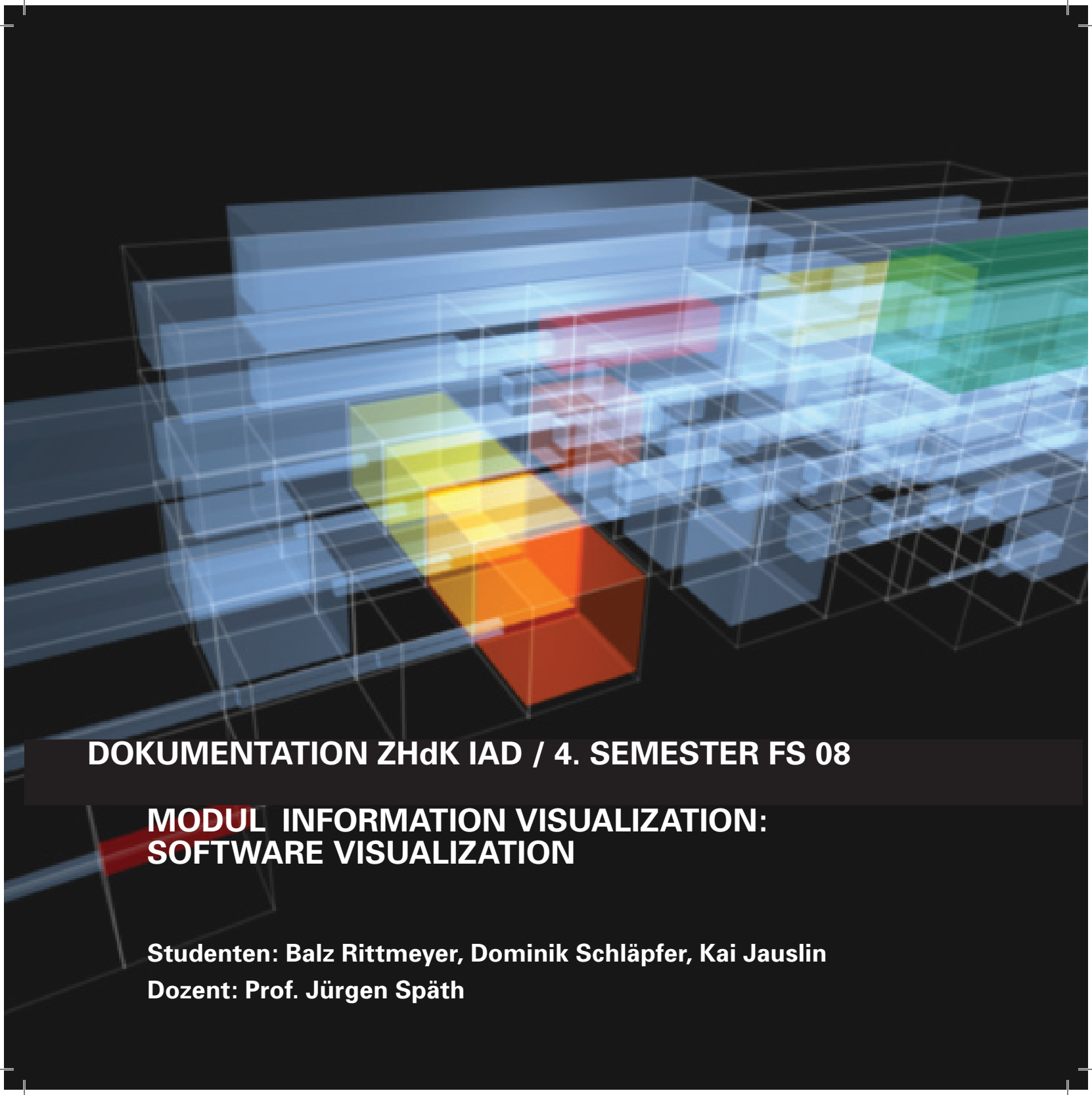


DOKUMENTATION ZHdK IAD / 4. SEMESTER FS 08

MODUL INFORMATION VISUALIZATION: SOFTWARE VISUALIZATION

Studenten: Balz Rittmeyer, Dominik Schläpfer, Kai Jauslin

Dozent: Prof. Jürgen Späth

INHALTSVERZEICHNIS

Abstract:

Moderne Objekt Orientierte Programme (in der Folge OOP genannt) sind hierarchische Systeme mit vielen tausende untereinander abhängigen Subsystemen.

Visualisierung hilft dem Entwickler diese grossen, komplexen Systeme besser zu verstehen. Dieses Thesenpapier reflektiert unsere Recherche und ist mit seinen Schlussfolgerungen Ausgangspunkt für die Ausarbeitung eines Prototypen für Software-Visualisierungen.

