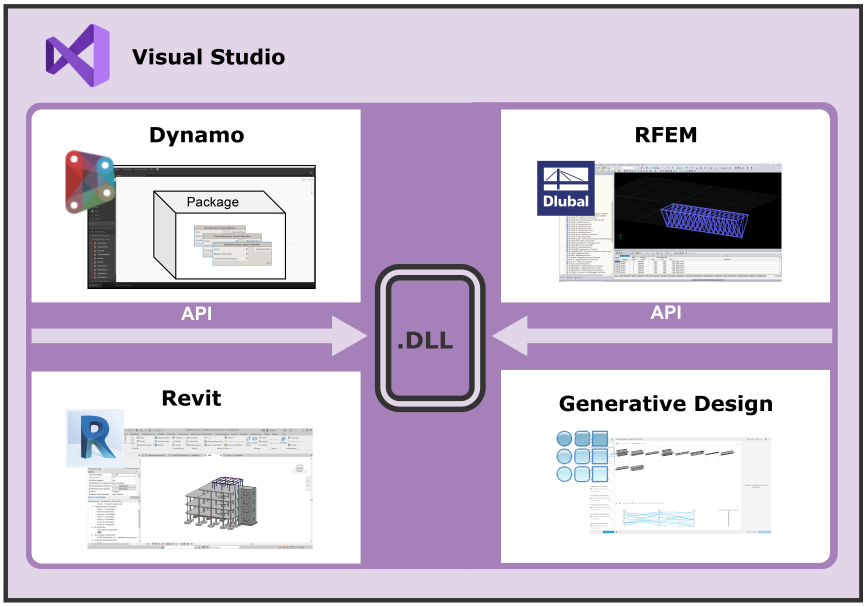


Abbildung 4.3.: Technischer Aufbau der Schnittstelle [5][7][9]

Aufgrund des parametrischen Konstruktionsansatzes ist die Modellierung der Tragstruktur an Dynamo gebunden. Dynamo bietet die Möglichkeit eigene Softwarepakete zu implementieren, die dann Funktionalität in Form von Knoten (siehe Unterabschnitt 3.3.2) bereitstellt. Darauf aufbauend findet die Kopplung der Programme über eine explizite Integration in Dynamo statt. In Abbildung 4.3 ist der technische Aufbau der Schnittstelle abgebildet. Das Ziel der Entwicklung ist die Erstellung einer Klassenbibliothek (.dll), die über ein sogenanntes *Dynamo-Package* eingebunden wird und über die enthaltenen Knoten das Modell überträgt. Als Grundlage der Entwicklung dient die Software Visual Studio. Hier wird ein Projekt erstellt, welches alle benötigen APIs der verwendeten Programme integriert und die enthaltenen programmspezifischen Funktionen nutzen kann. Versucht man dieses Projekt etwas alltagsbezogener zu beschreiben, entsteht ein abstrakter „Raum“ in dem sich die Programme „begegnen“ und eine Übersetzung der Sprachen erfolgt, damit sie sich „verstehen“. Der Übersetzer in diesem Fall ist die enthaltene Logik. Mittels des entstehenden Pakets wird dann die Möglichkeit geschaffen Dynamo, RFEM, Revit und Generative Design in einen größer gesehenen Kontext zu verknüpfen.
Ein Konzept, das der Paket-Entwicklung von Dynamo zugrunde liegt, nennt sich *Zero-Touch-Nodes*. Die Namensgebung dieses Konzepts basiert auf der Anwendungsart der entstehenden Dynamo-Knoten. Es ist dem Anwender nicht möglich Änderungen an der enthaltenen Logik vorzunehmen und somit kann sinngemäß „keine Hand mehr angelegt werden“[5]. Dies hat zum Vorteil, dass der Anwender kein weiteres Wissen über interne Abläufe bzw. Logik haben muss. Es ist lediglich die Aufgabe alle benötigten Inputs der Knoten zu erstellen. Andererseits stellt dies aus Sicht des Entwicklers die Herausforderung alle auftretenden Eventualitäten zu berücksichtigen, die bei der Benutzung der Knoten entstehen. Sollte dem nicht so sein, ist die Nutzung eingeschränkt und die allgemeine Anwendbarkeit nicht gegeben. Im Rahmen dieser Entwicklung ist vorgesehen, dass nicht alle Eventualitäten berücksichtigt werden, da es sich um eine Prototyp-Entwicklung handelt. Der zuvor angesprochene dritte Schritt der Berechnung wird nach dem Übertragen des Modells in RFEM durchgeführt. Dafür ist die Voraussetzung zu erfüllen, dass das Modell alle statisch notwendigen Elemente enthält. Alle weiteren Schritte werden durch RFEM autark durchgeführt. Abschließend sollten die Ergebnisse an Dynamo übermittelt werden.
Wichtige Aspekte die bei der Entwicklung bedacht werden müssen, betreffen die eindeutige Beschreibung der Elemente über einen Identifikator und die Einsortierung der Elemente sowie deren Verarbeitungsreihenfolge. Weiterhin ist zu berücksichtigen, dass meist ein Tragsystem schnell zu hoher Komplexität neigt, besonders wenn viel Parameter mit in das Modell einfließen. Sofern es sich um Ingenieurbauwerke wie eine Brücke, Halle oder ein mehrstöckiges Gebäude handelt, entstehen viele einzelne Elemente, die übertragen werden müssen. Daher spielt die Performance, also die Leistungsfähigkeit der Schnittstelle, eine große Rolle und sollte direkt mit ins Konzept einfließen. Den größten Rechenaufwand erzeugt generell das Schreiben in den Speicher. Daher ist zu überlegen wann und wie oft geschrieben wird.

## 4.3. Aufgaben im Entwicklungsprozess

In diesem Abschnitt werden die essenziellen Aufgaben nochmals etwas theoretischer behandelt, um das grundlegende Verständnis für die erforderlichen Schritte zu erlangen. Es wird nochmals spezifischer auf die teilweise verwendete Software und die damit verbundenen Konfigurationen und Besonderheiten eingegangen. Im Fokus stehen die grundle-

genden Bereiche, die für die Entwicklung des entstehenden Prototyps relevant sind.

### 4.3.1. Projektkonfiguration in Visual Studio 2019

Einer der ersten Schritte zur Umsetzung der Schnittstelle ist das Bereitstellen der benötigten Entwicklungsumgebung. Wie im vorherigen Abschnitt 4.2 erwähnt, findet die Entwicklung der Software in der IDE Visual Studio statt. Mit dem Ziel eine Klassenbibliothek zu entwerfen, kann beginnend ein Projekt erstellt werden. Dafür bietet Visual Studio sogenannte *project-templates*, also Projektvorlagen, die bezüglich der entsprechenden Auswahl vorkonfiguriert sind. Die Einteilung dieser Vorlagen durch Visual Studio erfolgt dabei nach Anwendungstyp, Betriebssystem, Programmiersprache und dem verwendeten Framework. Zur Entscheidungsfindung wurde die Programmiersprache als erstes Kriterium herangezogen. Die Sprache C# bietet den großen Vorteil, aufbauend auf dem .Net-Framework, mit der API von Dynamo und RFEM kompatibel zu sein. Somit muss keine weitere Übersetzung stattfinden, was den Arbeitsaufwand sowie die Verständlichkeit des Codes erheblich vereinfacht. Visual Studio Community 2019 muss den Ansprüchen der Entwicklung angepasst werden, indem die entsprechenden Komponenten installiert werden. Das verwendete Projekt basiert auf dem .Net-Framework und benötigt folgende Integration:

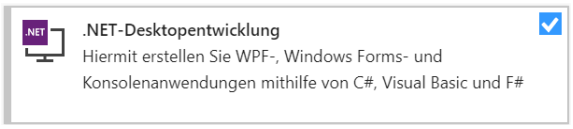


Abbildung 4.4.: Benötigte Visual Studio Integration zur Entwicklung mit .Net Framework

Für die Entwicklung mit C# wird das Paket mit dem Namen *.Net-Desktopentwicklung* implementiert, welches über den *Visual Studio Installer* bereitgestellt wird. Nach Abschluss dieser Integration ist die in Abbildung 4.5 dargestellte Projektvorlage verfügbar, die als Grundlage der Klassenentwicklung dient.


Abbildung 4.5.: Ausgewählte Visual Studio Projektvorlage für die Entwicklung der Klassenbibliothek

Dabei ist wichtig darauf zu achten welche .Net Framework Version das Projekt verwendet, da diese die Kompatibilität zu Revit und somit auch Dynamo bestimmt. Folgende Versionen sind miteinander kompatibel: [4]

- Revit 2021 nutzt .NET Framework 4.8
- Revit 2019 nutzt .NET Framework 4.7
- Revit 2017/2018 nutzt .NET Framework 4.6 (4.6.1 or 4.6.2)

Nachdem die Kompatibilität der Schnittstelle geklärt ist, können die zu verwendenden APIs ins Projekt eingebunden werden. Um diesen Schritt erheblich zu erleichtern, bietet es sich an den sogenannten *NuGet-Package-Manager* zu verwenden, welcher in Visual Studio integriert ist.

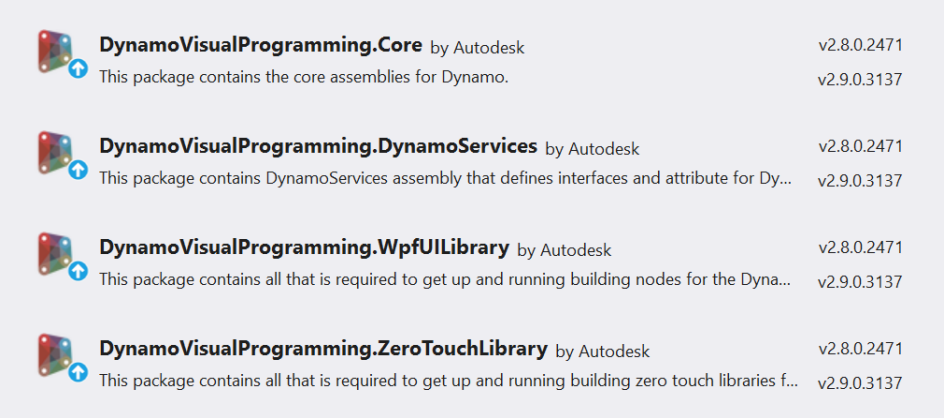


Abbildung 4.6.: Ausgewählte Nuget-Packages zur Integration der API von Dynamo

Über diesen können dann die in Abbildung 4.6 dargestellten Pakete integriert werden, welche alle benötigten *References* ins Projekt einbinden. Diese können anschließend im gleichnamigen Abschnitt der Projekteigenschaften wiedergefunden werden. Bei allen Referenzen handelt es sich um .dll-Dateien, also ebenfalls Bibliotheken. Das Paket DynamoVisualProgramming.ZeroTouchLibrary ermöglicht den Umgang mit den Geometrieobjekten in Dynamo. Dabei ist es abhängig vom Paket DynamoVisualProgramming.DynamoServices, das daher automatisch mit verknüpft wird. Das DynamoVisualProgramming.WpfUILibrary Paket wird benötigt, um die Benutzeroberfläche anzupassen, was in Unterabschnitt 4.3.4 durchgeführt wurde. Dieses besitzt eine Abhängigkeit zu DynamoVisualProgramming.Core. Mit diesen NuGet-Paketen wird lediglich der Zugang zu Dynamo geschaffen. Weiterhin ist die Anbindung an RFEM von Nöten. Dafür wird kein vorgefertigtes Paket angeboten und somit muss die Referenz händisch eingepflegt werden. Über die Funktion *Add Reference...* kann auf der lokalen Festplatte nach entsprechend diesen Bibliothek-Dateien gesucht werden. Zur Umsetzung des Workflows werden zwei Referenzen benötigt, die nur vorhanden sind, sofern das RFEM-Zusatzmodul RF-COM lokal installiert ist. Nachfolgend sind die lokalen Pfade aufgeführt:

- C:/Users/Public/Documents/Dlubal/SDK/Reference Assemblies/... x64/Dlubal.RFEM5.dll
- C:/Windows/assembly/GAC_ 64/Dlubal.STEEL_ EC3/... Version/Dlubal.STEEL_ EC3.dll

Über die Datei *Dlubal.RFEM5.dll* wird die API zum Hauptprogramm geladen. Für die Berechnung von Nachweisen aus dem Bereich Stahlbau des Eurocodes kann das Zusatzmodul mittels der Bibliothek *Dlubal.STEEL_ EC3.dll* geladen werden.
Für alle eingebundenen Referenzen ist zu beachten, dass die Eigenschaft *Copy Local* per Standard auf *true* gesetzt ist. Dieses sollte auf *false* abgeändert werden, da die Dateien sonst mit in die erstellte Klassenbibliothek kopiert werden und unnötig Speicher verschwendet wird (siehe Abbildung 4.7). Außerdem ist für die Bibliotheken von RFEM zu beachten, dass zusätzlich die Eigenschaft *Embed Interop Types* auf *false* gesetzt werden

muss, da sonst diverse Funktionen in RFEM nicht standardgemäß ausgeführt werden können.

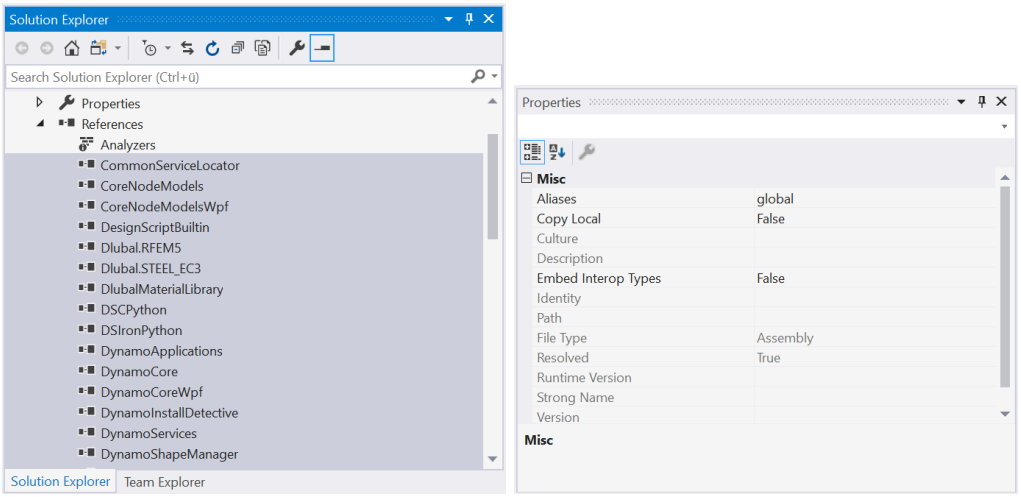


Abbildung 4.7.: Beispielausschnitt der verwendeten Referenzen und deren Eigenschaften.

Einen weiteren großen Vorteil bei der Verwendung einer IDE wie Visual Studio ist die integrierte Debugger-Umgebung. Unter dem Prozess des *debuggings* versteht man das Ausführen des entwickelten Codes unter Beobachtung des Debuggers. Im *Debugger-Mode* der IDE kann dann zu jedem Zeitpunkt der Ausführung die Applikation pausiert und der interne Speicher ausgelesen werden. Dies bietet den Vorteil, dass Variablen zur Laufzeit ausgelesen werden können und somit die Fehleranalyse um ein Vielfaches beschleunigt wird. Dies kann eingerichtet werden, indem über einen Rechtsklick auf das Projekt der Eigenschaften-Tab geöffnet und in das Menu „Debug“ gewechselt wird. Dort kann unter den „Startoptionen“ ein externes Programm hinzugefügt werden, welches dann vom Debugger gestartet und überwacht wird. Da Dynamo for Revit verwendet wird, muss dort auf die *executable* von Revit verwiesen werden. Ein bekannter Fehler der beim Debuggen von Revit das Programm zum Absturz bringt, kann vermieden werden, indem folgende Einstellung in Visual Studio vorgenommen wird [4]:

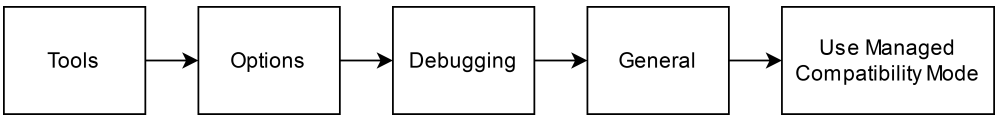


Abbildung 4.8.: Schrittabfolge zur Konfiguration des Debugmodus

In der nachfolgenden Abbildung 4.9 ist die ausgewählte Konfiguration abgebildet. Für das komfortablere Entwickeln eines Dynamo-Pakets können noch zusätzliche Einstellungen vorgenommen werden.

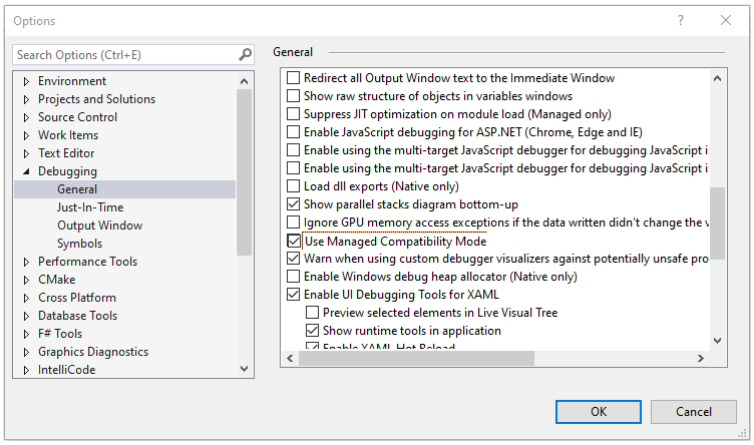


Abbildung 4.9.: Einstellung in Visual Studio zum Debuggen innerhalb von Revit

Der Prozess der Erstellung, auch *build* genannt, kompiliert für das Projekt die gewünschte .dll-Datei und legt sie in den dafür vorgesehenen Projektordner. Diese Datei muss anschließend, um in Dynamo eingelesen zu werden, in einen Paket-Ordner geladen werden. Diese Aktion kann von Visual Studio automatisch übernommen werden. Es bietet den großen Vorteil, dass sofern ein neuer *build* des Projekts erstellt wird, über die zuvor erwähnte Konfiguration Revit gestartet und die neue Bibliothek direkt exportiert wird. Der Arbeitsablauf wird beschleunigt und das manuelle Kopieren von Dateien entfällt. Eingerichtet werden kann dies wie folgt:
Im Eigenschaften-Menu des Projekts befindet sich der Reiter „Build Events“. Dort kann unter „Post-build event command line“ der Befehl zum Kopieren der benötigten Dateien hinzugefügt werden.

```
1
2  xcopy /Y "C:\Users\%name%\pfad\zum\projekt\bin\Debug\library.dll"
3  "$(AppData)\Dynamo\Dynamo Revit\2.5\packages\%packagename%\bin\ "
4
5  xcopy /Y "C:\Users\%name%\pfad\zum\projekt\src\package\pkg.json"
6  "$(AppData)\Dynamo\Dynamo Revit\2.5\packages\%packagename%\ "
7
```

Code 4.1: Befehle zum Kopieren der erstellten .dll- und .json-Datei in den Dynamo-Paket-Ordner

In Code 4.1 kann in der 2. und 3. Zeile der Befehl zum Kopieren der Bibliotheksdatei betrachtet werden. Die Funktion *xcopy /Y* ist ein Windows-Befehl, der das Kopieren vom ersten angegebenen Pfad zum zweiten übernimmt. Das */Y* besagt, dass alle kopierten Dateien am Zielort ohne Benutzeraufforderung überschrieben werden können. Der Befehl in Zeile 5-6 ist identisch seiner Funktion, wohingegen dieser eine .json-Datei kopiert, die für ein Dynamo-Paket erforderlich ist.

### 4.3.2. Entwicklung des Dynamo Pakets

Ein Dynamo-Paket ist ein komfortabler Weg, um neue Funktionalität in Dynamo einzubringen. Alternativ könnte auch die Bibliotheksdatei direkt in Dynamo eingebunden werden, was jedoch unflexibler ist, da der Debug-Prozess aufgrund der danach folgenden manuellen Arbeitsschritte verkompliziert wird. Ein Dynamo-Paket kann lokal erstellt

und bei Fertigstellung weiterführend für die Verwendung Dritter freigegeben werden. Um dies zu gewährleisten benötigt das Paket eine vorgegebene Struktur auf Dateisystemebene. Der Ablageort jeglicher integrierter Pakete befindet sich unter:

```
"%Appdata%\Dynamo\Dynamo Revit\2.5\packages"
```

Code 4.2: Allgemeiner Pfad zur Ablage von installierten Dynamo-Paketen

Generell werden beim Start von Dynamo alle Pakete unter dem in Code 4.2 angegeben Pfad geladen. Voraussetzung dafür ist folgende Struktur innerhalb des Paketordners [4]:

- *bin* - dieser Ordner enthält alle .dll-Dateien und die kopierten Referenzen, falls benötigt
- *dyf* - in diesem Ordner werden *customnodes* abgelegt (nicht verwendet für die Entwicklung)
- *extra* - dieser Ordner beherbergt alle zusätzlichen Dateien (.dyn, .svg, .xls, .jpeg, .sat, etc.)
- *pkg.json* - dies ist eine Datei und kein Ordner, diese enthält alle Paket-Eigenschaften in textueller Form

Wichtig für die Paketerstellung und dessen Veröffentlichung ist die Beachtung zukünftiger Updates von Dynamo. Weiterentwicklungen können dazu führen, dass bereits verwendete Funktionen geändert oder entfernt werden. Dies kann in einer ungewollten Funktionsunfähigkeit enden. Da mit dem Anlegen eines lokalen Dynamo-Pakets alle Startvoraussetzungen zum Entwickeln gegeben sind, wird im nachfolgenden Abschnitt betrachtet, wie der Inhalt in der Bibliothek erstellt wird.