01.01 EINLEITUNG

Software-Systeme gehören zu etwas vom Komplexesten was moderne Errungenschaften hervorgebracht haben. OOP-Systeme bestehen aus vielen Millionen Lines of Codes (LOC) und durchlaufen oftmals einen langen Evolutionsprozess. Visualisierung kann hierbei helfen den Unterhalt und ein allfälliges Redesign effizient zu unterstützen.

Softwarevisualisierung, als eine Unterkategorie von Informations-Visualisierung, ist die Disziplin um ein besseres Verständnis von Software-Systemen zu bewerkstelligen.

Dabei unterscheiden wir nach verschiedenen Programm-Aspekten:

01 Static Structure

02 Runtime Behaviour

03 Evolution, Development Process

In unserer Arbeit werden wir uns vorallem auf Punkt 01 fokussieren.

Es ist elementar für die Visualisierung, dass die bildhafte Repräsentation im Hinblick auf den Charakter der visualisierten Informationen interpretierbar ist. Hierbei kann der Gebrauch von Metaphern aus der physischen Welt sinnvoll sein, da der Betrachter seine gewohnte Wahrnehmungsfähigkeit auf die Visualisierung projizieren kann.

2 Software-Visualisierungen realisiert mit dem RIGI-Framework.

ftp://ls10-ftp.cs.uni-dortmund.de/pub/Technische-Berichte/Fronk_SWT-Memo-171.pdf

01.02 ANALYSE

Der populärste Ansatz zur Softwarevisualisierung ist UML. UML wie andere Diagramm Visualisierungen, beinhaltet keine grosse Bildsprache. Einerseits vereinfacht dies die interpretier- und lesbarkeit der Visualisierung, andererseits nimmt dadurch aber auch die Informationsdichte und die Kontrolle über den Grad der Abstraktion ab.

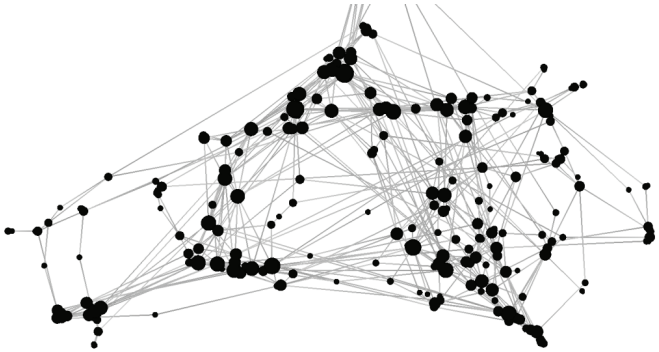
Es gibt diverse Tools, welche die Software in ihre einzelnen Komponenten zerlegen und diese dann mittels grafischen Repräsentanten visualisieren. Wir unterscheiden hierbei zwischen zwei- und dreidimensionalen Tools:

2D: Diese Visualisierungstechniken verfügen über gute Navigationseinrichtungen, welche es dem Betrachter erlauben, verschiedene "Views" bzw. verschiedene Abstraktions-Level, der verschiedenen Systemen und Subsystemen einzunehmen. Beispiele: Rigi, SHriMP, Portable Bookshelf.

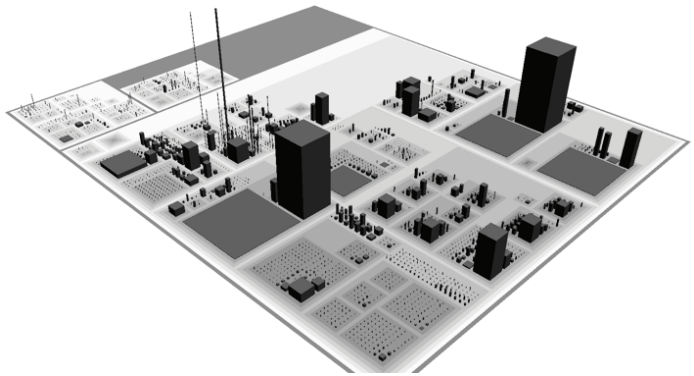
3D: Mit der dritten Dimension entsteht eine neue Ebene an Informationsdichte und neue Versuche den gewonnen Raum nutzbar zu machen. Beispiele: Koike, Narcissus, NestedVision3D, ArchView, CrocoCosmos.

Empirische Studien die 2D-Tools mit 3D-Tools vergleichen kommen zu unterschiedlichen Resultaten. Während bei 2D-Visualisierungen oft die fehlende dritte Dimension und der nicht vorhandene Informationsraum bemängelt wird, fehlt es bei 3-D Visualisierungen oft an Orientierung und Informationsobjekte werden unwillentlich verdeckt.

Ein interessanter Ansatz um dieses Problem aus der Welt zu schaffen, ist die sogenannte 2.5-dimensionale Visualisierung: Hierbei werden 3-dimensionale Objekte mit einer hohen Informationsdichte auf einem 2-dimensionalen Raum angeordnet. Beispiele: THEMA, Software World, Component City.