### 4.3.3. Erstellen eines Zero-Touch-Knotens in Visual Studio

Wie schon in Abschnitt 4.2 angesprochen, wird eine Zero-Touch-Node-Bibliothek entstehen. Umgesetzt wird diese in C# und im Kontext der objektorientierten Programmierung. Das Ziel ist die Erstellung von Dynamo-Knoten. Dafür ist zunächst zu ermitteln, welche Art von Knoten erstellt werden kann. Es gibt folgende drei Typen von Knoten, die jeweils mittels eines Symbols gekennzeichnet werden:


Constructor Node


Query Node


Action Node

Die unterschiedlichen Knotentypen erfüllen verschiedene Aufgaben und sind mit dem OOP stark assoziiert. Die *Constructor Nodes* erstellen Objekte einer Klasse und bedienen sich den angelegten Konstruktoren-Methoden dieser. Die *Query Nodes* dahingegen fragen Attribute einer Klasse ab und die *Action Nodes* führen dementsprechend Methoden der Klasse aus. Dynamo ist dazu in der Lage den in der .dll-Datei enthaltenen Quellcode zu lesen, entsprechend zu gruppieren und zu interpretieren. Im Folgenden wird anhand eines Beispiels aufgezeigt, wie Dynamo aus einer entwickelten Klasse Knoten und

somit Funktionen generiert.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DynamoClassLibrary
{
  public class ExampleClass
  {
  //generiert Query Node
    public string name { get; set;}

  //generiert keinen Query Node
    private double number { get; set; }

  //Konstruktor - generiert Constructor Node
    public ExampleClass(string objectName) {
      this.name = objectName;
    }
  //Konstruktor - generiert keinen Constructor Node
    private ExampleClass(string objectName, double objectNumber)
    {
      this.name = objectName;
      this.number = objectNumber;
    }

  // generiert einen Action Node mit Input der Klasse ExampleClass
    public int calculateNumber(int inputNumber)
    {
      return inputNumber + 2;
    }
  // generiert Action Node mit Input zweiter double
    public static double calculateDoubleNumber(double inputDouble1,
      double inputDouble2)
    {
      return inputDouble1 + inputDouble2 + 0.6666666;
    }
  }
}
```

Code 4.3: Quellcode der Beispiel-Bibliothek DynamoClassLibrary.dll

Im oberen Code 4.3 ist der Quellcode, der in Dynamo eingebundenen Klassenbibliothek namens *DynamoClassLibrary.dll* zu finden. Hierbei handelt es sich um die Klasse „ExampleClass“, anhand derer die grundlegenden Dynamo-Mechanismen gezeigt werden können, wie zum Beispiel das Interpretieren des Quellcodes. In der nachfolgenden Abbildung sind die interpretierten Knoten und der Aufbau der Klasse abgebildet.

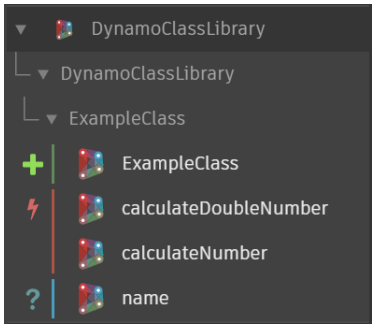


Abbildung 4.10.: Ergebnis der in Dynamo eingebundenen DynamoClassLibrary.dll

Mit dem grünen Plus-Symbol gekennzeichnet ist der Konstruktorknoten zu finden, der über die Zeilen 17-19 in Code 4.3 erstellt wird. Hier wird die Sichtbarkeit *public* verwendet, was Dynamo erlaubt den nachfolgenden Code zu interpretieren und als Knoten einzubinden. Die Sichtbarkeit *private* dahingegen berechtigt Dynamo nicht zum Anlegen eines Knotens (siehe Konstruktor Zeile 22 bis 26). Dieser Code wird ignoriert und kann nur für interne Funktionen genutzt werden.
Die beiden in Abbildung 4.10 dargestellten *Action Nodes* „calculateDoubleNumber“ und „calulateNumber“ werden durch die Zeilen 34-38 und 29-32 repräsentiert. Aufgrund deren Sichtbarkeit *public*, werden diese als Knoten angelegt. Der Unterschied der beiden Methoden besteht im Keyword *static*. Aus programmtechnischer Sicht wird damit ermöglicht, dass man Klassen, Methoden und Parameter, sofern sie als *static* deklariert wurden, auch ohne die sonst benötigte Instanziierung eines Objekts verwenden kann. Im Kontext von Dynamo wird somit kein Konstruktorknoten benötigt, um ein Objekt zu erstellen, da die Methode ohne auskommt. Der Unterschied der beiden Methoden kann anhand der dargestellten Knoten in Abbildung 4.11 nachvollzogen werden. Der Knoten „calculateNumber“ besitzt einen zusätzlichen Input, der eine „exampleClass“ fordert. Da die Methode nicht *static* ist wird eine konkrete Instanz benötigt, um die Methode auszuführen. Bereit gestellt wird diese durch den vorgelagerten Konstruktorknoten. „calculateDoubleNumber“ kommt dahingegen ohne Objekt aus, da die Methode *static* ist.

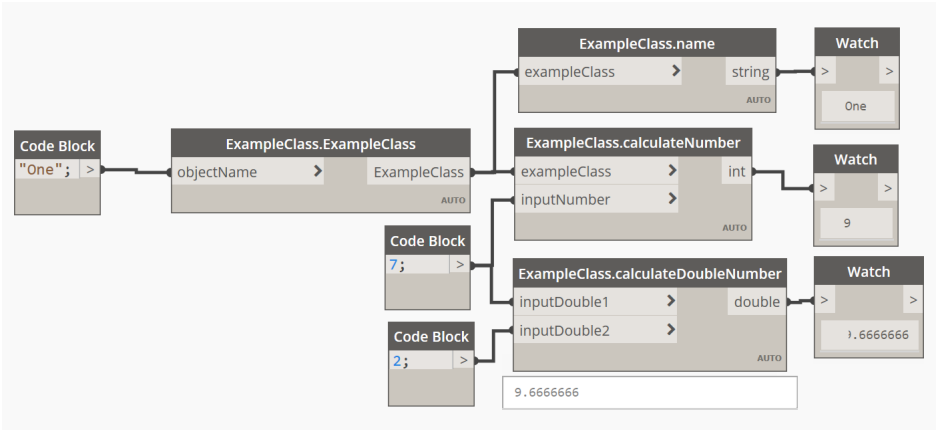


Abbildung 4.11.: *graph* mit den ausgeführten Knoten der Bibliothek DynamoClassLibrary.dll und deren Ergebnisse

Der Query Node „name“ referenziert auf die Zeile 12 in Code 4.3. Dort wird das Attribut der Klasse erstellt und ist mittels *get* und *set* erreichbar. Aufgrund der gewählten Sichtbarkeit *public* ist dieses Attribut erreichbar, im Gegensatz zu „number“ in Zeile 15. Dies sind die grundlegenden Verfahren zur Erstellung der drei verschiedenen Knotentypen. Allgemein kann gesagt werden, dass der Input eines Knotens über das Keyword *static* und deren Parameterliste gesteuert werden kann, sofern es sich um eine Methode handelt. Der Output dahingegen wird im Allgemeinen über den Rückgabe- beziehungsweise Dateityp gesteuert.

### 4.3.4. Erweiterung der UI von Dynamo und seinen Knoten

Eine weitere wichtige Fragestellung dreht sich um die Anpassung der Benutzeroberfläche von Knoten und in wie weit dies möglich ist mit einer Zero-Touch-Entwicklung. Eine Anpassung der Benutzeroberfläche ermöglicht dem Entwickler die Anwendbarkeit und Benutzerfreundlichkeit auf ein neues Level zu heben. Zum Beispiel kann eine zusätzliche Funktion direkt mit in die Knoten aufgenommen werden, die eigentlich einen zusätzlichen Knoten bedürfte. Eine Möglichkeit der Anpassung der Oberfläche eines Knotens ist in Abbildung 4.12 dargestellt. Hier kann eine essenzielle Größe für die Erstellung eines Gitters mit in die Oberfläche eingearbeitet werden. Durch die Integration des Sliders in den Knoten selbst, kann die gewünschte Funktionalität direkter und verständlicher designet werden.

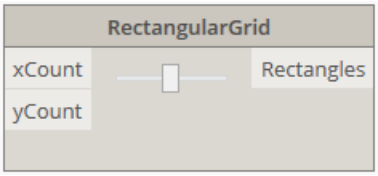


Abbildung 4.12.: Beispiel einer möglichen Anpassung der Benutzeroberfläche

Diese Anpassung und viele weitere sind möglich. Doch dafür wird weiteres Verständnis für den Aufbau einer Benutzeroberfläche in Dynamo benötigt. Die Softwarearchitektur namens *Model-View-Viewmodel* (MVVM) wird daher zunächst betrachtet, da diese die Basis darstellt.
Das MVVM-Entwurfsmuster entkoppelt die UI von der Datenstruktur, was in Abbildung 4.13 konzeptionell verdeutlicht ist. Die Verknüpfung der Benutzeroberfläche eines Knotens mit seiner Datenstruktur, wird automatisch von Dynamo übernommen. Um einen benutzerdefinierte UI zu erstellen, muss dann noch die Logik für das sogenannte *binding* zwischen der eigentlichen View-Ebene und der Modellebene gestaltet werden. Auf einer hohen Betrachtungsebene des MVVM müssen dafür die zwei folgenden Schritte durchgeführt werden. Zuerst muss das *model* (Knotenmodell) zur Definition der internen Logik beschrieben werden und anschließend die dazugehörige *view*-Klasse. Diese Klasse beschreibt, wie das Knotenmodell dargestellt werden soll. Diese Herangehensweise benötigt fortgeschrittene Kenntnisse im Umgang mit Windows Presentation Foundation (WPF) und gestaltet sich daher schwieriger. Für eine detaillierte Beschreibung dieser Entwicklung ist auf die Quellen [4] und [18] verwiesen.

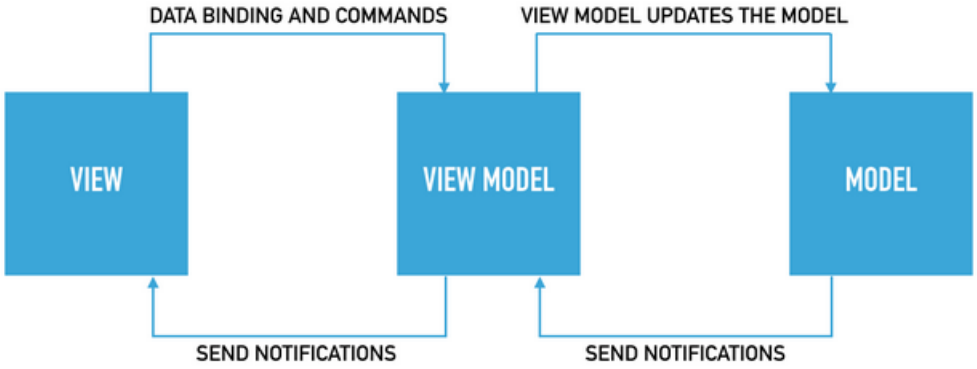


Abbildung 4.13.: Konzeptionelle Darstellung des Entwurfsmusters Model-View-Viewmodel [4]

Eine weitere und durchaus schnellere Modifikation von Knoten, kann durch die Möglichkeit der Vererbung geboten werden. Im späteren Abschnitt 5.2 wird gezeigt, wie ein Knoten um eine simple Dropdown-Liste erweitert werden kann. Generell ist es wichtig, dass Anpassungen der Benutzeroberfläche als ein Paket in Dynamo integriert werden.
Dynamo bietet weiterhin die Möglichkeit nicht nur Knoten zu ändern, sondern auch die Benutzeroberfläche von Dynamo selbst und die Erweiterung eigener Module mit einer grafischen Benutzeroberfläche. Im Rahmen dieser Arbeit wurde sich mit dieser Möglichkeit nur geringfügig beschäftigt und daher findet diese Thematik nur hier Erwähnung. Sollte diesbezüglich weiteres Interesse bestehen, sind folgende Quellen hilfreich: [4][18][19].

# 5. Entwicklungsdokumentation

In diesem Abschnitt wird der chronologische Entwicklungsprozess der Software detailliert erfasst. Es werden ausgewählte Sachverhalte dargestellt, die im Entwicklungsprozess einen Meilenstein darstellten oder eine Besonderheit beziehungsweise Lösung für eine Problematik bezeichnen. Somit stellt dieses Kapitel keine Anleitung dar, sondern eher eine Dokumentation des Wesentlichen. Auf spezielle technische Hintergründe wird nicht in allen Teilen eingegangen, da das Hauptaugenmerk des Abschnitts auf der Software an sich liegt. Für weitere Informationen zu technischen Hintergründen ist hiermit auf die Quellen [4] und [18] verwiesen.

## 5.1. Anlegen eines Knotens sowie Stabs in RFEM

Als eine der ersten zu realisierenden Aufgaben gilt es Punkte, die in Dynamo existieren, nach RFEM zu exportieren, da sie essenziell für die Erstellung komplexerer Strukturen sind. Im Visual Studio Projekt, welches in Unterabschnitt 4.3.1 erläutert wurde, wird eine C#- Klasse angelegt, in der die Entwicklung startet.
Der allgemeine Workflow zum Anlegen eines Elements in RFEM kann wie folgt standardisiert werden:

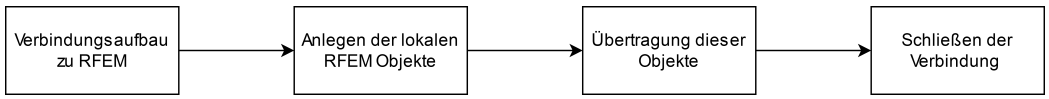


Abbildung 5.1.: Standardroutine des Elementexports zu RFEM

Der erste Schritt ist das Etablieren einer Verbindung zu RFEM. Dafür kann die zur Schnittstelle RF-COM 5.7 mitgelieferte Dokumentation verwendet werden. Dort werden die wichtigsten Schritte zum Verbindungsaufbau aufgeführt und Hinweise gegeben. Weiterhin bietet Sie eine Übersicht über die ansprechbaren Klassen und deren Verwendung. Die Herstellung einer Verbindung kann wie folgt durchgeführt werden:

```
// abrufen Schnittstelle zur laufenden RFEM Appliation
  IApplication app = Marshal.GetActiveObject("RFEM5.Application")
  as IApplication;

// ueberprueft Lizenz und sperrt RFEM
  app.LockLicense();

// abrufen Schnittstelle zum aktiven Modell
  IModel model = app.GetActiveModel();

// abrufen Modelldaten aus dem aktiven Modell
  IModelData modelData = model.GetModelData();

// Logik zur Uebertragung der Elemente
  ...
```

```
17
18 // entsperrt die Applikation
19     app.UnlockLicense();
20
```

Code 5.1: Herstellen einer Verbindung von Dynamo zu RFEM über RF-COM 5.7

Der Zugriff auf die interne Klassenstruktur erfolgt mittels RF-COM durch Schnittstellen-Klassen, die über ein großes „I“ im Namen gekennzeichnet sind, wie zum Beispiel: „IApplication, IModel, IModelData“. In Zeile 3 und 4 von Code 5.1 wird die geöffnete Session von RFEM der Variable *app* zugewiesen. Über diese Schnittstelle kann dann auf das geöffnete Modell zugegriffen werden (siehe Zeile 10). Für diese Anweisungen ist es erforderlich, dass sowohl die Anwendung selbst als auch ein Modell darin geöffnet sind, da diese Befehle sonst in einer Fehlermeldung enden. Um jedoch auf ein Modell und dessen Modelldaten zugreifen zu können, muss zuvor der Sperr-Befehl in Zeile 7 ausgeführt werden. Dieser bewirkt, dass die Oberfläche von RFEM unnutzbar wird und somit eine manuelle Änderung von Modelldaten unterbunden wird. In Zeile 13 werden dann die Modelldaten des aktiven Modells ausgelesen, in denen sich alle geometrischen Elemente befinden. Hier müssen später alle Elemente wie beispielsweise Knoten, Linien und Stäbe hinzugefügt werden. Dazu müssen diese erst einmal angelegt werden. Nach der Zeile 13 können dann Änderungen vorgenommen werden. Die Routine wird abgeschlossen mit dem Befehl der Zeile 18. Dort wird die Anwendung wieder entsperrt und mit dem Ende dieser Methode werden alle Variablen freigegeben und die Verbindung beendet.
Nachdem geklärt ist, wie eine Verbindung zu RFEM hergestellt werden kann, muss ermittelt werden, was für das Anlegen eines Knotens in RFEM benötigt wird und woher diese Informationen kommen. In der zuvor gezeigten Abbildung 3.9 ist dargestellt, wie ein Punkt in Dynamo erstellt wird. Definiert wird er nur über seinen Ort im Raum. Der erstellende Knoten gibt ein Objekt der Klasse *Autodesk.DesignScript.Geometry.Point* zurück, welche die benötigten Informationen bezüglich der dreidimensionalen Koordinaten enthält und somit die Informationsquelle symbolisiert. Die enthaltenen Informationen müssen anschließend einem Knoten von RFEM übergeben werden. Angelegt wird ein RFEM Knoten wie folgt:

```
1
2 // Aufruf des Konstruktors
3   Dlubal.RFEM5.Node node = new Dlubal.RFEM5.Node();
4
5 // Zuweisung der Parameter
6   node.No = 1;
7   node.X = 10;
8   node.Y = 20;
9   node.Z = 30;
10
```

Code 5.2: Anlegen eines Knotens für RFEM

Damit die Informationen übergeben werden können, muss zuerst ein Knoten angelegt werden (siehe Zeile 3). Das entstandene Objekt *node* wird „leer“ angelegt, da dem Konstruktor keine Parameter übergeben werden. Somit ist der Knoten mit Standardwerten versehen, die im Nachhinein angepasst werden müssen. Das Attribut *No* eines Knotens der Klasse *Dlubal.RFEM5.Node* spielt eine besondere Rolle. Die *No*, auch Nummer, gilt für einen Knoten als Attribut zur eindeutigen Identifikation. Wird zum Beispiel ein Kno-

ten mit der Nummer 1 zehn Mal in Folge mit unterschiedlichen Koordinaten angelegt, so existiert letztendlich nur der letzte in der Reihenfolge, da sich alle gegenseitig überschreiben. Dies gilt nicht nur für Knoten, sondern alle Elemente in RFEM und daher muss ein besonderes Augenmerk auf die richtige Vergabe der Nummern gelegt werden. In den Zeilen 7 bis 9 werden die X-,Y-,Z-Koordinaten festgelegt und der Punkt ist für RFEM vollständig und eindeutig beschrieben.
Nachdem nun der Knoten im lokalen Kontext angelegt wurde, muss er abschließend an RFEM übergeben werden. Wie schon zuvor erwähnt, werden die Änderungen im Modell und dessen Modelldaten vorgenommen. Letzteren müssen die angelegten Elemente übergeben werden. Die dafür benötigten Befehle sehen wie folgt aus:

```
1
2  // Vorbereitung zur Anpassung
3    modelData.PrepareModification();
4
5  // Uebergage eines Knotens
6    modelData.SetNode(node);
7
8  // Abschluss der Anpassung
9    modelData.FinishModification();
10
```

Code 5.3: Übergeben von Elementen an die Modelldaten

Für den Export von lokalen Elementen zu RFEM werden immer die in Code 5.3 gezeigten Befehle benötigt. Voraussetzung dafür ist, dass eine Verbindung hergestellt wurde und die Modelldaten des Modells ausgelesen wurden sowie in der Variable *modelData* gespeichert sind. Um den zuvor angelegten Punkt beziehungsweise Knoten zu exportieren, müssen die Modelldaten auf das Schreiben vorbereitet werden (siehe Zeile 3, Code 5.3). Daraufhin können die Elemente in die Modelldaten geschrieben und abschließend der Transfer über Zeile 6 abgeschlossen werden. Nachdem dies durchgeführt wurde, sollten die Elemente im RFEM erscheinen. Um den Prozess abzuschließen, muss dann noch die zuvor gesperrte Lizenz freigegeben werden.
Das Anlegen eines Stabs in RFEM fällt im Gegensatz zum Knoten komplexer aus. Dies ist nachfolgend in Code 5.4 dargestellt. Der abgebildete Pseudocode stellt nur die logische Ausführung der Methode *createMember()* dar und ist nicht voll funktionsfähig.

```
1
2  public static void createMember(AutodeskLine line)
3  {
4
5  // Aufbau der Verbindung zu RFEM
6    IApplication app = Marshal.GetActiveObject("RFEM5");
7    app.LockLicense();
8    IModel model = app.GetActiveModel();
9    IModelData modelData = model.GetModelData();
10
11 // Ermittung und Erstellung des Anfangs- und Endknotens
12   Point startpoint = line.PointAtParameter(0);
13   Point endpoint = line.PointAtParameter(1);
14
15   Node node1 = new Node();
16   Node node2 = new Node();
17
18   node1.No = 1;
```

```
    node1.X = startpoint.X;
    node1.Y = startpoint.Y;
    node1.Z = startpoint.Z;

    node2.No = 2;
    node2.X = endpoint.X;
    node2.Y = endpoint.Y;
    node2.Z = endpoint.Z;

  // Erstellung einer Linie anhand der Knoten
    Dlubal.RFEM5.Line line1 = new Dlubal.RFEM5.Line();
    line1.NodeList = "1,2";

  // Erstellung eines Materials
    Material mat1 = new Material();
    mat1.TextID = "Beton C30/37";
    mat1.No = 1;

  // Erstellung eines Querschnitts
    CrossSection croS1 = new CrossSection();
    croS1.TextID = "IPE 100";
    croS1.MaterialNo = 1;
    croS1.No = 1;

  // Erstellung eines Stabs
    Member mem1 = new Member();
    mem1.LineNo = 1;
    mem1.StartCrossSectionNo = 1;
    mem1.No = 1;

  // Schreiben der Daten ins Modell
    modelData.PrepareModification();
    modelData.SetNodes([ref node1,ref node2]);
    modelData.SetLine(ref line1);
    modelData.SetMaterial(ref mat1);
    modelData.SetCrossSection(ref croS1);
    modelData.SetMember(ref mem1);
    modelData.FinishModification();

  // Ende der Verbindung zu RFEM
    app.UnlockLicense();
  }
```

Code 5.4: Erstellen eines Stabs als Pseudocode

In diesem Beispiel wird über das Keyword *static* der zusätzliche Input der eigenen Klasse des Dynamoknoten unterbunden, da es nicht notwendig ist ein Objekt zu erstellen. Die Methode verlangt als einzigen Parameter eine Dynamo-Linie, die am Knoten als Input erscheint. Im Methodenrumpf selbst beginnt die Logik. So muss eingangs eine Verbindung zu RFEM hergestellt werden und alle relevanten Modellinformationen geladen werden. In einem zweiten Schritt werden der Anfangs- und Endpunkt dem Input-Parameter *line* entnommen und äquivalente Repräsentationen im RFEM-Kontext geschafften (siehe Zeile 10-27). Mittels dieser Punkte kann dann eine RFEM-Linie erzeugt werden, die für die Erstellung eines Stabs essenziell ist, da ein Stab nur mittels einer Linie angelegt werden kann. Außerdem besitzt ein Stab weitere statische Informationen wie das Material (siehe Zeile 32-36) sowie einen Querschnitt (siehe 37-41), die ebenfalls angelegt werden.

Abschließend werden alle erstellten Objekte dem Modell übergeben (siehe Zeile 49-57) und die Verbindung zu RFEM beendet. Über diese Methode ist es nun möglich, einen einzigen Stab in RFEM parametrisch anzulegen.
Der nächste Schritt zur angestrebten Funktionalität ist das Anlegen mehrere Stäbe zur gleichen Zeit. Dafür muss die Methode in Code 5.4 nur geringfügig erweitert werden. Es muss lediglich erreicht werden, dass mehrere Linien als Input angegeben werden können und diese Knoten-intern entsprechend verarbeitet werden. Dafür wurde der Dateityp des Parameters *line* zu einer Liste umgebaut und dieser intern berücksichtigt. Dies geschieht durch das mehrfache Erstellen von Objekten, die in Listen verarbeitet werden, wobei darauf geachtet werden muss, dass sich der eindeutige Identifikator *Nummer/No* richtig anlegt, um keine Objekte zu überschreiben. Somit wurde *No* stets mit jedem Ausführen des *graphs* mit der 1 initialisiert und schrittweise vergrößert.
Es wurde identifiziert, dass besonders auf die Identifikatoren geachtet werden muss, sofern es sich um Konstruktionen handelt, bei denen mehrere Stäbe an einen Knoten angeschlossen sind. Diese Thematik wird genauer in Abschnitt 5.3 betrachtet. Das Ergebnis der ersten Methode zur mehrfachen Erzeugung von Stäben kann in der nachfolgenden Abbildung betrachtet werden.

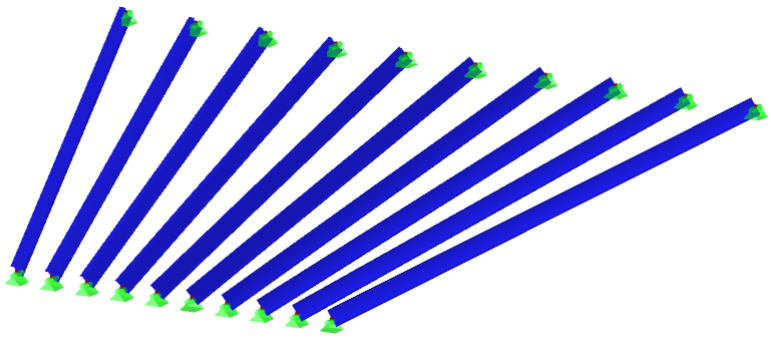

Abbildung 5.2.: Erstmaliges Anlegen mehrerer Stäbe in RFEM

## 5.2. Erstellung eines Knotens mit benutzerdefinierter Oberfläche

Nachdem das grundlegende Übertragen von Elementen geklärt war, wurde sich damit beschäftigt eine benutzerdefinierte Oberfläche zu schaffen. Diese Thematik wurde allgemein in Unterabschnitt 4.3.4 schon einmal beschrieben. Ein guter Anwendungsfall für die Anpassung der UI bietet sich bei den Dynamoknoten zur Erstellung von „Material“ und „Querschnitt“. Es soll eine Dropdown-Liste in die Knoten eingefügt werden, da dies die Anwenderfreundlichkeit steigert. Als Output der Knoten soll eine Zeichenkette dienen, die der jeweiligen *TextID* nachempfunden ist (siehe Code 5.4 Zeile 33 und 38). Die Erzeugung der Elemente selbst ist nicht in die Knoten integriert. Der Sinn dahinter besteht darin, die benötigten Informationen zu erstellen und diese dann an Knoten weiterzuleiten, die die Erstellung übernehmen. Nicht alle vorhandenen Materialien und Querschnitte werden dabei berücksichtigt, da diese Knoten nur das mögliche Potential aufzeigen sollen. Die Benutzeroberfläche eines Knotens kann durch zwei verschiedene Herangehensweisen angepasst werden. Zum einen kann die Benutzeroberfläche mittels

der WPF von Grund auf neu entwickelt oder über vorgefertigte Klassen manipuliert werden. Letzterer Lösungsansatz bedient sich dafür der Klasse *DSDropDownBase*, um eine Dropdown-Liste in einen Knoten zu integrieren. Dafür muss die Klasse erben und die geforderten Methoden implementieren, was dazu führt, dass die ganze Klasse nur auf einen Dynamo-Knoten begrenzt ist (zum Vergleich siehe Unterabschnitt 4.3.3).

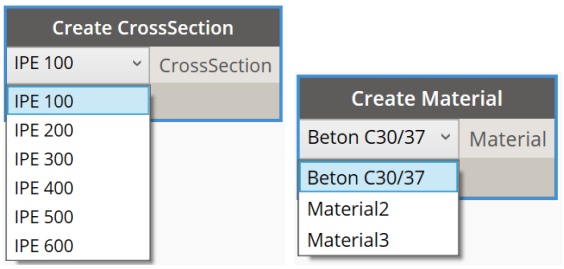


Abbildung 5.3.: Beispiel für Anpassung der Benutzeroberfläche anhand einer Dropdown-Liste in Knoten

Die gewünschten Drop-Down-Elemente können anschließend mittels einfacher *Strings* in die Knoten eingearbeitet werden (siehe Abbildung 5.3). Es ist zu erwähnen, dass die Anpassung mittels einer existierenden Klasse geringerer Aufwand bedeutet. Jedoch wird die Flexibilität der Gestaltung einschränkt, da jeweils nur ein Mal geerbt werden kann. Große Anpassungen können mit dieser Vorgehensweise nicht realisiert werden. Daher ist bei größeren Änderungen ein Neudesign mittels der WPF vorzusehen.
Außerdem wurde ein Anzeigefehler während des Debug-Prozesses festgestellt, der sich dahingehend äußert, dass ein zusätzlicher Output mit dem gleichen Namen erstellt wird und keiner dieser Outputs einen gültigen Wert zurückgibt. Dies stellte bei der Entwicklung einen Störfaktor dar, sodass von einer Benutzung der Knoten abgesehen wurde.

## 5.3. Anlegen einer komplexen Stabstruktur

In diesem Abschnitt geht es um die Erweiterung der Funktionsfähigkeit der Methode createMember() (siehe Code 5.4) dahingehend, dass komplexe Stabtragwerke übertragen werden können. Unter komplexen Tragstabwerken werden Strukturen gezählt, die an einem Knotenpunkt mehrere Anschlüsse verschiedener Stäben besitzen. Dabei bezieht sich die Bezeichnung „komplex“ weniger auf die Geometrie als auf die technische Umsetzung eines schlüssigen Algorithmus.
Im nachfolgenden Code 5.5 ist der erste Entwurf der Methode zum Anlegen dieser Strukturen gezeigt. Die enthaltene Symbolik {...} bedeutet, dass sich der Code zur vorherigen Abbildung nicht verändert hat und die Funktion erhalten bleibt.

```
1
2  public static void createMultiMember(AutodeskLine[] lines)
3  {
4  //  Aufbau der Verbindung zu RFEM
5    {...}
6
7  //  Anlegen der Variablen und Indizes
8    int listIndex = 0;
9    int listNodeIndex = 0;
10   int elementID = 1;
11   int elementNodeID = 1;
```

```
  int lineCount = lines.Count();

//  Erstellen des Materials
  {...}

//  Erstellen des Querschnitts
  {...}

//  Erstellen der "leeren" Array
  RFEM5.Node[] RFEMnodes = new RFEM5.Node[lineCount * 2];
  RFEM5.Line[] RFEMlines = new RFEM5.Line[lineCount];
  RFEM5.Member[] RFEMmembers = new RFEM5.Member[lineCount];

  foreach (Autodesk.DesignScript.Geometry.Line line in lines)
  {
//  Ermittlung der Punkte
  Point startpoint = line.PointAtParameter(0);
  Point endpoint = line.PointAtParameter(1);

//  Erstellen des Anfangsknotens
  RFEMnodes[listNodeIndex].ID = elementNodeID.ToString();
  RFEMnodes[listNodeIndex].No = elementNodeID;
  RFEMnodes[listNodeIndex].X = startpoint.X;
  {..}

//  Erstellen des Endknotens
  RFEMnodes[listNodeIndex + 1].ID = (elementNodeID+1).ToString();
  RFEMnodes[listNodeIndex + 1].No = elementNodeID+1;
  RFEMnodes[listNodeIndex + 1].X = endpoint.X;

//  Erstellen der Linien
  Dlubal.RFEM5.Line line1 = new Dlubal.RFEM5.Line();
  REMlines[listIndex].NodeList = $"{RFEMnodes[listNodeIndex].ID},
    {RFEMnodes[listNodeIndex + 1].ID}";
  RFEMlines[listIndex].ID = elementID.ToString();
  RFEMlines[listIndex].No = elementID;

// Erstellen des Stabs
  Member mem1 = new Member();
  RFEMmembers[listIndex].LineNo = RFEMlines[listIndex].No;
  RFEMmembers[listIndex].StartCrossSectionNo = 1;
  RFEMmembers[listIndex].No = elementID;

// Erhoehen der Indizes
  elementID ++;
  listIndex ++;
  elementNodeID += 2;
  listNodeIndex += 2;

// Schreiben der Daten ins Modell
  modelData.PrepareModification();
  {...}
  modelData.FinishModification();

// Ende der Verbindung zu RFEM
  app.UnlockLicense();
}
```

Code 5.5: Erster Entwurf der Methode zur Erstellung komplexer Stabtragwerke

Die erste Übertragung der Modellgrößen an RFEM wurde bezüglich des Identifikators *No* kaum gesteuert. Es wurden fortlaufende Indizes benutzt, die entsprechend der Objektanzahl aufsummiert wurden. Punkte die zu mehreren Stäben gehören, wurde mehrfach gesetzt, ohne vorher zu sortieren oder abzufragen. Dies führte zu einem erheblich größeren Schreibaufwand als notwendig, was die Performance sogar schon bei kleineren Strukturen merklich beeinflusste. Dabei ist die Anzahl der geschriebenen Elemente nicht gleich der benötigen, was anhand der Beispielstruktur in Abbildung 5.4 einfach gezeigt werden kann. Insgesamt besteht dieser Fachwerkträger aus 131 Stäben und 51 Knoten. An einem Knoten sind bis zu fünf Stäbe angeschlossen, was eine geringe Zahl an Knoten im Vergleich zu den Stäben ergibt, bedenkt man die Tatsache, dass jeder Stab zwei eigenständige Knoten besitzen könnte. Abweichend von dieser geringen Anzahl an Knoten, müssen tatsächlich Knoten in Summe gleich $[2 \cdot \textit{Anzahl der Linen}]$ geschrieben werden. Aufgrund des Übergabeformats der in Dynamo erstellten Linien als Liste (siehe Zeile 2) besteht zwischen den einzelnen Linien noch kein Zusammenhang, der eine Verknüpfung impliziert. Die enthaltenen Objekte könnten in einer zufälligen vom Anwender abhängigen Reihenfolge angelegt werden. Dies sollte auch die Voraussetzung sein, um jegliche Eventualität der Liste abbilden zu können und somit volle Funktionsfähigkeit zu bieten. Bei dieser ersten Umsetzung wurde dies nicht berücksichtigt und stattdessen die Anzahl der indizierten Linien mal zwei gerechnet (siehe Zeile 21). Dies führte dazu, dass eine mehr als fünf-mal so große Anzahl an Knoten übergeben wurde als benötigt. In Zahlen sind das insgesamt 262 statt nur 51 Knoten.

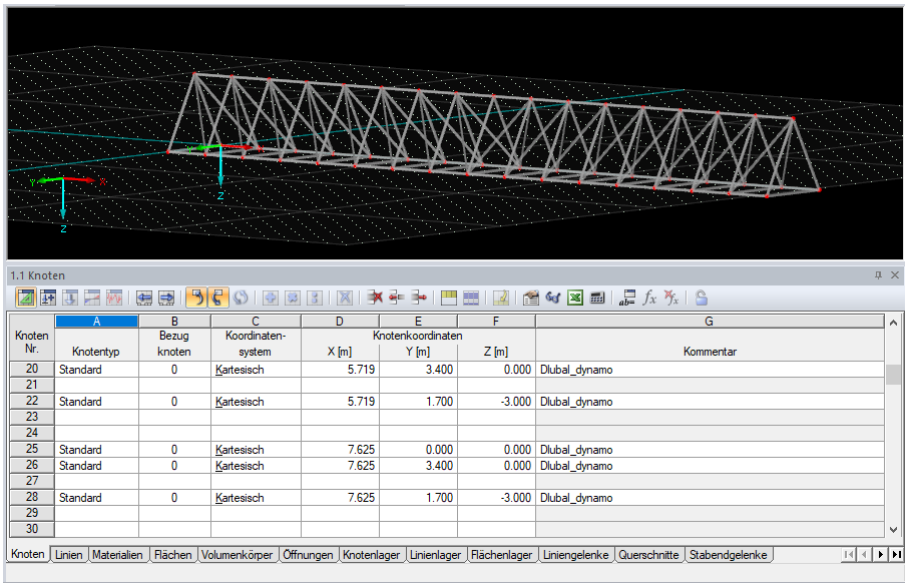


Abbildung 5.4.: Erste Ergebnisse der neuen Methode createMultiMember() und dessen Problemstellung

Die Auswirkungen dieser Methode auf das Modell kann in Abbildung 5.4 betrachtet werden. Auf den ersten Blick wurde die gewünschte Struktur an RFEM erfolgreich übergeben. Dennoch können im unteren Teil der Abbildung große Lücken in der Knotentabelle erkannt werden, die durch das mehrfache Hinzufügen von bereits existierenden Knoten entstehen. Die Indizes werden fortlaufend aufsummiert, was die Knoten verschiebt. Das Modell wird somit nicht korrekt übertragen und die Qualität sinkt.
Resultierend musste sich eine neue Logik zur Verwalten der Identifikatoren erarbeitet

werden, dem Sortieralgorithmus.

## 5.4. Sortieralgorithmus und Exportkonzeptwahl

Die Ausarbeitung eines Algorithmus zur Vergabe der richtigen Identifikatoren der zu übergebenden Elemente führte dazu, dass zuvor eine übergeordnete Fragestellung beantwortet werden musste. Dabei handelte es sich um die Gestaltung des Exports zu RFEM, da je nach Anzahl der Exportstellen sich der Vergabemechanismus ändert. Diese Thematik betreffend können zwei Richtungen eingeschlagen werden, die das Design der Schnittstelle grundlegend beeinflussen. Die Konzepte gestalten sich wie folgt:

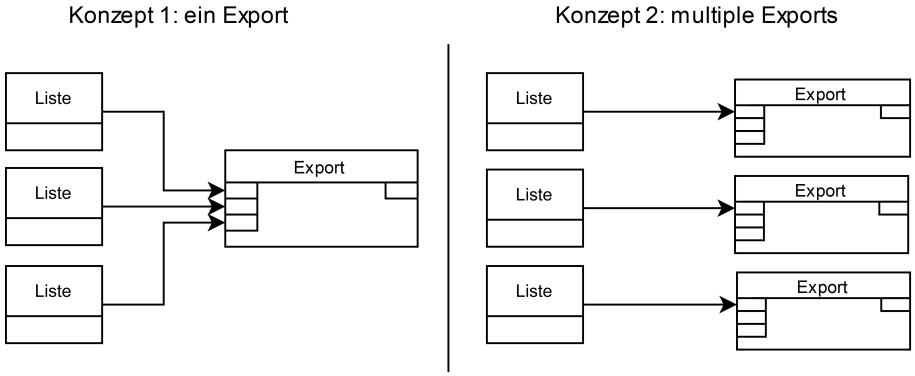


Abbildung 5.5.: Exportkonzepte der Schnittstelle

Zur Auswahl standen die Konzepte mittels eines einfachen oder mehrfachen Exports. Ersteres bietet den Vorteil, dass nicht mehrere Zugriffe auf RFEM benötigt werden, um die komplette Struktur anzulegen. Für den Algorithmus zur Vergabe der Identifikatoren würde dies ebenfalls bedeuten, dass keine Rücksicht auf andere Exporte genommen werden müsste und somit die Entwicklung simpel ausfallen würde.
Ein mehrfacher Export hätte wiederum den Vorteil, dass nur Teilstrukturen exportiert werden könnten und dies wiederum für mehr Flexibilität in der Gestaltung eines *graphs* sorgen würde. Betrachtet man zusätzlich die zuvor entwickelten Methoden in Code 5.4 und Code 5.5 wird deutlich, dass das dort verwendete Design schon dem Konzept 2 entspricht, da jeweils ein Export pro Knoten existiert. Aufgrund der Idee für eine Weiterentwicklung nur Teilstrukturen eines Modells anpassen zu können, ist der Export an mehreren Stellen vorteilhaft. Diese beiden Tatsachen bilden den ausschlaggebenden Faktor weshalb das Konzept 2 verfolgt wird. Mit dieser bewussten Entscheidung steigt die Komplexität des Designs für den Sortieralgorithmus erheblich an, da ein mehrfacher Zugriff auf die Modelldaten stattfindet. Dennoch drängt sich die Thematik des Multithreadings in den Vordergrund, da die Modelldaten nur durch mehrere Knoten angelegt werden können, sobald die mehrfache Verbindung zu RFEM realisiert werden kann. Tests diesbezüglich ergeben, dass ein paralleler Export von mehreren Knoten zu Fehlern im Modell oder sogar zum Absturz von RFEM führt. Es wurde statt eines parallelen ein sequenzieller Export mit mehreren Knoten angestrebt, um den Zugriff in Reihe zu schalten und somit eine Warteschlange für den RFEM-Zugriff zu schaffen. Der sequenzielle Export wird nun zur neuen Randbedingung, die beim Anlegen und Ausführen eines *graphs* zu berücksichtigen gilt. Daraufhin konnte abschließend mit der Implementierung des Sortieralgorithmus begonnen werden.

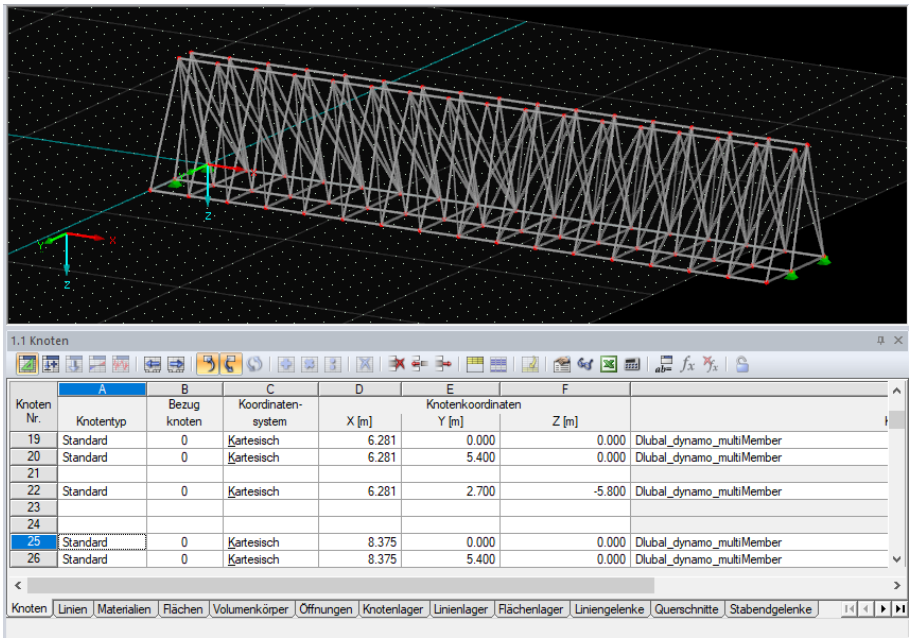


Abbildung 5.6.: Ergebnis der ersten Implementation von zwei Exportknoten in einem *graph*

Es wurde eine erste Logik entwickelt, die alle Export-Knoten befähigt auf vorherige Exports zu achten, indem die Bestandsobjekte dieser nicht überschrieben werden. Realisiert wurde dies über eine Abfrage des zuletzt vergebenen Identifikators, welcher anschließend fortlaufend vergeben wurde. Die ersten erkenntnisreichen Tests ergaben die Situation in Abbildung 5.6. Hier handelt es sich um eine doppelte Ausführung eines *graphs* im gleichen Modell. Es konnte erreicht werden, dass separat von den Stäben die Auflager der Struktur exportiert werden konnten. Jedoch ist auch deutlich zu erkennen, dass aufgrund des globalen Auslesens der letzten verwendeten Nummerierungen das vorherige Modell überschrieben wird. Mit der erneuten Ausführung des *graphs* wurde die vorherige Struktur erkannt und die Identifikatoren entsprechend weitergeführt, was in der Verschmelzung der Modelle resultierte.
Für diese Problematik wurden erneut zwei mögliche Lösungen ausgearbeitet. Es konnte einerseits eine Lösung herbeigeführt werden, indem die vorherigen beziehungsweise alten Modelldaten gelöscht werden bevor die neuen erstellt werden. So wird gewährleistet, dass immer in einen leeren Datensatz geschrieben wird und eine Verschmelzung nicht stattfindet. Diese Lösung hat jedoch zum Nachteil, dass jede manuelle Änderung beziehungsweise Konfiguration in RFEM mit erneuter Ausführung der Knoten gelöscht wird. Daher ist das Ändern von bestehenden Daten eine sinnvolle Bestrebung. Genau diesen Ansatz verfolgt die zweite mögliche Lösungsstrategie. Dabei handelt es sich um ein durchaus komplexeres Design von Klassen, da eine Art interne Datenbank entsteht, die die RFEM- und Dynamo-Elemente logisch miteinander verknüpft. Somit wird es möglich die Elemente nur noch anzupassen und nicht erneut zu schreiben, was bei großen Modellen vermutlich zu einer erheblichen Performance-Steigerung führt.
Aufgrund der begrenzten Bearbeitungszeit der Arbeit und der hohen Komplexität des letzteren Designs, wurde sich für das simple Löschen der Daten entschieden. Bei einer Weiterentwicklung dieser Schnittstelle sollte möglichst das alternative Design in Betracht

gezogen werden, da dies dem Anwender deutlich mehr Möglichkeiten bietet und die Arbeitseffizienz fördert. Die erste Umsetzung des Löschens der Daten wurde mittels eines separaten Knotens realisiert. Das Löschen soll zu Beginn des *graphs* angeordnet werden, um sicher zu stellen, dass die Modelldaten geleert werden. Dabei ist zu erwähnen, dass der Knoten alleinstehend, ohne Verbindung zu anderen Knoten angedacht wird. Wie sich in einer anschließenden Testphase herausstellte, verhält sich Dynamo nicht wie geplant. Es wurde erkannt, dass Knoten in Dynamo nur ausgeführt werden, sofern Sie eine Änderung erfahren. Dies kann durch das Ändern des Knoten selbst geschehen oder durch die Anpassung vorgelagerter über einen Pfad verbundener Knoten. Ein alleinstehender Knoten, der keinen Input benötigt, wird schlussfolgernd nur ein einziges Mal ausgeführt. Resultierend führte dies zu keiner Lösung und ein neuer Ansatz wurde benötigt. Die Lösung des Problems konnte über die Integration eines zusätzlichen *Boolean-flag* in jedem Export-Knoten erzielt werden. Das *flag* steuert eine globale Variable, die von allen Knoten abgefragt wird und somit das Löschen initiierte. Der Anwender bestimmt welcher Export der letzte im Knotennetz ist, um bei einer erneuten Ausführung das Löschen durch den ersten Knoten anzustoßen. Die Umsetzung dieses Knotens kann in Abbildung 5.7 betrachtet werden.

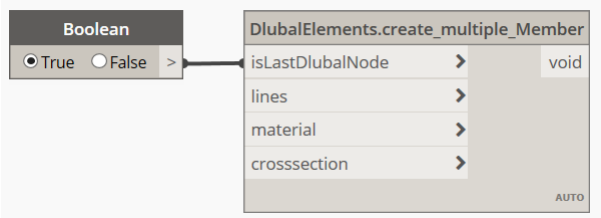


Abbildung 5.7.: Knotendesign zur Integration des Löschens der Modelldaten

## 5.5. Erstellung von Lagern und Stablasten

Für ein vollständiges FEM Berechnungsmodell wird nicht nur das statische System als Geometrie selbst benötigt, auch die Lagerung stellt einen essenziellen Baustein dar. Aufgrund der dreidimensionalen Betrachtung und die daraus resultierenden sechs Freiheitsgrade ist eine Vielzahl an Variationen in der Lagergestaltung möglich. Daher ist aus statischer Sicht die Anordnung der verschiedenen Lagertypen von großer Wichtigkeit für ein System und bietet großes Potential für eine Optimierung. Das Anlegen eines Lagers in RFEM lässt sich mit folgendem Code durchführen:

```
// Anlegen einer Liste mit neuen Knotenlagern
  NodalSupport[] nodalSupport = new NodalSupport[nodalSupCount];

// Zuweisung der Freiheitsgrade
  nodalSupport[listIndex].No = nodalSupNo;
  nodalSupport[listIndex].NodeList = nodesupplist;
  nodalSupport[listIndex].RestraintConstantX = -1;
  nodalSupport[listIndex].RestraintConstantY = 0;
  nodalSupport[listIndex].RestraintConstantZ = -1;
  nodalSupport[listIndex].SupportConstantX = -1;
  nodalSupport[listIndex].SupportConstantY = -1;
  nodalSupport[listIndex].SupportConstantZ = -1;
```

14

Code 5.6: RFEM Befehle zum Anlegen einer Knotenlagerung

Der Code 5.6 zeigt den essenziellen Teil der Logik zum Anlegen eines Knotenlagers und seinen Attributen. Dabei handelt es sich um konstante Lagergrößen, wobei RFEM auch in der Lage ist nichtlineare Lagerkomponenten zu berücksichtigen. Für die Prototypentwicklung wurde sich jedoch auf das Anlegen einfacher Lager beschränkt. Über die Zeilen 8 bis 13 werden die Freiheitsgrade des Lagers gesetzt, wobei $RestraintConstant_i$ jeweils die Verdrehbarkeit $\varphi_i$ in $[Nm/rad]$ und $SupportConstant_i$ die Verschieblichkeit $u_i$ in $[N/m]$ beschreibt. Werden diese Attribute zu $-1$ gesetzt, so bedeutet dies, dass diese Freiheitsgrade blockiert sind. Wohingegen die 0 freie Beweglichkeit widerspiegelt. Eine Drehfeder beziehungsweise eine Wegfeder kann simuliert werden, indem ein Wert $> 0$ gewählt wird.
Aufgrund der Fülle an Konfigurationsmöglichkeiten bietet es sich an, die Oberfläche dieses Knotens anzupassen, um dem Anwender Konfigurationen vorzugeben und somit die Fehleranfälligkeit zu minimieren. Dabei könnte jeder Freiheitsgrad zum Beispiel über einen Button aktiviert werden und ihm so ein Wert zugewiesen werden. Von einer Anpassung der Oberfläche dieses Knotens wurde ihm Rahmen der Arbeit aus Zeitgründen verzichtet und lediglich ein einfaches Design gestaltet, das die in Code 5.6 dargestellte Lagersituation anlegt.

Vervollständigt wird das RFEM Modell durch Lasten. Für die Prototypentwicklung wurde sich aufgrund der hohen Vielfalt von Lasten nur auf die Stablast beschränkt, da diese für eine Berechnung ausreichend erscheint. Für das Hinzufügen einer Last in RFEM wird nicht nur die Last selbst benötigt, sondern auch die dazugehörigen Lastfälle. Diese können wiederum zu Last- und Ergebniskombinationen zusammengefügt werden, anhand derer die Berechnungen durchgeführt wird. Für all diese Elemente wurden Dynamoknoten erstellt, die in der Abbildung 5.8 in der Gruppe „Lasten“ begutachtet werden können. Hier ist die hierarchische Lastorganisation in RFEM sowie die Verknüpfung dieser zu erkennen. In diesem *graph* wird die schon zuvor dargestellte Struktur unter Abbildung 5.2 aufgebaut und mittels mehrerer Exportknoten an RFEM übergeben. Dieser Test legte eine zuvor nicht erkannte Problemstellen offen. Die Gestaltung der bisher erstellten Knoten ließ es nicht zu, einen vollständig sequenziellen Ablauf der Knotenstruktur zu realisieren. Verursacht wird dies durch die vorgesehenen Inputs der Knoten und deren automatische Ausführungsreihenfolge durch Dynamo. Somit läuft die Erstellung der Lasten parallel und nicht sequenziell, wie benötigt. Dies führte zur schon zuvor erkannten Problemstellung (siehe Abschnitt 5.4, Seite.42), dass ein paralleler Export nicht umsetzbar ist. Wie ebenfalls in Abschnitt 5.4 benannt, werden Knoten nur erneut ausgeführt, sofern diese eine Änderung erfahren. Betrachtet man mit diesem Wissen die Abbildung 5.8, fallen sofort Design-technische Probleme auf. Der Block zur Erstellung der Lasten läuft parallel zum Rest und besitzt lediglich in seinem letzten Knoten „create_MemberLoad“ eine Verknüpfung zum Ausgangspunkt der Parametrisierung, der durch den Input repräsentiert wird. Resultierend werden die Knoten zum Anlegen der Lastfälle und Lastfallkombinationen nicht erneut ausgeführt, obwohl zuvor diese Elemente gelöscht wurden. Die Stablast benötigt jedoch zwingend einen Lastfall, weshalb der Knoten einen Fehler wirft und die Ausführung fehlschlägt.

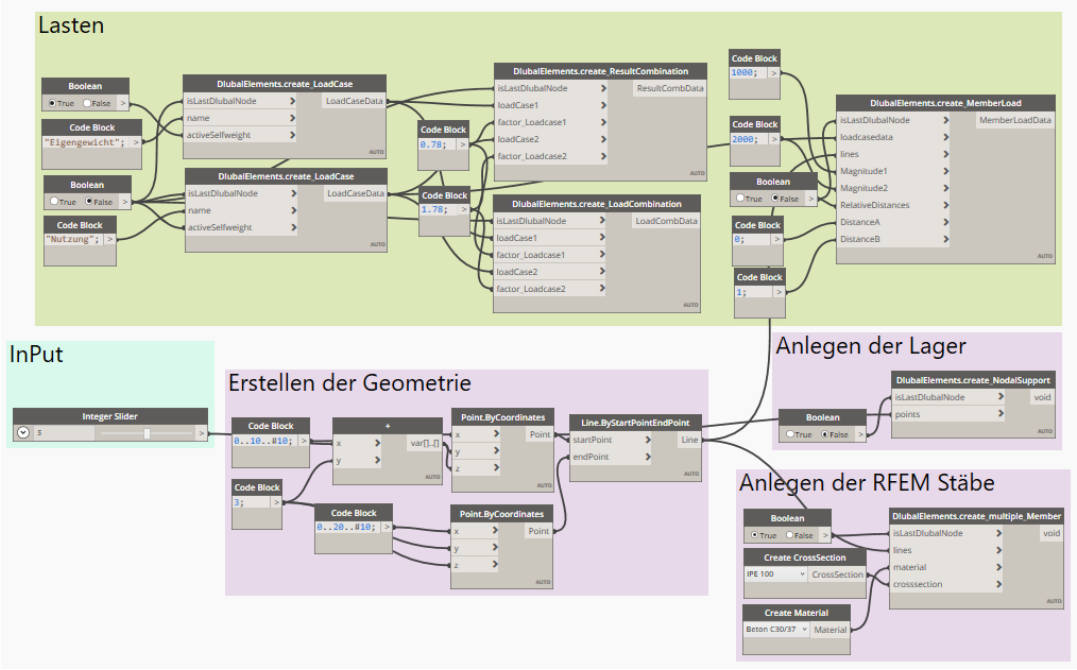


Abbildung 5.8.: Problematisches Design des Exports

Es werden weitere Bestrebungen getätigt den parallelen Ablauf in Reihe zu schalten. Dafür werden neue Klassen erstellt, die als Exportformate dienen. Mittels dieser Exportformate kann die Parallelität des bisherigen *graphs* aufgehoben und das erneute Ausführen der Knoten erzwungen werden. Dennoch tauchen vermehrt neue Schwierigkeiten auf, die viel Zeit fordern. Nach weiteren Tests kann nicht mit 100%iger Sicherheit ermittelt werden, welcher Knoten in Abbildung 5.8 der zuletzt ausgeführte Export ist. Dies variiert mit jedem Neustart von Dynamo.
Alle Bestrebungen in die Richtung eines multiplen Exports stoßen im Laufe des Entwicklungsprozesses auf schwerwiegende Probleme, die mittels eines Wechsels des Export-Konzepts gelöst werden können. Daher wird für den weiteren Verlauf der Entwicklung auf das Export-Konzept 1 nach Abbildung 5.5 gewechselt.

## 5.6. Konzeptwechsel

Mit dem Konzeptwechsel des Exports haben sich große strukturelle Änderungen der Schnittstelle ergeben. Der neue allgemeine Aufbau verfolgt nun das in Abbildung 5.5 dargestellte Konzept 1. Dies besagt, dass der Verbindungsaufbau sowie der dazugehörige Export nach dem Vorbild der Methode unter Code 5.4 Zeile 49-56 nicht mehr in jeden Knoten implementiert wird. Folglich musste die allgemeine Logik der bisherigen Knoten und deren im vorherigen Abschnitt erwähnen Exportformate angepasst werden.
Resultierend bedeutet diese Anpassung, dass alle Knoten, die zuvor Objekte anlegten, nur noch Informationen zur Erstellung dieser Elemente sammeln mussten. Diese Informationen werden im Kontext der Informationstechnik als *Metadaten* bezeichnet. Die Logik der Erstellung sowie des Exports aller Objekte verschoben sich in den Exportknoten. Die vorhandenen Exportformate mussten somit befähigt werden, alle nötigen Metadaten zu speichern und an den Export weiterzuleiten. Diese erhebliche Änderung

wird am nachfolgenden Beispiel des Knotens zur Erstellung von Stäben verdeutlicht.

```
// Neuer Knoten zur Erstellung der Metadaten
public static MemberData Member(Line[] Lines, int MaterialNumber,
  int CrosssecNumber)
{
  return new MemberData(Lines, MaterialNumber, CrosssecNumber);
}
```

Code 5.7: Neuer Code zum Anlegen eines Stabes nach dem Konzeptwechsel

Vergleicht man den Code 5.7 zur vermeintlichen Erstellung eines Stabs mit dem benötigten Code 5.5, werden die Unterschiede deutlich. Der Umfang des in den Knoten enthaltenen Quelltext hat sich aus der Sicht des Entwicklers stark verändert und ist auf ein Minimum geschrumpft. Jegliche Logik befindet sich nun gebündelt im Exportknoten. Die sichtbaren Änderungen für den Anwender dahingegen fallen jedoch sehr gering aus, wie die Abbildung 5.9 verdeutlicht.

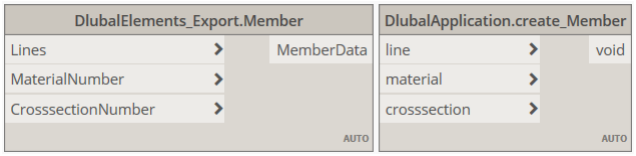


Abbildung 5.9.: Sichtbare Änderung des Konzeptwechsels aus Sicht des Anwenders

Für den Anwender bleibt ein Knoten erhalten, der an sich die gleiche Art von Informationen benötigt. Es müssen Inputs in Form der zu exportieren Linien mitgegeben und Aussagen bezüglich Material und Querschnitt getroffen werden. Letztere haben sich dahingehend verändert, dass nur noch ein *Integer* und nicht mehr ein *String* übergeben wird. Dies bietet zusätzliche Vorteile, die ausführlich in Kapitel 6 betrachtet werden. Die in den vorherigen Kapiteln angesprochenen Probleme bezüglich der Ausführungsreihenfolge sowie der erneuten Ausführung von Knoten konnten mit dem Konzeptwechsel behoben werden. Ersteres löst sich, da nur noch eine Verbindung zu RFEM aufgebaut wird und somit die RFEM-Anfragen nicht mehr kollidieren. Die Lösung für die Problematik mit der wiederholten Ausführung ist im Detail der Knotenausführung zu finden. Wie schon erwähnt, werden die Knoten ohne eine Änderung nicht erneut ausgeführt. Dennoch wird das Ergebnis der letzten Ausführung als Rückgabewert im Knoten gespeichert. Sofern keine Änderung stattfindet, wird dieser Wert bei erneuter Ausführung weitergereicht. Dies begünstigt zusätzlich das neue Exportkonzept, da die benötigten Metadaten dennoch weitergegeben werden. Eine erneute Ausführung wird in diesem Fall nicht benötigt. Der neue Exportknoten wird somit immer ausgeführt, da alle Metadaten mit diesem verknüpft sind und über die Parameter mindestens eine Änderung erfolgt.

## 5.7. Anbindung Zusatzmodul Stahl EC3

Um die Erweiterbarkeit der Schnittstelle zu zeigen, wurde sich dafür entschieden, die erforderlichen Berechnungen mit dem Stahl EC3 Modul von RFEM durchzuführen. Dieses Modul ist nicht in die Kernapplikation von RFEM integriert und muss daher auch

separat an das Visual Studio Projekt angeschlossen werden. Dieser Prozess wurde in Unterabschnitt 4.3.1 erläutert. Dabei sei noch einmal darauf verwiesen, dass auch bei dieser Bibliotheksdatei die Einstellung: *Embed Interop Types = false* zwingend zu setzen ist. Anderenfalls kann dies zu einem schwer deutbaren Fehler führen. Mit der Anbindung, Durchführung und Rückgabe der Berechnung ist der Schritt „Berechnung der Nachweise in RFEM“ des angestrebten Workflows unter Abbildung 3.7 komplett.

```
1
2  // connect to STEEL EC3
3    STEEL_EC3.Module steelModule = model.GetModule("STEEL_EC3");
4
5  // Alle vorherigen Modulfaelle loeschen
6    int caseCount = steelModule.moGetCaseCount();
7    if (caseCount != 0)
8    {
9      for (int i = 1; i <= caseCount; i++)
10     {
11       steelModule.moDeleteCase(i,STEEL_EC3.ITEM_AT.AT_INDEX);
12     }
13   }
14
15 //ModulFall anlegen
16   STEEL_EC3.ICase moCase = steelModule.moSetCase(1, "name");
17   moCase.moSetMemberList(memberlist);
18
19 // Zuweisung der Berechnungsfaelle
20   STEEL_EC3.ULS_LOAD[] loads = new STEEL_EC3.ULS_LOAD[1];
21   loads[0].DesignSituation = STEEL_EC3.DESIGN_SITUATION.DS_FUNDAMENTAL;
22   loads[0].No = 1;
23   loads[0].Type = STEEL_EC3.ILOAD_TYPE.ILOAD_CASE;
24   moCase.moSetULSLoads(loads);
25
26 // Starten und Auslesen der Berechnung
27   moCase.moCalculate();
28   STEEL_EC3.IResults results = moCase.moGetResults();
29   STEEL_EC3.RESULTS_DESIGN[] resultdesign =
30     results.moGetDesignByLoadULSAll();
31
32 // Beenden der Verbindung zu RFEM
33   exitRFEM();
34
35 // Ausgabe der Ergebnisse
36   return resultdesign[resultdesign.Length-1].DesignRatio;
37
```

Code 5.8: Pseudocode zur Anbindung des Stahl EC3 Moduls

Die Anbindung des EC3 Moduls ist ähnlich aufgebaut wie die allgemeine Verbindung zur Kernapplikation RFEM selbst (siehe Code 5.8). Zuerst muss das Modul gefunden werden und als dieses in einer Variable hinterlegt werden. Somit ist man in der Lage mit dem Modul zu kommunizieren und kann einen Berechnungsfall erstellen (siehe Zeile 16). Zur Sicherheit werden alte Berechnungen gelöscht. Nachdem über den Befehl in Zeile 17 die gewünschte Liste der zu berechnenden Stäbe übergeben wurde, müssen die Lastfälle angelegt und übergeben werden (Zeile 20-24). Abschließend werden die Ergebnisse ausgelesen und zurückgegeben.
Nachdem nun alle Voraussetzungen für die Auswertung der Ergebnisse geschaffen wur-

den, konnte eine Weiterverarbeitung in Form einer Untersuchung mit dem Dynamo Modul „Generative Design“ angestrebt werden.

## 5.8. Anbindung des Moduls Generative Design

Die Anbindung des Moduls an den Prozess musste nicht manuell stattfinden, da Generative Design standardmäßig in Dynamo integriert ist. Um die angestrebte Optimierung der gegebenen Struktur durchzuführen, wird deren erzeugender *graph* benötigt. Im *graph* selbst müssen mindestens ein Input sowie Output für die Optimierungsstudie angegeben werden. Nachdem dies erledigt ist, kann eine allgemeine Studie angelegt und anschließend mit entsprechender Konfiguration ausgeführt werden. Die ersten Studien werden an dem bereits bekannten Beispiel der zehn nebeneinander angeordneten Träger durchgeführt.

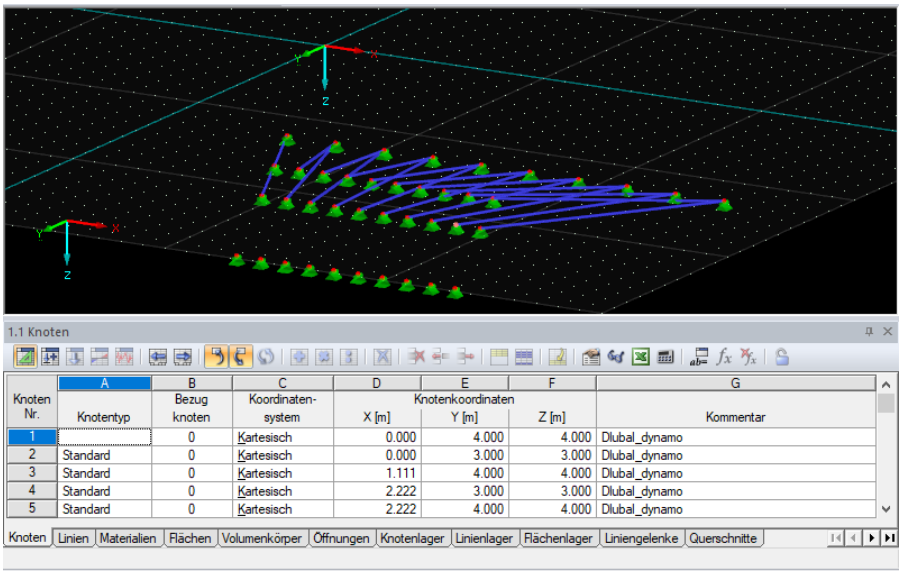


Abbildung 5.10.: Erstes Ergebnis des Generative Design -Moduls und die daran zu erkennende Problematik des Programmablaufs

In der Abbildung 5.10 ist die Benutzeroberfläche von RFEM zu sehen, nachdem die Studie durchgeführt wurde. In das aktive RFEM Modell wurden mehrfach verschiedenste Trägersysteme geschrieben, die durch unterschiedliche Parameterkonfigurationen erstellt wurden. Während der Ausführung konnte visuell erkannt werden, dass der Prozess des Übertragens der Elemente gleichzeitig stattfindet. Resultierend konnten keine Berechnungen durchgeführt werden, da das abgebildete Modell aufgrund der „Verschmelzung“ invalide ist. Bei weiteren Versuchen verursachte die Fallstudie den Absturz von RFEM, was zum totalen Datenverlust führte. Aufgrund der geschilderten Ereignisse wurde vermutet, dass die Anwendung Generative Design auf einen parallelen Prozess aufbaut, was sich im Laufe der Entwicklung bewahrheitete. Der erste Lösungsansatz für diese Problematik befasst sich damit, den Berechnungsablauf parallel beizubehalten und den entsprechenden Zugriff auf die RFEM-Instanz zu reglementieren. Dafür wurde sich den Möglichkeiten des Multithreadings bedient. Über Klassen, wie beispielsweise *Mutex*, wurde versucht den Zugriff auf RFEM zu steuern, was ohne Erfolg endete. Ein weiterer Ansatz beinhaltete das Starten mehrerer RFEM-Instanzen für jeden parallelen Prozess. Das mehrfache Starten von RFEM ist über die unterschiedlichen *Threads* möglich und

kann im Taskmanager des Betriebssystems nachvollzogen werden. Die gestartete Studie in Generative Design kann durchgeführt werden, gibt jedoch keine Ergebnisse zurück. Der Nachteil an mehreren RFEM-Instanzen ist, dass sie im Hintergrund ausgeführt und daher der Anwender keinen Einblick gewinnt, ob Fehler auftreten oder Ergebnisse berechnet werden. Um dies zu umgehen, wurde am Ende der Berechnung das Speichern des Modells angestrebt. Leider führt dies ebenfalls zu keinem Ergebnis.
Somit änderte sich der Ansatz dahingehend, dass nur eine Instanz die parallel laufenden Prozesse verwaltet und ebenfalls alle Berechnungen durchführt. Detaillierte Untersuchungen des Aufbaus von Generative Design ergaben, dass mit der Durchführung einer Studie sechs Instanzen von Dynamo-Core gestartet werden, die parallel den ausgewählten *graph* mit unterschiedlichen Inputparametern starten (siehe Abbildung 5.11).

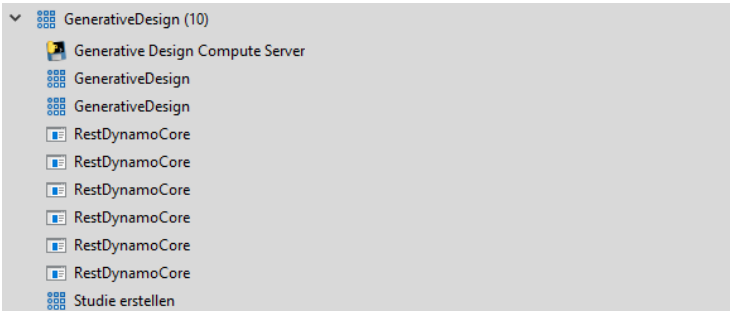


Abbildung 5.11.: Laufende Anwendung nach dem Start von Generative Design im Taskmanager des Windows Betriebssystems

Resultierend bedeutet dies, dass mehrere *Threads* gleichzeitig auf RFEM zugreifen und in das Modell schreiben. Abhängig vom Fortschritt der jeweiligen Berechnung in den Threads, starten unterschiedlichste Befehle, die RFEM nicht ausführen kann, da der erste Thread die Anwendung sperrt, stoppen die Threads nacheinander und RFEM stürzt ab. Die Lösung dieser Problematik konnte durch eine Konfiguration von Generative Design erzielt werden. Es gibt die Möglichkeit, über eine Konfigurationsdatei die Nutzung der verschiedenen DynamoCore-Threads zu steuern. Dafür muss folgende Einstellung vorgenommen werden:

```
// Pfad zur Konfigurationsdatei
  "%AppData%/Roaming/generative-design-client/config.json"

// Inhalt
  {"concurrency":1}
```

Code 5.9: Konfiguration der Parallelisierung von Generativ Design

Über diese Datei kann mittels eines Integers die Standard-Parallelität von vier auf eins beschränkt werden, was dazu führt, dass nur noch eine Instanz von Dynamo-Core verwendet wird und somit der Prozess sequenziell abläuft.
Der Vorteil von parallelen Prozessen ist die Zeitersparnis. Da FEM-Programme jedoch meist alle ungenutzten, freien Ressourcen des Computers für die Berechnung heranziehen können, entsteht kein zeitlicher Gewinn durch den sequenziellen Ablauf. So werden beispielsweise alle freien Ressourcen durch die Anzahl der parallelen Instanzen geteilt

und die Berechnungsdauer steigt mit dem gleichen Faktor an. Daher kann für den Prototyp auf die Nutzung von *Multithreadings* verzichtet werden.

Mit dem erfolgreichen Test des Generativ Designs sind nun alle notwendigen Komponenten verknüpft, die für den angestrebten Workflow benötigt werden. Im nachfolgenden Kapitel wird detaillierter über die Schnittstelle und all ihre Funktionen berichtet und der Workflow anhand eines Beispiels getestet.

# 6. Ergebnisse

## 6.1. Aufbau der Schnittstelle

Als Ergebnis der Entwicklung ist eine Bibliotheksdatei mit dem Namen „DlubalParametrics“ entstanden. Im gleichnamigen Visual Studio Projekt wurden alle nach Unterabschnitt 4.3.1 vorgegebenen Konfigurationen vorgenommen und somit die Voraussetzung zur Verknüpfung der beteiligen Programme geschaffen. Die entstandene .dll-Datei wurde über ein Paket in Dynamo integriert und alle enthaltenen Knoten beziehungsweise Funktionen werden implementiert. Dabei unterliegt der Aufbau des Pakets den in Unterabschnitt 4.3.2 angegebenen Konfigurationen. Der Quellcode der Schnittstelle kann im Detail dem Anhang A entnommen werden. Nachfolgend wird auf die Verwendung und den Aufbau der Schnittstelle in Dynamo eingegangen.

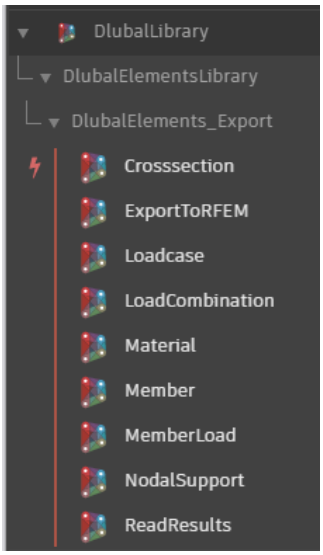


Abbildung 6.1.: Aufbau des Dynamopakets und Gliederung der Knoten in Dynamo

Wie in der oberen Abbildung dargestellt, enthält das DlubalParametrics Paket ausschließlich *Action Nodes*, da für die Übertragung weder das explizite Anlegen von Objekten in Dynamo über die *Constructor Nodes*, noch das Abrufen von Attributen über die *Query Nodes* benötigt wird. Jegliche Funktionalität kann über eine Aktion durchgeführt werden. In der Summe sind sieben Knoten entstanden, die Metadaten erstellen und diese abschließend an den Knoten „ExportToRFEM“ übergeben. Der Knoten „ReadResults“ dient dem Zweck des Auslesens der Ergebnisse. Die erste Gruppe an Knoten, welche nachfolgend genauer betrachtet wird, befasst sich mit der Erstellung strukturbezogener Informationen des Modells.

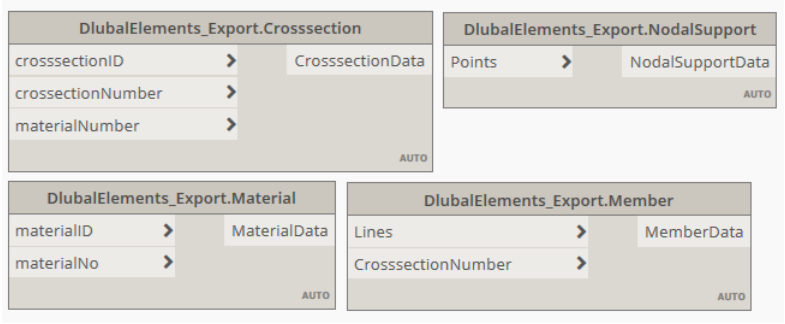


Abbildung 6.2.: Darstellung der Ergebnisse der Dynamo-Knoten: Crosssection, Material, NodalSupport, Member

Für die Erstellung eines statischen Modells in RFEM ist es zuerst notwendig ein Material sowie einen Querschnitt zu erstellen, bevor ein Stab angelegt werden kann. Dafür sind die Knoten *Material* und *Crossection* zuständig (siehe Abbildung 6.2 links). Um ein Material anzulegen, wird eine *materialID* benötigt, die der internen Beschreibung von RFEM angepasst sein muss. Nur mit einer richtigen ID kann ein Abgleich mit der internen Datenbank geschehen. Eine eindeutige Verbesserung dieses Knotens ließe sich über eine Anpassung der Benutzeroberfläche, wie in Unterabschnitt 4.3.4 erwähnt, erzielen. Weiterhin ist jedes Element, welches in RFEM angelegt wird, über den Identifikator „No“ eindeutig beschrieben und die „materialNo“ muss als Input definiert werden. Aufgrund der Abhängigkeit eines Querschnitts zu seinem Material, werden zur Erstellung eines Querschnitts drei Inputs benötigt. Um die Abhängigkeit jedoch so gering wie möglich ausfallen zu lassen, wurde der Input „materialNumber“ in den Knoten eingebaut. Diesem muss eine Integer zugewiesen werden, der auf die „materialNo“ referenziert. Dies bietet den programmtechnischen Vorteil, dass die Klassen entkoppelt sind und zusätzlich Anwendungsmöglichkeiten offengelegt werden. So könnte beispielsweise ein Integer-Parameter eingeführt werden, der das Material oder den Querschnitt über einen Slider anpasst und dieser zur Optimierung herangezogen wird.
Außerdem können in Abbildung 6.2 die Knoten zur Erstellung von Stäben und Knotenlagern betrachtet werden. Der Knoten „NodalSupport“ dient zur Erstellung von Knotenlagern. Dafür werden lediglich die Knoten benötigt, an denen ein Lager vorgesehen ist. Da sich auch hier eine Anpassung der Benutzeroberfläche anbietet, wurde eine Anpassung der Lagersituation nicht mit in den Knoten integriert. Daher ist nur das Standardlager nach Code 5.6 im Prototyp erstellbar. Der Knoten mit dem Namen „Member“ erstellt die Informationen für die Stäbe. Das Design des Knotens im Gegensatz zu Abbildung 5.7 wurde noch einmal vereinfacht und letztendlich werden nur die Linien und ein Querschnitt als Input gefordert. Diese vier beschriebenen Knoten dienen der Erstellung von Elementen, legen dennoch nur Metadaten an.
Die nachfolgenden in Abbildung 6.3 dargestellten drei Dynamoknoten erstellen ebenfalls Metadaten und befassen sich mit der Erstellung von Lasten für das statische Modell. Der Knoten „Loadcase“ stellt die Basis der kompletten Last-Thematik dar, weil diesem die einzelnen Lasten zugeordnet werden müssen. Anschließend besteht die Möglichkeit die Lastfälle zu kombinieren. Ein Lastfall wird über die Inputs *name*, *loadcaseNumber* und *activeSelfweight* erstellt. Dabei können erstere vom Benutzer über eine Zeichenkette und Zahl definiert werden, wohingegen der Input *activeSelfweight* einen booleschen Wert bedarf. Dieser steuert, ob der anzulegende Lastfall das Eigengewicht der Konstruktion

berücksichtigt und somit automatisch mit in die Berechnung einfließt. In Abbildung 6.3 auf der rechten Seite kann der Knoten zur Erstellung einer Kombination von Lastfällen begutachtet werden. Die Funktionsfähigkeit dieses Knotens ist beschränkt auf Kombinationen von zwei Lastfällen. Dabei kann jeweils ein Kombinationsfaktor mit einbezogen werden. Das gezeigte Design ist nicht vollständig ausgereift und sollte in einer Weiterentwicklung neugestaltet werden, da es nicht den vollen Funktionsumfang von Kombinationen bietet.

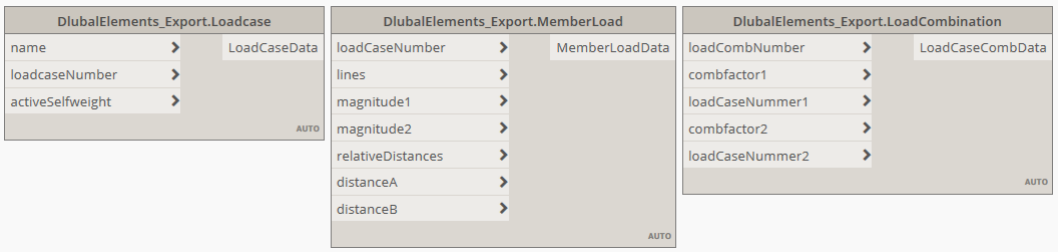


Abbildung 6.3.: Darstellung der Ergebnisse der Dynamo-Knoten: Loadcase, MemberLoad, LoadCombination

Der Knoten „Memberload“ repräsentiert die einzige, konfigurierbare Last in der Bibliothek, was für den Prototyp eine ausreichende Basis darstellt. Dabei ist wichtig, dass die Stablast einem Lastfall über den Input „loadCaseNumber“ zugeordnet wird, da sie alleinstehend nicht in RFEM implementiert werden kann. Über „magnitude1“ sowie „magnitude2“ kann die Größe der Last bestimmt und bei Bedarf eine Trapezlast designet werden. Der boolesche Wert „relativeDistances“ regelt, ob die Länge der Last prozentual zur Stablänge oder in entsprechender Maßeinheit an „distanceA“und „distanceB“ übergeben werden muss. Abschließend sind dem Knoten die zu den Stäben äquivalenten Linien zu übergeben, um einen Stablast anzulegen.

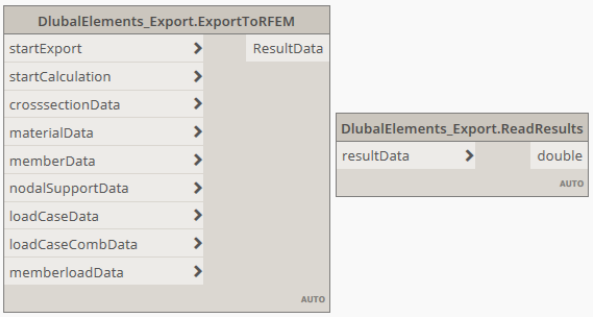


Abbildung 6.4.: Darstellung der Ergebnisse der Dynamo-Knoten: ExportToRFEM, ReadResults

Wurden alle Metadaten über die zuvor gezeigten Knoten anlegt, können die jeweiligen Outputs an den Exportknoten in Abbildung 6.4 übergeben werden. Es sind alle zuvor gezeigten Rückgabetypen als Input des Knotens designet. Diese nehmen nicht nur einzelne Objekte dieser Typen an, sondern ebenfalls Listen mit mehreren Datensätzen. Außerdem wurden die booleschen Inputs „startExport“ sowie „startCalculation“ hinzugefügt, über die der Export und die Berechnung gesteuert werden können. Der Rückgabewert des Exportknotens kann anschließend vom Knoten „ReadResults“ ausgelesen und interpretiert werden. Dabei handelt es sich um eine Logik mit geringer Funktionalität, die

großes Potenzial zur Weiterentwicklung birgt.

## 6.2. Dynamoknoten und Ihre Funktion

In diesem Abschnitt wird der Quellcode und das dazugehörige Design genauer beleuchtet. Dafür ist es wichtig noch einmal das Konzept des Exports unter Abbildung 5.5 mit einzubeziehen. Das gewählte Konzept 1 mit einem einzigen Export-Knoten bestimmt die Umsetzung der Knoten maßgebend, da im selben Schritt die spezifischen Dateitypen in die Schnittstelle integriert wurden. Der Quellcode der Knoten änderte sich dahingehend, dass lediglich die Informationen gesammelt und an den Exportknoten übergeben werden. Dafür wurden spezifische Exportdateitypen erstellt, die im nachfolgenden Code 6.1 schematisch dargestellt sind.

```
namespace ModelDataTyps
{
  // Anlegen eine Exportdatenstruktur
  [IsVisibleInDynamoLibrary(false)]
  public struct MemberData
  {
    public MemberData(Line[] Lines, int CrosssectionNumber)
    {
      this.Lines = Lines;
      this.CrosssectionNumber = CrosssectionNumber;
    }
    public Line[] Lines { get;}
    public int CrosssectionNumber { get;}
  }

  [IsVisibleInDynamoLibrary(false)]
  public struct NodalSupportData
  {
    ...
  }

  [IsVisibleInDynamoLibrary(false)]
  public struct ResultData
  {
    ...
  }

  [IsVisibleInDynamoLibrary(false)]
  public struct MaterialData {
    ...
  }

  [IsVisibleInDynamoLibrary(false)]
  public struct CrosssectionData
  {
    ...
  }

  [IsVisibleInDynamoLibrary(false)]
  public struct LoadCaseData
  {
    ...
  }
```

```
  [IsVisibleInDynamoLibrary(false)]
  public struct LoadCaseCombData
  {
    ...
  }

  [IsVisibleInDynamoLibrary(false)]
  public struct MemberLoadData
  {
    ...
  }
}
```

Code 6.1: Quellcode für Exporttypen

Die speziellen Exportformate wurden als Typ einer *struct*, für Englisch *structure*, angelegt. Eine Struktur kann als Klasse mit geringerer Funktionalität angesehen werden, die für die Aufgabe des Speicherns von Metadaten bestens geeignet ist. Am Beispiel der Struktur *MemberData* ist in Code 6.1 verdeutlicht, dass jeglicher Input der Parameterlist des Konstruktors in einer internen Variable gespeichert wird. Beispielsweise werden für das Anlegen der Metadaten von Stäben Linien und Querschnitte benötigt. Für alle anderen Elemente werden äquivalent zu diesem Beispiel die jeweiligen Inputs in der dafür vorgesehenen Struktur gespeichert. Aufgrund der automatischen Analyse der Bibliotheksdatei durch Dynamo, würden diese Strukturen ebenfalls als Knoten angelegt werden. Dem kann durch die Verwendung des nachfolgenden Befehls entgegengewirkt werden: *[IsVisibleInDynamoLibrary(false)]*. Wie der Name schon verrät, wird der darauffolgende Code nicht durch Dynamo interpretiert. Insgesamt entstehen acht Exportformate, die einen essenziellen Baustein der Schnittstelle darstellen, da der Quellcode der jeweiligen Metadaten-erzeugenden Knoten auf ihnen aufbaut (siehe Code 6.2).

```

  // Neuer Knoten zur Erstellung der Metadaten
  public static NodalSupportData NodalSupport(Point[] Points)
  {
    return new ModelDataTyps.NodalSupportData(Points);
  }

```

Code 6.2: Quellcode zur Erstellung von Metadaten eines Knotenlagers

Die Logik in den Knoten beschränkt sich lediglich auf die Speicherung der Metadaten, wie im Code 6.2 dargestellt wird. Aus der Methode „NodalSupport“ wird durch die verwendete Sichtbarkeit *public* durch Dynamo der gleichnamige Knoten mit „points“ als Input generiert (siehe Abbildung 6.2). Äquivalent zum Code 6.2 ist der Quellcode aller weiteren Knoten gestaltet, die Informationen sammeln. Der Großteil an Funktionalität ist im Exportknoten enthalten, da dieser jegliche Erstellung und Übertragung übernimmt. In Abbildung 6.5 wird die Logik des Exportknotens vereinfacht dargestellt und der Prozess anhand dessen erläutert. Für einen Einblick in den Quellcode des Knotens sei auf den Anhang dieser Arbeit verwiesen.

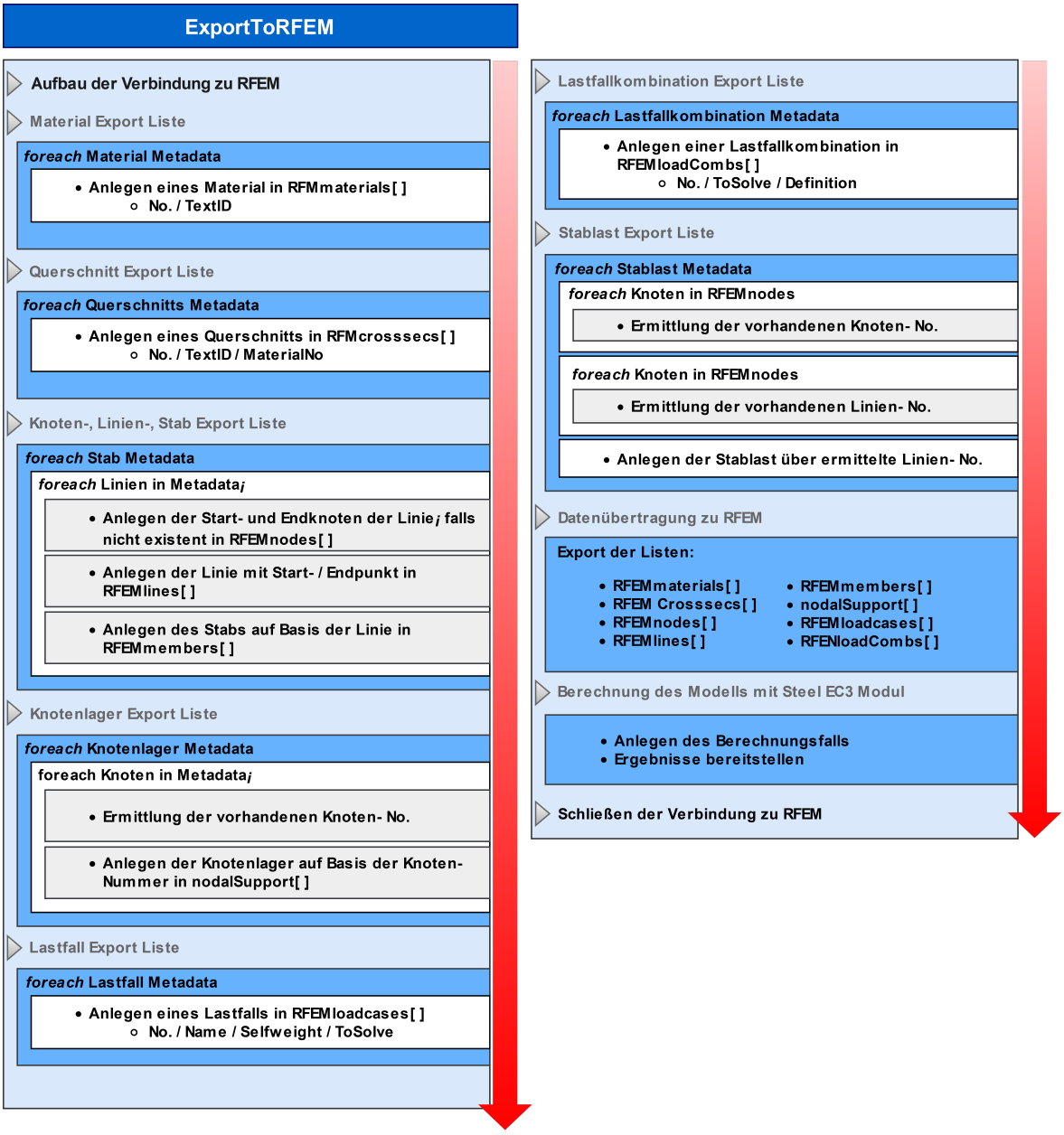


Abbildung 6.5.: Ablaufdiagramm der Logik des Export- Knotens der Schnittstelle

Für den Export der Daten ist jeder Input des Knotens „ExportToRFEM“ zu besetzten. Sind die Inputs „startExport“ und „startCalculation“ mit *true* beantwortet, wird der volle Funktionsumfang, welcher in Abbildung 6.5 dargestellt ist, ausgeführt. Beginnend wird die Verbindung zu RFEM aufgebaut und etabliert. Anschließend werden die Materialien und Querschnitte erstellt, welche die Voraussetzung für das Anlegen von Stäben sind. Im darauffolgenden Schritt werden anhand der Dynamo-Linien die entsprechenden Punkte und Linien erstellt. Die enthaltene Logik achtet dabei auf schon existierende Knoten, um nur die benötigte Anzahl zu übertragen. Aufbauend auf den Linien in RFEM können dann die Stäbe erstellt werden. Nach diesem Schritt ist die komplette Geometrie existent und alle weiteren Aktionen bauen auf dieser auf. So werden die Knotenlager den Modelldaten hinzugefügt, indem die Dynamo-Knoten, an denen ein Lager erstellt wird, mit dem existierenden RFEM-Knoten abgeglichen und deren Identifikator an das Lager übergeben werden. Anschließend werden die Lastfälle angelegt und in einer Lastfallkombination zusammengefügt. Der letzte Schritt, bevor die Berechnung gestartet werden kann, ist das Erstellen der Stablasten. Diese referenzieren auf die angehängten Dynamo-Linien, anhand derer die Stabnummer ermittelt werden kann, welche essenziell für die Last ist. Nachdem über mehrere Schleifen eine Liste der belasteten Stäbe erstellt wurde, kann die Last angelegt werden. Alle erstellten Informationen werden in den in Abbildung 6.5 angegebenen Listen zusammengetragen und anschließend exportiert. Zu diesem Zeitpunkt im Prozess ist das Modell vollständig und die Berechnung über das Stahl EC3 Modul von RFEM kann angelegt werden. Die Berechnung erfolgt automatisch woraufhin die Ergebnisse im erstellten Rückgabeformat abgelegt und bereitgestellt werden.
Somit ist das statische Modell vollständig berechnet und der *graph* ist in der Ausführung am letzten Knoten angekommen. Der Knoten „ReadResults“ kommt nun zum Einsatz, dessen Logik nachfolgend in Code 6.3 abgebildet ist.

```
1
2  public static double ReadResults(ModelDataTyps.ResultData resultData)
3  {
4    double result = resultData.Results[0].DesignRatio;
5    for (int i = 1; i < resultData.Results.Length; i++)
6    {
7      if (resultData.Results[i].DesignRatio > result)
8        result = resultData.Results[i].DesignRatio;
9    }
10   return result;
11 }
12
```

Code 6.3: Quellcode des Auslesens der maximalen Ergebnisse

## 6.3. Optimierung eines Fachwerkträgers

Die rechnergestützte Optimierung stellt einen wichtigen Schritt im praxisnahen Workflow dar. Wie schon im Unterabschnitt 3.3.3 vorgestellt, wird für die Optimierung die Software „Generative Design“ verwendet, welche aufgrund der starken Verknüpfung mit Dynamo einen guten Lösungsansatz darstellt. Die gewählte Optimierungsaufgabe bezieht sich auf die Struktur einer Stahlhalle.

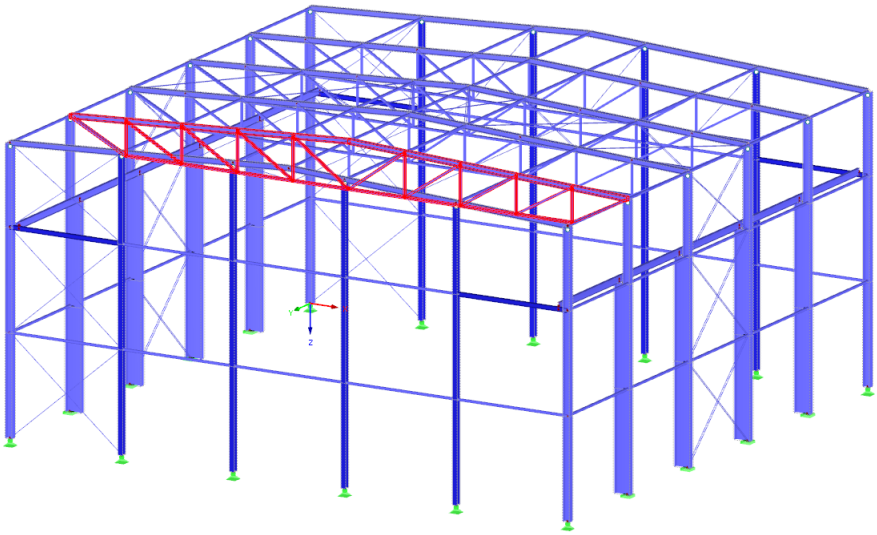

Abbildung 6.6.: Hallenstruktur zur Optimierung des Dachbinders

Zu optimieren ist ein in Fachwerkbauweise ausgeführter Dachbinder einer Halle (siehe Abbildung 6.6). Wie in Kapitel 3 erläutert, lassen sich die nötigen Parameter und zu erreichenden Ziele bestens mit einer initialen Fragestellung oder These herleiten. Für diese Optimierungsaufgabe wurde folgendes behauptet:

„Das Tragwerk ist optimal designet, sofern es maximal genutzt wird."

Diese These lässt Interpretationsspielraum zu, da nicht definiert ist, worauf sich das Maximum bezieht. Dennoch ist wichtig, dass eine maximierte Nutzung angestrebt wird. Für dieses Beispiel wird angenommen, dass eine maximale Nutzung bedeutet, dass eine 100%ige Ausnutzung des Tragwiderstands der Struktur angestrebt wird. Nachdem nun die Optimierungsaufgabe definiert ist, sollten die Randbedingungen sowie die zu definierenden Parameter ermittelt werden.

| **Randbedingungen** | | **Parameter** |
|---|---|---|
| Länge | 29,5[m] | Anzahl Felder |
| Material | Baustahl S 235 | Höhe des Trägers |
| Querschnitte | RRO 180x100x6.3 | |
| | QRO 100x5 | |
| | HEA 240 | |

Tabelle 6.1.: Randbedingungen und Parameter der Optimierungsaufgabe

Tabelle 6.1 enthält die aus der Tragwerksgeometrie resultierenden Randbedingungen. Es wird sich auf die Optimierung der Geometrie beschränkt, da diese ausreichend Optimierungspotential bietet, um den angestrebten Workflow abzubilden. Nachdem die Ausgangssituation definiert ist, kann mit der Umsetzung begonnen werden. Entsprechend dem angestrebten, praxisnahen Workflow in Abbildung 3.7 ist der erste Schritt die Erstellung eines parametrischen Modells. Zu Beginn der Umsetzung ist wichtig die richtige Herangehensweise zu wählen, um die gewünschten Parameter in das Modell einfließen zu lassen. Dafür biete sich an direkt mit der Erstellung der gewünschten Parameter

zu beginnen, um die Geometrie des Modells an ihnen auszurichten. Zusätzlich ist auf den Rückgabetyp der verwendeten Dynamo-Knoten zu achten, da nicht alle Geometrien durch die Knoten der Schnittstelle verarbeitet werden können. So ist es beispielsweise leichter das parametrische Modell mittels einer Polylinie statt einer einfachen Linie zu erzeugen, erstere kann jedoch nicht in entsprechende RFEM-Stäbe übersetzt werden. In

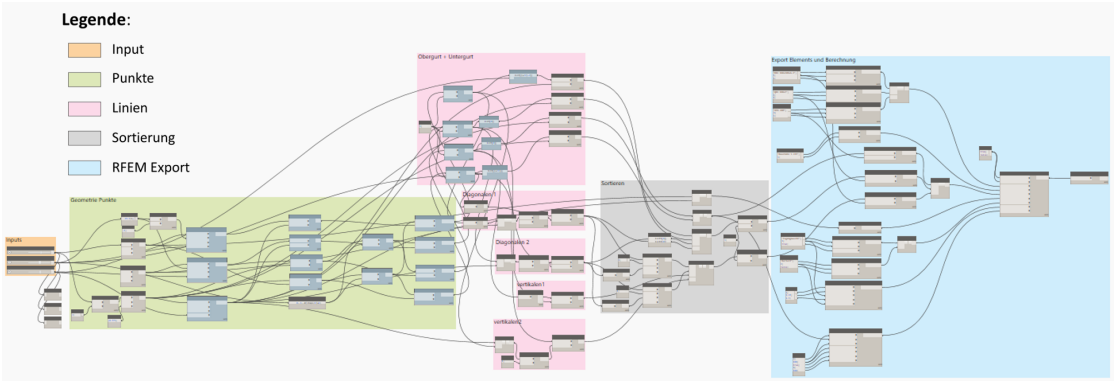


Abbildung 6.7.: Grober Aufbau des *graphs* für die Erstellung des Fachwerkbinders der Optimierungsaufgabe

der Abbildung 6.7 ist der zur Erstellung und Übertragung verwendete *graph* aufgeführt. Die enthaltene Legende benennt die gruppierten Funktionen und liefert einen groben Überblick über die Komplexität der Logik. In Orange ist der Input gestaltet, der nach Fertigstellung des *graphs* die einzige Schnittstelle zum Anwender darstellt. In der nachfolgenden Abbildung 6.8 ist dieser dargestellt. Die in Tabelle 6.1 definierten Parametern ergänzend, wird zusätzlich eine variable Länge des Trägers vorgesehen. Diese wird im Beispiel jedoch auf die vorgegebenen $29,5$[m] festgeschrieben, da die Randbedingung dies vorgibt. Die Länge als Parameter bietet den Vorteil, dass diese Art von Fachwerkträger flexibel an andere Aufgaben angepasst werden kann. Es ist denkbar alle möglichen Parameter für einen Fachwerkträger zu implementieren, um auf viele Konstruktionen vorbereitet zu sein. Der im *graph* in Grün gekennzeichnete Bereich ist für die Erstellung

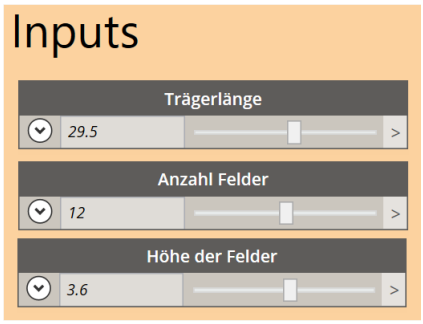


Abbildung 6.8.: Erstellte Inputs des *graphs* zur Erstellung des Fachwerkträgers

der notwendigen Punkte zuständig, anhand derer anschließend alle fachwerkträgerbeschreibenden Linien erstellt werden (pinker Bereich). Mit diesen beiden Abschnitten ist der Träger vollständig in Dynamo erstellt und muss weiterverarbeitet werden. Der in Grau dargestellte Schritt der Weiterverarbeitung ist für das Modell essenziell, da verschiedenste Querschnitte verwendet werden, welche dem Modell nur mittels verschiede-

ner Listen übergeben werden können, in denen sich die entsprechenden Stäbe befinden. Somit werden Listen für den Obergurt, Untergurt sowie für Pfosten und Diagonalen erstellt. Im letzten, blauen Bereich werden dann anhand der Listen die Metadaten der Elemente erstellt und an den Exportknoten übergeben. Mit der Ausführung des Knotennetzes wird die Struktur zu RFEM übertragen und berechnet (siehe Abbildung 6.9). Aufbauend auf dieser Berechnung kann dann die Optimierung über Generative Design

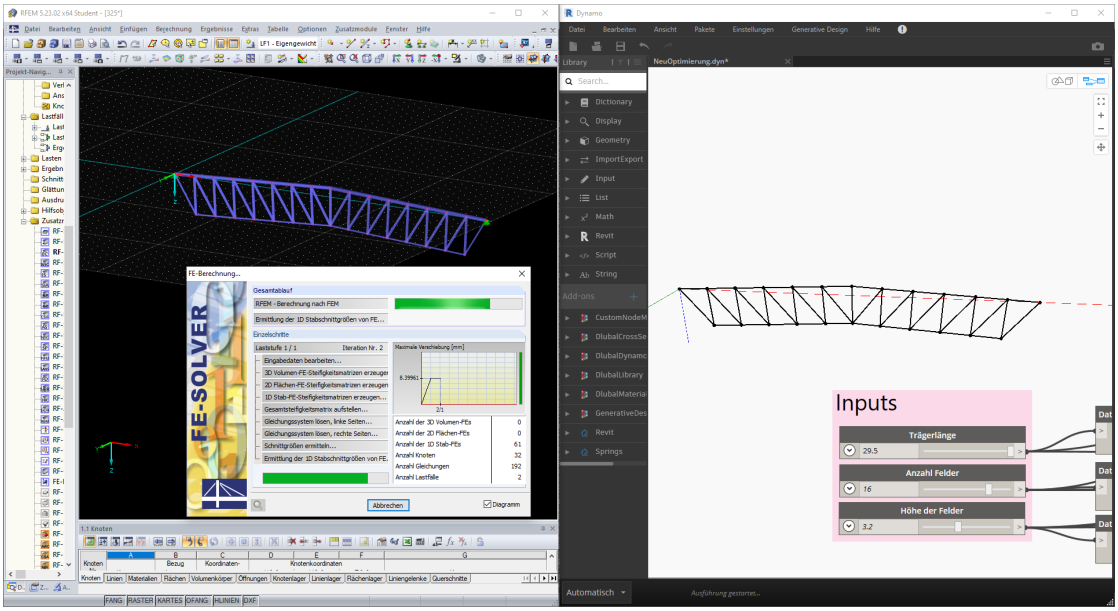


Abbildung 6.9.: Ausführung des *graphs* und das parallele Ausführen der Berechnung

durchgeführt werden. Dafür ist es notwendig, die zu optimierenden Inputs auch als solche zu kennzeichnen. Dies kann über einen Rechtsklick auf den entsprechenden Knoten durchgeführt werden, was ebenfalls für die Deklaration eines Outputs gilt. Daraufhin kann eine Studie erstellt werden, über die der *graph* untersucht werden kann. Die Erstellung einer Studie bewirkt, dass die .dyn-Datei des *graphs* im Kontext von Generative Design lokal gespeichert wird, sofern alle Voraussetzungen erfüllt sind. Über die Funktion „Studie erstellen...“ kann eine solche angelegt werden. Die dazugehörige Konfiguration ist in der nachfolgenden Abbildung 6.10 dargestellt. Als Parameter der Untersuchung werden die Größen „Höhe der Felder“ und „Anzahl der Felder“ ausgewählt, womit es sich um eine Optimierung von mehreren Parametern handelt. Im Bereich „Ziele festlegen“ wird dann entschieden, ob es sich um eine Minima- oder Maxima-Optimierung handelt, deren Zielintervall der Untersuchung über die Felder „Min.“ und „Max.“ festgelegt werden kann. Der Bereich „Einstellungen der Generierung“ bestimmt dann, wie komplex die Untersuchung durchgeführt wird. Die Anzahl „Seed“ beschreibt mit wie vielen Ausgangskonfigurationen gerechnet wird. Die „Generation“ dahingegen definiert wie viele iterative Berechnungen der „Populationsgröße“ durchgeführt werden, womit sich eine Anzahl durchzuführenden Berechnungen nach folgender Formel ergibt:

$$\text{Anzahl der Berechnungen} = \text{Seeds} * \text{Generationen} * \text{Populationsgröße} \tag{6.1}$$

Nach Formel (6.1) steigt der Aufwand der Berechnung schnell an, da es sich hier um Faktoren handelt. Somit sollte für die ersten Tests aufmerksam mit der Anzahl umgegangen werden, um keine zu großen Studien zu starten.

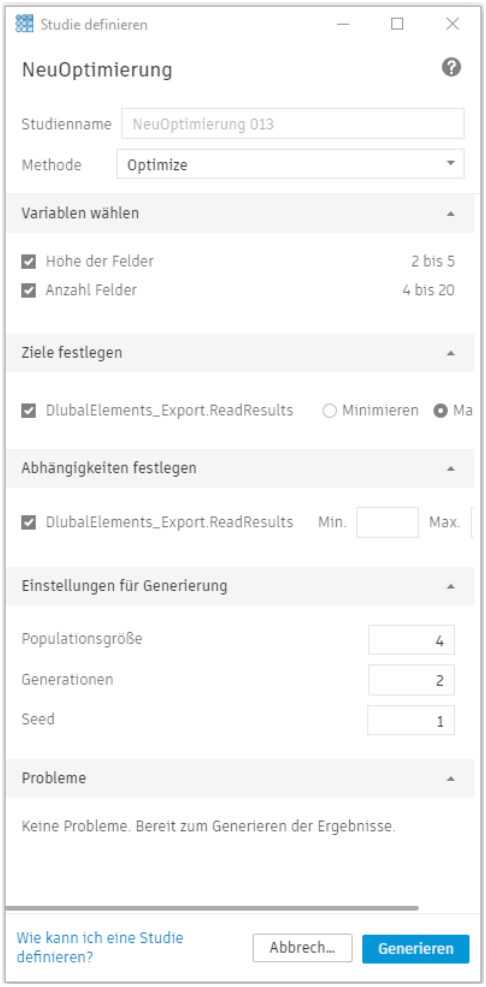


Abbildung 6.10.: Ergebnis der Optimierung

In Abbildung 6.11 ist das Ergebnis der zuvor beschriebenen Studie dargestellt. Leider werden nicht alle berechneten Modelle gespeichert und mit in den Ergebnispool aufgenommen, was standardmäßig nicht auftreten sollte. Somit ist noch ein Fehler in der Anwendung des Optimierungs-Moduls von Generative Design enthalten, der durch weitere Untersuchungen auf die Konfiguration von Generative Design selbst beziehungsweise auf den Aufbau des *graphs* begrenzt werden konnte. Weitere Untersuchungen werden diese Problematik beheben und eine vollständige Anwendung ermöglichen. Nichtsdestotrotz wurde das Maximum der Auslastung des Fachwerkträgers ermittelt.
Es sei jedoch darauf hingewiesen, dass die erzielten Ergebnisse nicht weiter bezüglich der in RFEM erstellten Korrektheit der Knicklängen der Stäbe untersucht wurden. Somit können nicht reale Auslastungen entstanden sein, die keine treffende Aussagekraft bezogen auf ihre Größe besitzen. In Bezug auf die Prototypentwicklung spielt dies jedoch eine untergeordnete Rolle, da die Anwendbarkeit und Funktionsfähigkeit im Fokus stehen. Für eine aussagekräftige Optimierung dieser Struktur müssen weitere Konfigurationen vorgenommen werden. Nach Abbildung 6.11 ist die maximale Auslastung auf eine ca. 60%ige Auslastung der Tragfähigkeit beschränkt, da die Randbedingungen dies bedingen.

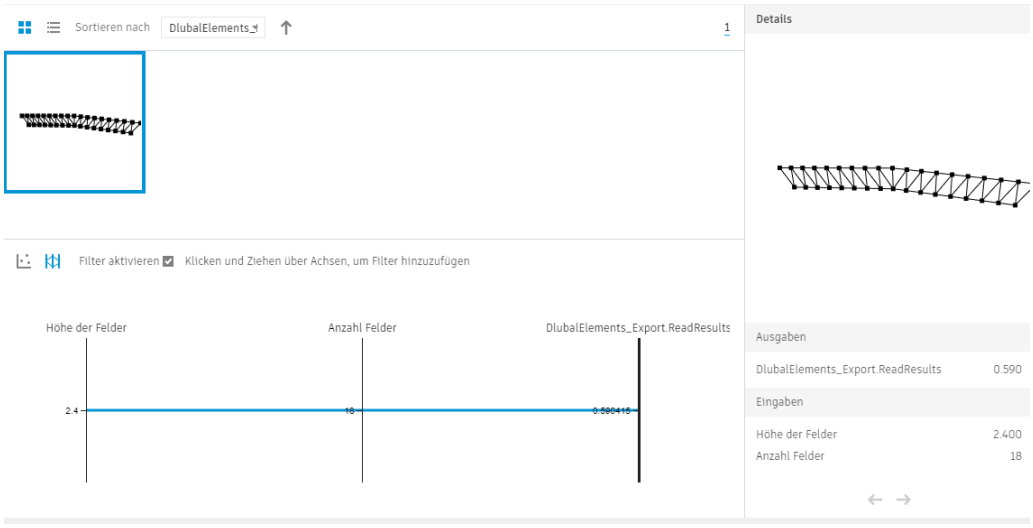


Abbildung 6.11.: Ergebnis der Optimierung des Fachwerkträgers

Den Beweis dafür, dass Generative Design das richtige Werkzeug für Untersuchungen dieser Form ist, kann über eine Fallstudie erbracht werden. Da die ausgeführte Logik hinter einer Fallstudie (Randomize, siehe Abbildung 3.10) eine andere ist, können die Ergebnisse entsprechend ermittelt und zurückgegeben werden. Die Abbildung 6.12 zeigt das

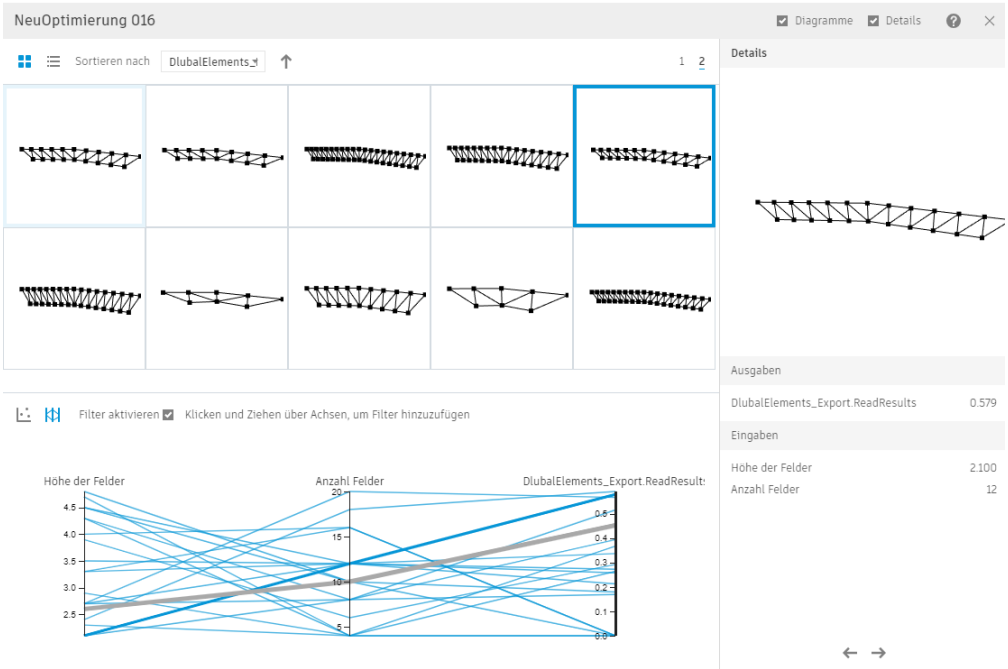


Abbildung 6.12.: Ergebnis der Fallstudie des Fachwerkträgers

Ergebnis einer Fallstudie, in der 20 verschiedene Konfigurationen des Fachwerkträgers ermittelt wurden. Auf die Korrektheit der Ergebnisse durch fehlende Berechnungskonfigurationen wurde verzichtet, da auch hier noch keine vollständige Funktionsfähigkeit erreicht wurde. Zu erkennen ist dies anhand der ermittelten Auslastung von $\eta = 0.0$, was allein aufgrund der Belastung durch das Eigengewicht nicht möglich ist. Somit werden

auch bei diesen Betrachtungen Ergebnisse nicht richtig an Generative Design übermittelt, was wiederum auf einen Fehler im Design des *graphs* hinweist. Dies konnte im Rahmen der Arbeit jedoch noch nicht bewiesen werden und weitere Untersuchungen sind notwendig.

## 6.4. Anlegen von Modelldaten in Revit

Der letzte Schritt des praxisnahen Workflows unter Abbildung 3.7 ist der Import der Struktur durch Revit. Dafür wird schon ein in Dynamo implementiertes Paket bereitgestellt, dass alle notwendigen Knoten enthält. In der Abbildung 6.13 ist der zur Über-

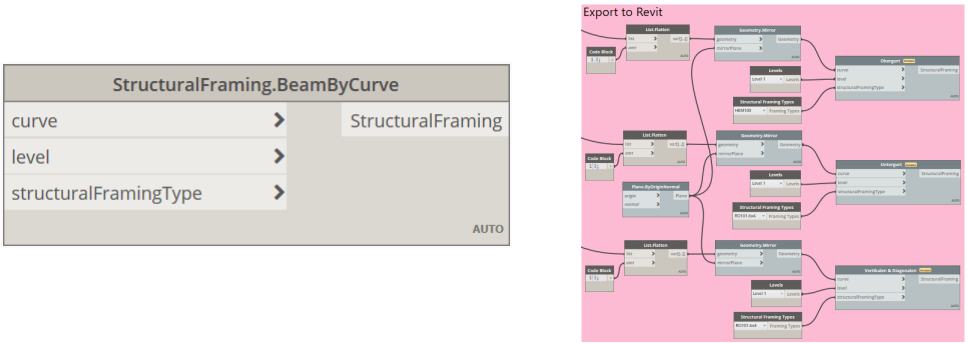


Abbildung 6.13.: BeamByCurve- Knoten und Knotennetz zum Export des Fachwerkträgers zu Revit

tragung benötigte Knoten und das dazugehörige Knotennetz als Übersicht abgebildet. Mittels „BeamByCurve“ können dem Input „curve“ zuvor in Dynamo erstellte Linien übergeben werden, die im angegebenen „level“ angelegt werden und über „structuralFramingType“ ihren benötigten Querschnitt verliehen bekommen. Die in Pink dargestellte Gruppe an Knoten gibt eine Übersicht über die Anzahl der verwendeten Knoten. Anhand dieser ist zu erkennen, dass die Übertragung der Struktur nur geringer Aufwand bedeutet, da die Geometrie aufgrund von variierenden Koordinatensystemen noch mittel vier Knoten gedreht werden muss. Außerdem ist zu beachten, dass die verwendeten Querschnitte zunächst in das Revit-Projekt als *Familie* importiert werden müssen, um in Dynamo zugeordnet werden zu können. Das Ergebnis der Übertragung kann in Abbildung 6.14 betrachtet werden. Weiterhin ist darauf zu achten, dass die gewählten Maß-

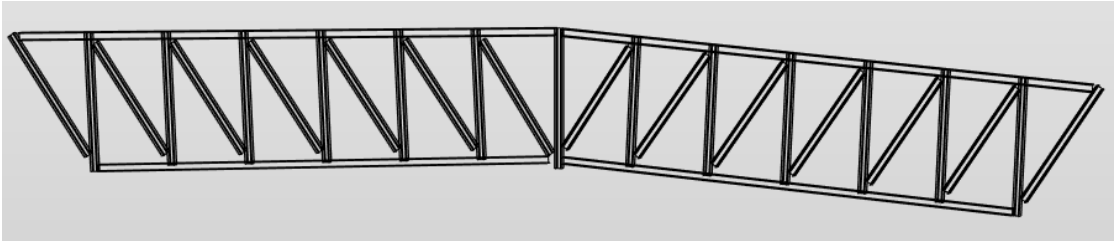

Abbildung 6.14.: Ergebnis des exportierten Fachwerkträgers in Revit

einheiten übereinstimmen, da eine Abweichung dazu führt, dass die Struktur fehlerhaft übergeben wird. Mit der Übertragung des Fachwerkträgers zu RFEM ist der praxisnahe

Workflow abgeschlossen und dessen Funktionsfähigkeit über die erstellte Schnittstelle als Prototyp bewiesen.

## 6.5. Ergebnisbewertung

Zusammenfassend ist mittels der Entwicklung der Schnittstelle ein Prototyp entstanden, der den Prozess der Tragwerksplanung in den BIM-Prozess integriert. Es können einfache, parametrische, statische Modelle in Dynamo for Revit 2.5 erstellt werden und an die FEM-Software RFEM 5.23 übergeben werden. Dazu wurden neun Dynamo-Knoten entwickelt, die über eine Bibliotheksdatei(.dll) in Dynamo eingebunden werden können. Der Anwender der Schnittstelle wird in die Lage versetzt, die folgenden Elemente eines Modells in RFEM zu erstellen: Materialien, Querschnitte, Stäbe, Knotenlager, Lastfälle, Lastfallkombinationen und Stablasten. Weiterhin kann eine statische Berechnung über das RFEM-Modul *RF-STAHL EC3 - Bemessung nach Eurocode 3* durchgeführt und anschließend die maximale Auslastung ausgelesen werden.
Der Umfang der Knoten ist ausreichend oder genügt einer ersten Betrachtung. Bei intensiver Verwendung wird dennoch recht schnell deutlich, dass die Möglichkeiten beschränkt sind und weitere Funktionalität notwendig ist. Mit dieser Entwicklung kann jedoch aufgezeigt werden, dass sich hinter dieser Methodik ein großes Potential verbirgt, welches die alltägliche Arbeit im Bereich der Statik nachhaltig ändern könnte.
Zusätzlich wurde das Modul Generative Design von Revit 2021 mit in die Betrachtung integriert, um einen weiteren Mehrwert zu schaffen. Das Modul ist in der Lage, die verknüpfte Logik zur Erstellung des Modells weiteren Betrachtungen zu unterziehen. Somit können Fallstudien und Optimierungen in kürzester Zeit durchgeführt werden, wobei verschiedenste Strukturvarianten untersucht und hinsichtlich definierter Parameter bewertet werden. Resultierend steigt die Qualität und Effizienz der Arbeit im Bereich der Statik.
Der Ablauf der Prozesse in Generative Design ist mit der erarbeiteten Konfiguration dieser Arbeit noch nicht bei einer 100%igen Funktionsfähigkeit angelangt, da noch kleinere Fehler enthalten sind. Diese sollten jedoch in einer Weiterentwicklung behoben werden können. Nichtsdestotrotz stellt Generative Design für diesen Prozess einen erheblichen Mehrwert dar, auf den in Zukunft ein großes Augenmerk der Entwicklung gelegt werden sollte.
Außerdem wird abschließend die Möglichkeit geboten, über die in Dynamo integrierten Funktionen das statische Modell an Revit zu übergeben. Wie in Abbildung 6.13 gezeigt, ist die Übergabe von Stäben recht simpel umzusetzen, dennoch lassen die Detailpunkte wie die Anschlüsse noch Raum für Verbesserungen zu. Mittels der Möglichkeit Stahl Verbindungen zu erstellen, sollte diese Umsetzung durch Revit realisierbar sein.
Abschließend zeigt diese Entwicklung, dass die Digitalisierung auch im Bereich der Statik eine gute Anbindung in den BIM-Prozess liefern kann. Nun gilt es dieses Potential allgemein zu erschließen, um somit den Bereich der Planung nachhaltig und positiv zu beeinflussen.

# 7. Ausblick

Mit dieser Arbeit wurde eine erste Herangehensweise an die Thematik der parametrischen, statischen Dimensionierung und dessen Anbindung an den allgemeinen BIM-Prozess erarbeitet. Mit vermutlich steigender Anzahl an BIM-Projekten in der Bauwirtschaft werden sich neue Wege erschließen, um Arbeitsaufgaben zu verbessern und zu optimieren. Der entwickelte Prototyp zeig auf, wohin eine Entwicklung streben kann und welches Potential sie birgt.
Angelehnt an den zuvor erarbeiteten Prozess (siehe Abbildung 7.1), ist die Schnittstelle in der Lage, die Software Dynamo for Revit 2.5 und RFEM 5.23 zu verbinden. Die Arbeit aus Sicht der Tragwerksplanung ändert sich dahingehend, dass die Modellerstellung von RFEM zu Dynamo verlagert wird und ein parametrisches Modell entsteht.

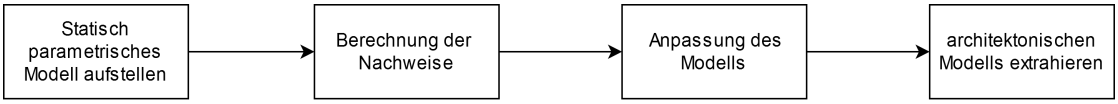


Abbildung 7.1.: Umgesetzter, praxisnaher Workflow zur statisch parametrischen Modellierung

Eine Weiterentwicklung der Schnittstelle ist in der Automatisierung zu finden. Der unterstütze Workflow basiert auf der manuellen Erstellung eines parametrischen Modells in Dynamo. Dieser Prozessschritt kann noch nicht automatisch durch den Prototyp ausgeführt werden, da der essenzielle Schritt zum Ableiten des statischen Modells aus der architektonische Vorlage nicht implementiert ist.
Die Entwicklung eines intelligenten Algorithmus, der aus einem architektonischen Modell das statische extrahiert, wird für den Bereich der Statik eine große bleibende Veränderung darstellen. Für die Umsetzung einer solchen Logik muss jedoch anfangs ein umfängliches Entwicklungskonzept entwickelt werden, dass zahlreiche Nachforschungen bedarf. Es müssen Randbedingungen und Standards erarbeitet werden, um eine Datengrundlage zu schaffen. Weiterhin muss der komplexe Ablauf für die Integration von Parametern in ein bestehendes Modell untersucht werden. In erster Linie muss erarbeitet werden, in welchem Rahmen dieser Vorgang mit aktueller Technologie umsetzbar ist.
Weiterhin ist denkbar eine künstliche Intelligenz (KI) in den Prozess mit einzubeziehen. Über eine KI können statische Modelle bezogen auf eine existierende Vorlage generiert und anschließend die passenden Rechenmodelle hinterlegt werden. Für diese Weiterentwicklung ist ebenfalls die Thematik der Integration von Parametern in das Modell von hohem Interesse.
Die erste Weiterentwicklung der Schnittstelle sollte sich mit den zahlreich in der Arbeit erwähnten Verbesserungen beschäftigen. Daraufhin sollte der Funktionsumfang der Schnittstelle auf Flächen- und Schalentragwerke erweitert werden, um in der Lage zu sein, alle Systeme betrachten zu können.
Mit Blick auf die ständig fortschreitende Digitalisierung im Bauwesen, kann auch dieser Prototyp ein Schritt in eine zukunftsweisende Richtung sein.

# Literaturverzeichnis


[1] *Forschungsinitiative Zukunft Bau*. Bd. 2756,2: *Planungsleitfaden Zukunft Industriebau: Ganzheitliche Integration und Optimierung des Planungs- und Realisierungsprozess für zukunftsweisende und nachhaltige Industriegebäude*. Stuttgart : Fraunhofer-IRB-Verl., 2011. – ISBN 978-3-8167-8517-0

[2] *Pro C# 7: With .NET and .NET Core*. 8th edition. Berkeley, CA : Apress, 2017. – ISBN 978-1-4842-3018-3

[3] 1000LOGOS.NET: *Visual Studio Logo*. – URL `https://1000logos.net/visual-studio-logo/`

[4] ALVARO ORTEGA PICKMANS: *Dynamo Development - London Hackathon 2019*. – URL `https://alvpickmans.github.io/DynamoDevelopment-London-Hackathon-2019/`. – Zugriffsdatum: 01.12.2020

[5] AUTODESK: *The Dynamo Primer For Dynamo v2.0: a comprehensive guide to visual programming in Autodesk Dynamo Studio*. 2020. – URL `https://primer.dynamobim.org/`. – Zugriffsdatum: 18.09.2020

[6] AUTODESK GENERATIVE DESIGN: *Generative Design Primer*. 2020. – URL `https://www.generativedesign.org`. – Zugriffsdatum: 03.11.2020

[7] AUTODESK REVIT: *Revit 2021*. 2020. – URL `https://www.autodesk.de`. – Zugriffsdatum: 03.11.2020

[8] BORRMANN, André ; KÖNIG: *Building Information Modeling*. Springer Fachmedien Wiesbaden, 2015. – ISBN 978-3-658-05605-6

[9] DLUBAL GMBH: *Website Dlubal GmbH*. 2020. – URL `https://www.dlubal.com/de`. – Zugriffsdatum: 25.10.2020

[10] DR. JAN LINXWEILER: *Modellierung und Simulation I: Vorlesung 2*. 2018

[11] ECMA INTERNATIONAL: *C# Language Specification: ECMA-334 5th Edition / December 2017: C# Language*. – URL `https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-334.pdf`. – Zugriffsdatum: 25.11.2020

[12] KAY, Alan C.: The early history of Smalltalk. In: *ACM SIGPLAN Notices* 28 (1993), Nr. 3, S. 69–95. – ISSN 03621340

[13] LAHRES, Bernhard ; RAÝMAN, Gregor: *Objektorientierte Programmierung: Das umfassende Handbuch ; [objektorientierte Programmierung verständlich erklärt ; von den Prinzipien über den Entwurf bis zur Umsetzung ; Praxisbeispiele in UML, Java, C#, C++, JavaScript, Ruby, Python und PHP*. 2., aktualisierte u. erw. Aufl., korr. Nachdr. Bonn : Galileo Press, 2011 (Galileo computing). – ISBN 978-3-8362-1401-8

[14] LEVEL.HTML https://www.ergotec.de/de/safety: *Abbildung Fahrräder*. – URL `https://www.ergotec.de/de/safety-level.html`

[15] MICROSOFT: *Was ist eine dll?*. – URL `https://docs.microsoft.com/de-de/troubleshoot/windows-client/deployment/dynamic-link-library`. – Zugriffsdatum: 25.11.2020

[16] MICROSOFT: *.Net Framework*. 2020. – URL `https://visualstudio.microsoft.com/de/vs/community/`. – Zugriffsdatum: 25.10.2020

[17] MICROSOFT: *Visual Studio Community*. 2020. – URL `https://visualstudio.microsoft.com/de/vs/community/`. – Zugriffsdatum: 25.10.2020

[18] MODE LAB, Autodesk: *Dynamo Developer Resources*. – URL `https://developer.dynamobim.org/`. – Zugriffsdatum: 01.12.2020

[19] RACEL WILLIAMS, RADU GIDEI, THOMAS MAHON: *Autodesk University - Dynamo for Software Developers: C#, Python, and More - Workshop*. 2019. – URL `https://www.autodesk.com/autodesk-university/class/Dynamo-Software-Developers-C-Python-and-More-Workshop-Part-1-4-2019`

[20] STANGE, Matthias: *Building Information Modelling im Planungs- und Bauprozess: Eine quantitative Analyse aus planungsökonomischer Perspektive*. 1. Auflage 2020. Wiesbaden : Springer Fachmedien Wiesbaden GmbH and Springer Vieweg, 2020. – ISBN 978-3-658-29837-1

# A. Anhang



## Erklärung der Projektdateien

Die mit dieser Arbeit mitgelieferten Dateien beinhalten das entwickelte Projekt in Visual Studio Community 2019. Über die Projektmappe namens „DlubalSolution“ können die folgenden drei Projekte geöffnet werden:

- DlubalCrossSectionLibrary - Anpassung Benutzeroberfläche Querschnitts-Knoten
- DlubalMaterialLibrary - Anpassung Benutzeroberfläche Material-Knoten
- DlubalParametrics - Schnittstelle des praxisnahen Workflows

Die Projektdaten können unter folgendem Link heruntergeladen werden:

https://1drv.ms/u/s!AktQTykv6Uopg9EgTv9IkJBVvTouyg?e=AsIcRX

```
using System.Runtime.InteropServices;
using Dlubal.RFEM5;
using Autodesk.DesignScript.Geometry;
using Autodesk.DesignScript.Interfaces;
using Autodesk.DesignScript.Runtime;
using Dynamo.Graph.Nodes;
using CoreNodeModels;
using System.Collections.Generic;
using ProtoCore.AST.AssociativeAST;
using Dynamo.Utilities;
using System;
using System.Linq;
using System.Windows.Media.Animation;
using System.Data;
using DesignScript.Builtin;
using System.Windows.Markup;
using System.Security.Permissions;
using NUnit.Framework.Constraints;
using System.Linq.Expressions;
using Line = Autodesk.DesignScript.Geometry.Line;
using Microsoft.Practices.Prism.Interactivity.InteractionRequest;
using System.Threading;
using System.IO;   // für threadtest
using System.Text;
using System.Windows;

namespace DlubalElementsLibrary
{
    public class DlubalElements_Export
    {

        // private Constructor to avoid showing up in Dynamo
        private DlubalElements_Export() { }

        // Create Metadata for LoadCase
        public static ModelDataTyps.LoadCaseData Loadcase(string name, int loadcaseNumber, bool
activeSelfweight)
        {
            return new ModelDataTyps.LoadCaseData( name, loadcaseNumber, activeSelfweight);
        }

        // Create Metadata for Member
        public static ModelDataTyps.MemberData Member(Line[] Lines, int CrosssectionNumber)
        {
            return new ModelDataTyps.MemberData(Lines, CrosssectionNumber);
        }

        // Create Metadata for NodalSupport
        public static ModelDataTyps.NodalSupportData
NodalSupport(Autodesk.DesignScript.Geometry.Point[] Points)
        {
            return new ModelDataTyps.NodalSupportData(Points);
        }

        // Create Metadata for Material
        public static ModelDataTyps.MaterialData Material(string materialID, int materialNo)
        {
              return new ModelDataTyps.MaterialData(materialID, materialNo);
        }
```

```
        // Create Metadata for Crosssection
        public static ModelDataTyps.CrosssectionData Crosssection(string crosssectionID, int
crossectionNumber, int materialNumber)
        {
            return new ModelDataTyps.CrosssectionData(crosssectionID, materialNumber,
crossectionNumber);
        }

        // Create Metadata for MemberLoad
        public static ModelDataTyps.MemberLoadData MemberLoad(int loadCaseNumber, Line[] lines,
double magnitude1, double magnitude2, bool relativeDistances, double distanceA, double distanceB)
        {
            return new ModelDataTyps.MemberLoadData(loadCaseNumber, lines, magnitude1,
magnitude2, relativeDistances, distanceA, distanceB);
        }

        // Create Metadata for LoadCombination
        public static ModelDataTyps.LoadCaseCombData LoadCombination(int loadCombNumber, double
combfactor1, int loadCaseNummer1, double combfactor2, int loadCaseNummer2)
        {
            return new ModelDataTyps.LoadCaseCombData(loadCombNumber, combfactor1,
loadCaseNummer1, combfactor2, loadCaseNummer2);
        }

        // Read ResultData from ExportNode and return of highest result
        public static double ReadResults(ModelDataTyps.ResultData resultData)
        {
            double result = resultData.Results[0].DesignRatio;

            for (int i = 1; i < resultData.Results.Length; i++)
            {
                if (resultData.Results[i].DesignRatio > result)
                    result = resultData.Results[i].DesignRatio;
            }

            return result;
        }

        // Create all elements, Export to RFEM, Start and read calculation
        public static ModelDataTyps.ResultData ExportToRFEM(bool startExport = false, bool
startCalculation = false,
            ModelDataTyps.CrosssectionData[] crosssectionData = null, ModelDataTyps.MaterialData[]
materialData = null,
            ModelDataTyps.MemberData[] memberData = null, ModelDataTyps.NodalSupportData[]
nodalSupportData = null,
            ModelDataTyps.LoadCaseData[] loadCaseData = null, ModelDataTyps.LoadCaseCombData[]
loadCaseCombData = null,
            ModelDataTyps.MemberLoadData[] memberloadData = null)
        {
            try
            {
                /////////////// initialisation ////////////////////////////////////////////////////

                // declare Variable
                int matNo = 1;
                int nodeNo = 1;
                int lineNo = 1;
                int memberNo = 1;
                bool isStartpointExisting = false;
                int existingStartPointNo = 0;
                bool isEndpointExisting = false;
```

```
int existingEndPointNo = 0;
int nodeListIndex = 0;
int listIndex = 0;

// establish RFEM connection
IApplication app = Marshal.GetActiveObject("RFEM5.Application") as IApplication;
app.LockLicense();
IModel model = app.GetActiveModel();
IModelData modelData = model.GetModelData();
ILoads modelLoads = model.GetLoads();

// delete ModelData
modelData.PrepareModification();
modelData.Clean();
modelData.FinishModification();

/////////////// create Material List ////////////////////////////////////////////////////

if (materialData == null)
{
    throw new Exception("missing MaterialData, insert it to solve problem");
}
// create Export-list
int matCount = materialData.Count();
Material[] RFMmaterials = new Material[matCount];

for (int i = 0; i < matCount; i++)
{
    RFMmaterials[i].TextID = materialData[i].MaterialID;
    RFMmaterials[i].No = matNo;
    matNo++;
}

/////////////// create Crosssection List ////////////////////////////////////////////////

if (crosssectionData == null)
{
    throw new Exception("missing crosssectionData, insert it to solve problem");
}
// create Export-list
int crssCount = crosssectionData.Count();
CrossSection[] RFMcrosssecs = new CrossSection[crssCount];

for (int i = 0; i < crssCount; i++)
{
    RFMcrosssecs[i].TextID = crosssectionData[i].CrosssectionID;
    RFMcrosssecs[i].MaterialNo = crosssectionData[i].MaterialNo;
    RFMcrosssecs[i].No = crosssectionData[i].CrosssectionNo;
}

/////////////// create Member List ////////////////////////////////////////////////////

// create output RFEM lists
int nodeCount = 0;
int lineCount = 0;
int memberCount = 0;

if (memberData == null)
{
    throw new Exception("missing memberData, insert it to solve problem");
}
```

```
            foreach (ModelDataTyps.MemberData memD in memberData)
            {
               nodeCount += memD.Lines.Count() * 2;
               lineCount += memD.Lines.Count();
               memberCount = lineCount;
            }
            // create Export-lists
            Dlubal.RFEM5.Node[] RFEMnodes = new Dlubal.RFEM5.Node[nodeCount];

            Dlubal.RFEM5.Line[] RFEMlines = new Dlubal.RFEM5.Line[lineCount];
            Dlubal.RFEM5.Member[] RFEMmembers = new Dlubal.RFEM5.Member[memberCount];

            // foreach set of input Memberdata
            foreach (ModelDataTyps.MemberData memData in memberData)
            {
               // foreach line in selected Memberdata
               foreach (Autodesk.DesignScript.Geometry.Line line in memData.Lines)
               {
                  // get start and endpoint of dynamoline
                  Autodesk.DesignScript.Geometry.Point startpoint = line.StartPoint;
                  Autodesk.DesignScript.Geometry.Point endpoint = line.EndPoint;

                  // Query for points existing
                  foreach (Node node in RFEMnodes)
                  {
                     if (Math.Round(node.X, 5) == Math.Round(startpoint.X, 5) && Math.Round(node.Y,
5) == Math.Round(startpoint.Y, 5) && Math.Round(node.Z, 5) == Math.Round(startpoint.Z, 5) &&
node.Comment == "Dlubal_dynamo") // letzte argument filtert ob ICH den punkt gesetzt habe somit
werden andere Punkte vernachlässigt. Was das soll muss ich nochmal rausfinden
                     {                                                                                    //
Keine Ahnung warum ich den scheiß gemacht habe
                        isStartpointExisting = true;
                        existingStartPointNo = node.No;
                     }

                     if (Math.Round(node.X, 5) == Math.Round(endpoint.X, 5) && Math.Round(node.Y, 5)
 == Math.Round(endpoint.Y, 5) && Math.Round(node.Z, 5) == Math.Round(endpoint.Z, 5) &&
node.Comment == "Dlubal_dynamo") // letzte argument filtert ob ICH den punkt gesetzt habe
                     {
                        isEndpointExisting = true;
                        existingEndPointNo = node.No;
                     }

                     if (isEndpointExisting == true && isStartpointExisting == true)
                     {
                        break;
                     }
                  }

                  //Set startnode
                  if (isStartpointExisting != true)
                  {
                     RFEMnodes[nodeListIndex].No = nodeNo;
                     RFEMnodes[nodeListIndex].X = startpoint.X;
                     RFEMnodes[nodeListIndex].Y = startpoint.Y;
                     RFEMnodes[nodeListIndex].Z = startpoint.Z;

                     nodeListIndex++;
                     nodeNo++;
                  }
```

```
                    //Set Endnode
                    if (isEndpointExisting != true)
                    {
                        RFEMnodes[nodeListIndex].No = nodeNo;
                        RFEMnodes[nodeListIndex].X = endpoint.X;
                        RFEMnodes[nodeListIndex].Y = endpoint.Y;
                        RFEMnodes[nodeListIndex].Z = endpoint.Z;

                        nodeListIndex++;
                        nodeNo++;
                    }

                    //Set Line, 4 options/possibilities

                    // only Endnode exists, 1 Node created previously
                    if (isEndpointExisting == true && isStartpointExisting == false)
                    {
                        RFEMlines[listIndex].NodeList = $"{RFEMnodes[nodeListIndex - 1].No},
{existingEndPointNo}";  // nodeListIndex - 1, da nach dem Erstellen erhöht wurde
                        RFEMlines[listIndex].No = lineNo;

                        lineNo++;
                    }
                    // only Startnode exists, 1 Node created previously
                    if (isEndpointExisting == false && isStartpointExisting == true)
                    {
                        RFEMlines[listIndex].NodeList = $"{existingStartPointNo},{ RFEMnodes[nodeListIndex
 - 1].No}";
                        RFEMlines[listIndex].No = lineNo;

                        lineNo++;
                    }

                    // both nodes existing, 0 Nodes created previously
                    if (isEndpointExisting == true && isStartpointExisting == true)
                    {
                        RFEMlines[listIndex].NodeList = $"{existingStartPointNo},{existingEndPointNo}";
                        RFEMlines[listIndex].No = lineNo;

                        lineNo++;
                    }

                    // no node existing, 2 Nodes created previously
                    if (isEndpointExisting == false && isStartpointExisting == false)
                    {
                        RFEMlines[listIndex].NodeList = $"{RFEMnodes[nodeListIndex - 2].No},
{RFEMnodes[nodeListIndex - 1].No}";
                        RFEMlines[listIndex].No = lineNo;

                        lineNo++;
                    }

                    // create Member
                    RFEMmembers[listIndex].LineNo = RFEMlines[listIndex].No;
                    RFEMmembers[listIndex].StartCrossSectionNo = memData.CrosssectionNumber;
                    RFEMmembers[listIndex].No = memberNo;

                    memberNo++;
                    listIndex++;
```

```
                //reset flag
                isStartpointExisting = false;
                isEndpointExisting = false;

            }
        }

        /////////////// create Member List End ////////////////////////////////////////////////
        /////////////// create NodalSupport list  ////////////////////////////////////////////

        if (nodalSupportData == null)
        {
            throw new Exception("missing nodalSupportData, insert it to solve problem");
        }
        // declare Variable
        int nodalSupCount = nodalSupportData.Count();
        int nodalSupPointsCount = 0;
        int nodalSupNo = 1;
        listIndex = 0;

        foreach (ModelDataTyps.NodalSupportData nodD in nodalSupportData)
        {
            nodalSupPointsCount += nodD.Points.Count();
        }
        // create Export-list
        NodalSupport[] nodalSupport = new NodalSupport[nodalSupCount];

        // foreach set of input nodalSupportData
        foreach (ModelDataTyps.NodalSupportData nodD2 in nodalSupportData)
        {
            bool firstTime = true;
            string nodesupplist = "";
            // foreach point in seleced nodalSupportData
            foreach (Autodesk.DesignScript.Geometry.Point point in nodD2.Points)
            {
                // Query for point-No., assumption all points already existing
                foreach (Node existingNode in RFEMnodes)
                {
                    if (Math.Round(existingNode.X, 5) == Math.Round(point.X, 5) &&
Math.Round(existingNode.Y, 5) == Math.Round(point.Y, 5) && Math.Round(existingNode.Z, 5) ==
Math.Round(point.Z, 5))
                    {

                        if (firstTime == true)
                        {
                            nodesupplist = existingNode.No.ToString();
                            firstTime = false;
                        }
                        else
                        {
                            nodesupplist += $",{existingNode.No}";
                        }

                        break;
                    }

                }

            }
```

```
    // Create nodalSupport
    nodalSupport[listIndex].No = nodalSupNo;
    nodalSupport[listIndex].NodeList = nodesupplist;
    nodalSupport[listIndex].RestraintConstantX = -1;
    nodalSupport[listIndex].RestraintConstantY = 0;
    nodalSupport[listIndex].RestraintConstantZ = -1;
    nodalSupport[listIndex].SupportConstantX = -1;
    nodalSupport[listIndex].SupportConstantY = -1;
    nodalSupport[listIndex].SupportConstantZ = -1;

    nodalSupNo++;
    listIndex++;
}

/////////////// create NodalSupport List End ////////////////////////////////////////
/////////////// create LoadCase List ////////////////////////////////////////////////

if (loadCaseData == null)
{
    throw new Exception("missing loadCaseData, insert it to solve problem");
}

// declare variable & create Export-list
int loadCaseCount = loadCaseData.Count();
LoadCase[] RFEMloadcases = new LoadCase[loadCaseCount];
listIndex = 0;

// foreach set of input loadCaseData
foreach (ModelDataTyps.LoadCaseData loadDa in loadCaseData)
{
    // create loadCaseData
    RFEMloadcases[listIndex].Description = loadDa.name;
    RFEMloadcases[listIndex].Loading.No = loadDa.loadcaseNumber;
    if (loadDa.activeSelfweight == true)
    {
        RFEMloadcases[listIndex].SelfWeight = true;
        RFEMloadcases[listIndex].SelfWeightFactor.Z = 1;
    }
    else
        RFEMloadcases[listIndex].SelfWeight = false;

    RFEMloadcases[listIndex].ToSolve = true;
    RFEMloadcases[listIndex].ActionCategory = ActionCategoryType.Permanent;

    listIndex++;
}

/////////////// create LoadCase List End ////////////////////////////////////////////
/////////////// create LoadCase Combination List ////////////////////////////////////

// declare Variable
LoadCombination[] RFEMloadCombs = null;
int loCoCounter = 0;

// logic for catching missing Input
if (loadCaseCombData != null)
{
    listIndex = 0;

    try
    {
```

```
                loCoCounter = loadCaseCombData.Count();
            }
            catch (Exception e)
            {
                loCoCounter = 1;
            }

            // create Export-list
            RFEMloadCombs = new LoadCombination[loCoCounter];

            for (int i = 0; i < loCoCounter; i++)
            {

                // Create LoadCaseCombination
                RFEMloadCombs[listIndex].Loading.No = loadCaseCombData[i].loadCombNumber;
                RFEMloadCombs[listIndex].ToSolve = true;
                RFEMloadCombs[listIndex].Definition = $"
{loadCaseCombData[i].combfactor1}*lc{loadCaseCombData[i].loadCaseNummer1} +
{loadCaseCombData[i].combfactor2}*lc{loadCaseCombData[i].loadCaseNummer2}";

                listIndex++;
            }

        }

        /////////////// create LoadCase Combination List End ////////////////////////////////
        /////////////// create MemberLoad List  ////////////////////////////////////////////

        // declare Variable
        MemberLoad[] RFEMmemberloads = null;

        if (memberloadData != null)
        {
            int memLoadCount = memberloadData.Count();
            RFEMmemberloads = new MemberLoad[memLoadCount];
            listIndex = 0;
            bool isfirstmemberobject = true;
            string memberObjectList = "";
            int MemberloadNumber = 1;

            // foreach set of input MemberLoadData
            foreach (ModelDataTyps.MemberLoadData memLoDa in memberloadData)
            {
            // Query for line-No.(equal to member-no.)
               // Query for node - No. to find line
                foreach (Line aline in memLoDa.lines)
                {
                    Autodesk.DesignScript.Geometry.Point startPoint = aline.StartPoint;
                    Autodesk.DesignScript.Geometry.Point endPoint = aline.EndPoint;
                    int rfemStartNodeNumber = -1;
                    int rfemEndNodeNumber = -1;

                    foreach (Node node in RFEMnodes)
                    {
                        if (node.X == startPoint.X && node.Y == startPoint.Y && node.Z == startPoint.Z)
                        {
                            rfemStartNodeNumber = node.No;
                        }

                        if (node.X == endPoint.X && node.Y == endPoint.Y && node.Z == endPoint.Z)
                        {
```

```
                rfemEndNodeNumber = node.No;
            }

            if (rfemEndNodeNumber != -1 && rfemStartNodeNumber != -1)
            {
                break;
            }
        }


        string lineNodeList = $"{rfemStartNodeNumber},{rfemEndNodeNumber}";
        int rfemLineNumber = -1;

        Dlubal.RFEM5.Line[] rfemLines = RFEMlines;

        // find line
        foreach (Dlubal.RFEM5.Line rline in rfemLines)
        {
            if (rline.NodeList == lineNodeList)
            {
                rfemLineNumber = rline.No;
                break;
            }
        }

        // set memberList for Load
        if (isfirstmemberobject == true)
        {
            if (rfemLineNumber == -1)
                break;
            memberObjectList = rfemLineNumber.ToString();
            isfirstmemberobject = false;
        }
        else
        {
            if (rfemLineNumber == -1)
                break;
            memberObjectList += $",{rfemLineNumber}";
        }

    }

    RFEMmemberloads[listIndex].No = MemberloadNumber;
    RFEMmemberloads[listIndex].ObjectList = memberObjectList;
    RFEMmemberloads[listIndex].Distribution = LoadDistributionType.TrapezoidalType;
    RFEMmemberloads[listIndex].Magnitude1 = memLoDa.magnitude1;
    RFEMmemberloads[listIndex].Magnitude2 = memLoDa.magnitude2;
    RFEMmemberloads[listIndex].RelativeDistances = memLoDa.relativeDistances;
    RFEMmemberloads[listIndex].DistanceA = memLoDa.distanceA;
    RFEMmemberloads[listIndex].DistanceB = memLoDa.distanceB;

    listIndex++;
    MemberloadNumber++;
  }
}
/////////////// create MemberLoad List End  ////////////////////////////////////////////////
/////////////// Export ////////////////////////////////////////////////////////////////////

if (startExport == true)
{
  modelData.PrepareModification();
```

```
                modelData.SetMaterials(RFMmaterials);
                modelData.SetCrossSections(RFMcrosssecs);
                modelData.SetNodes(RFEMnodes);
                modelData.SetLines(RFEMlines);
                modelData.SetMembers(RFEMmembers);
                modelData.SetNodalSupports(nodalSupport);
                modelData.FinishModification();

                modelLoads.PrepareModification();
                modelLoads.SetLoadCases(RFEMloadcases);
                if (loadCaseCombData != null)
                {
                    modelLoads.SetLoadCombinations(RFEMloadCombs);
                }

                modelLoads.FinishModification();

                for (int i = 0; i < memberloadData.Count(); i++)
                {
                    ILoadCase loadcase = modelLoads.GetLoadCase(memberloadData[i].loadCaseNumber,
ItemAt.AtNo);
                    loadcase.PrepareModification();
                    loadcase.SetMemberLoad(RFEMmemberloads[i]);
                    loadcase.FinishModification();
                }

            }
            ////////////////////// Calculation ////////////////////////////////

            Dlubal.STEEL_EC3.RESULTS_DESIGN[] results = null;

            if (startCalculation == true)
            {
                // connect to STEEL EC3

                Dlubal.STEEL_EC3.Module steelModule =
(Dlubal.STEEL_EC3.Module)model.GetModule("STEEL_EC3");

                // delete all calc-cases
                int caseCount = steelModule.moGetCaseCount();
                if (caseCount != 0)
                {
                    for (int i = 1; i <= caseCount; i++)
                    {
                        steelModule.moDeleteCase(i, Dlubal.STEEL_EC3.ITEM_AT.AT_INDEX);
                    }
                }

                //Create Calc-Case
                Dlubal.STEEL_EC3.ICase moCase = steelModule.moSetCase(1, "Dynamo Calculation");
                moCase.moSetMemberList("all");
                int setLoadCombcounter = 0;

                Dlubal.STEEL_EC3.ULS_LOAD[] loads = new Dlubal.STEEL_EC3.ULS_LOAD[loCoCounter +
loadCaseCount];

                //set loads
                for (int i = 0; i < loCoCounter; i++)
                {
                    loads[i].DesignSituation = Dlubal.STEEL_EC3.DESIGN_SITUATION.DS_FUNDAMENTAL;
                    loads[i].No = RFEMloadCombs[i].Loading.No;
```

```
                    loads[i].Type = Dlubal.STEEL_EC3.ILOAD_TYPE.ILOAD_GROUP;
                    setLoadCombcounter++;
                }

                for (int i = 0; i < loadCaseCount; i++)
                {
                    loads[setLoadCombcounter + i].DesignSituation =
Dlubal.STEEL_EC3.DESIGN_SITUATION.DS_FUNDAMENTAL;
                    loads[setLoadCombcounter + i].No = RFEMloadcases[i].Loading.No;
                    loads[setLoadCombcounter + i].Type = Dlubal.STEEL_EC3.ILOAD_TYPE.ILOAD_CASE;
                }

                moCase.moSetULSLoads(loads);
                moCase.moCalculate();
                results = moCase.moGetResults().moGetDesignByMemberAll();
            }

            // unlock RFEM
            app.UnlockLicense();
            return new ModelDataTyps.ResultData(results);
        }
        catch (Exception e)
        {
            IApplication app = Marshal.GetActiveObject("RFEM5.Application") as IApplication;
            app.UnlockLicense();
            throw new Exception($"{e.Message}");
        }
    }
  }
}

namespace ModelDataTyps
{
  // Create custom returntyps

  [IsVisibleInDynamoLibrary(false)]
  public struct MemberData
  {
    public MemberData(Line[] Lines, int CrosssectionNumber)
    {
      this.Lines = Lines;
      this.CrosssectionNumber = CrosssectionNumber;
    }

    public Line[] Lines { get;}
    public int CrosssectionNumber { get;}
  }

  [IsVisibleInDynamoLibrary(false)]
  public struct NodalSupportData
  {
    public NodalSupportData(Autodesk.DesignScript.Geometry.Point[] Points)
    {
      this.Points = Points;
    }

    public Autodesk.DesignScript.Geometry.Point[] Points { get; }

  }

  [IsVisibleInDynamoLibrary(false)]
```

```
public struct ResultData
{

    public ResultData(Dlubal.STEEL_EC3.RESULTS_DESIGN[] Results)
    {
        this.Results = Results;
    }

    public Dlubal.STEEL_EC3.RESULTS_DESIGN[] Results { get; }

}

[IsVisibleInDynamoLibrary(false)]
public struct MaterialData {

    public MaterialData(string MaterialID, int materialNo)
    {
        this.MaterialID = MaterialID;
        this.materialNo = materialNo;
    }

    public string MaterialID { get; }

    public int materialNo { get; }
}

[IsVisibleInDynamoLibrary(false)]
public struct CrosssectionData
{
    public CrosssectionData(string crosssectionID, int materialNo, int crosssectionNo)
    {
        this.CrosssectionID = crosssectionID;
        this.MaterialNo = materialNo;
        this.CrosssectionNo = crosssectionNo;
    }

    public string CrosssectionID { get; }
    public int MaterialNo { get; }
    public int CrosssectionNo { get; }
}

[IsVisibleInDynamoLibrary(false)]
public struct LoadCaseData
{
    public LoadCaseData(string name, int loadcaseNumber, bool activeSelfweight )
    {
        this.name = name;
        this.activeSelfweight = activeSelfweight;
        this.loadcaseNumber = loadcaseNumber;
    }
    public string name { get; }
    public bool activeSelfweight { get; }

    public int  loadcaseNumber { get; }

}

[IsVisibleInDynamoLibrary(false)]
public struct LoadCaseCombData
{
    public LoadCaseCombData(int loadCombNumber, double combfactor1, int loadCaseNummer1,
```

```
double combfactor2, int loadCaseNummer2)
        {
            this.loadCaseNummer1 = loadCaseNummer1;
            this.loadCaseNummer2 = loadCaseNummer2;
            this.combfactor1 = combfactor1;
            this.combfactor2 = combfactor2;
            this.loadCombNumber = loadCombNumber;
        }

        public int loadCombNumber { get; }
        public double combfactor1 { get; }
        public double combfactor2 { get; }
        public int loadCaseNummer1 { get; }

        public int loadCaseNummer2 { get; }
    }

    [IsVisibleInDynamoLibrary(false)]
    public struct MemberLoadData
    {
        public MemberLoadData(int loadCaseNumber, Line[] lines, double magnitude1, double  magnitude2,
 bool relativeDistances, double distanceA, double distanceB)
        {
            this.loadCaseNumber = loadCaseNumber;
            this.lines = lines;
            this.magnitude1 = magnitude1;
            this.magnitude2 = magnitude2;
            this.relativeDistances = relativeDistances;
            this.distanceA = distanceA;
            this.distanceB = distanceB;
        }
        public int loadCaseNumber { get; }
        public Line[] lines { get; }
        public double magnitude1 { get; }
        public double magnitude2 { get; }
        public bool relativeDistances { get; }
        public double distanceA { get; }
        public double distanceB { get; }
    }

}
```