Crococosmos Software-Visualisierung
http://www-sst.informatik.tu-cottbus.de/CrocoCosmos/



Code-City
http://www.inf.unisi.ch/phd/wettel/codecity.html

01.03 STRUKTURMODELL VON OOP

Ein Struktur-Modell beschreibt die existierenden Einheiten, deren Hierarchie und deren Beziehungen untereinander. Vereinfacht dargestellt sieht dies so aus:

Die Knoten repräsentieren hierbei die einzelnen Einheiten. Die Linien die Beziehungen und die Baum-Struktur die Hierarchie.

Dieses Modell beinhält vier verschiedene Einheiten:

Packages / Classes /Methods / Attributes.

Die Visualisierung ist eine erste Abstraktion der Methoden aus dem Source-Code, auf welche sich eine Skalierung anwenden lässt, ohne dass dabei wesentliche Informationen über Einheiten, Beziehung und Hierarchie verloren gehen.

Aus dem Modell lassen sich auch noch weitere wichtige Informationen lesen:

Class-Inheritance / Method Calls / Attribute Calls

Klassen können von anderen Klasse erben, Methoden können andere Methoden aufrufen und Methoden können auf Attribute zugreifen.

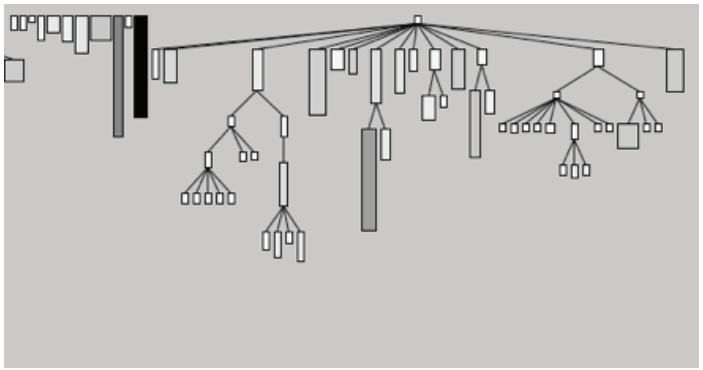
Solche Modelle lassen sich direkt aus der Source von OOP-Code generieren. Java-Beispiel: SNiFF+ (Windows), Sotograph.

Um das erwähnte OOP-Modell weiter zu abstrahieren und dessen Visualisierung zu erweitern, ist es von Bedeutung zu wissen, was für Entwickler relevante Grössen in Softwareprojekten sind. Dazu zusammengefasse ein Interview mit dem Titel: "Erik Doernenburg on Software Visualization"3.

Der unkompillierte Source-Code an sich ist eine denkbar schlechte Form um in einer Software den Überblick zu bewahren. Dieser muss in irgendeiner Form visualisiert und verdichtet werden. Ohne die Verdichtung ist es unmöglich in einer Software, die über mehrere Millionen LOC (lines of code) verfügt, den Durchblick zu bewahren.

Daher ist es wichtig mit messbaren, vergleichbaren Einheiten zu arbeiten. Für Entwickler relevante Masse sind:

Länge einer Klasse / Länge einer Methode / n-Pfad-Komplexität / Klassenabhängigkeit / Klasseninstanzierung



Code-Crawler
http://www.inf.unisi.ch/faculty/lanza/codecrawler.html

Tarantula
http://pleuma.cc.gatech.edu/aristotle/Tools/tarantula/

01.04 FAZIT RECHERCHE

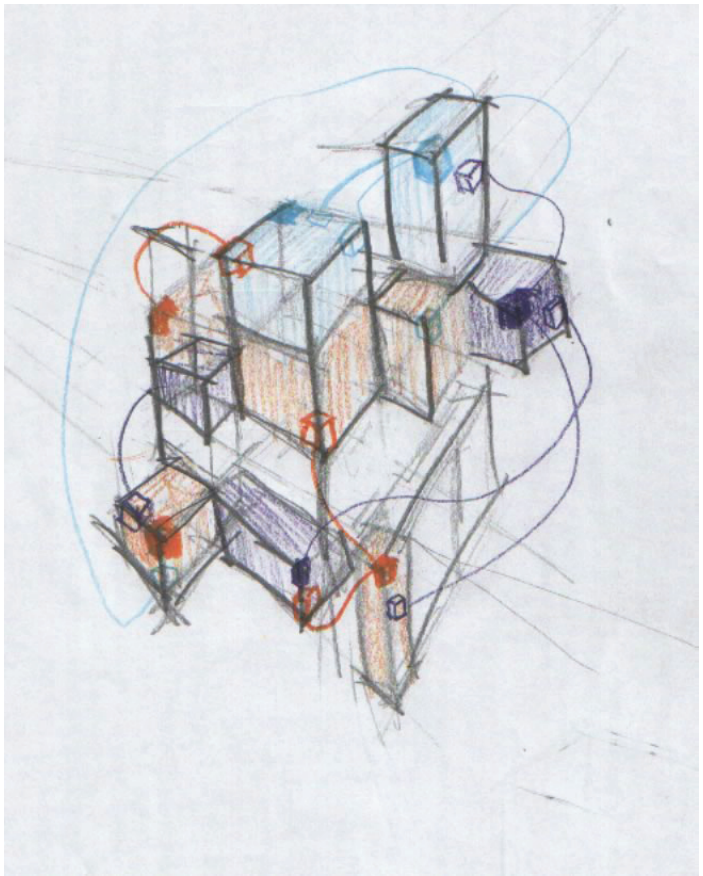
In einem nächsten Schritt werden mit den erarbeiteten Parametern erste Skizzen und Visualisierungen erstellt. Ziel ist es mit den Skizzen schon in der Recherche erörterte Fragen zu beantworten. Dies wären insbesondere:

-Räumliche Aspekte (2D, 3D)

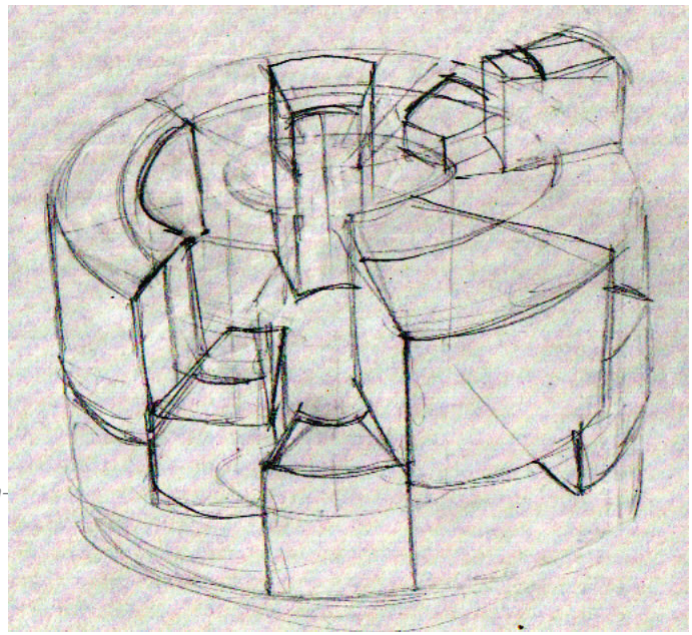
-Verwendung einer Metapher (ja, nein?)

-Informations-Verdichtung

-Makro- Mikro-Ebene



Software-Konstrukt mit Würfeln, deren Grösse und Farbe verschiedene Software-Metriken darstellen.



3-D Software-Visualisierung mit Zylindersegmenten.

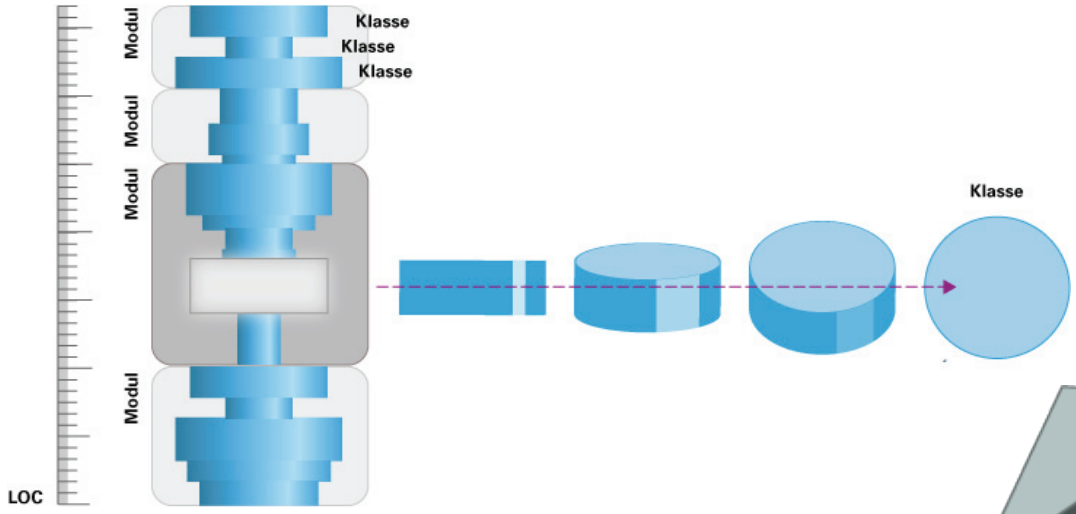
02.01 SKIZZEN ANALOG

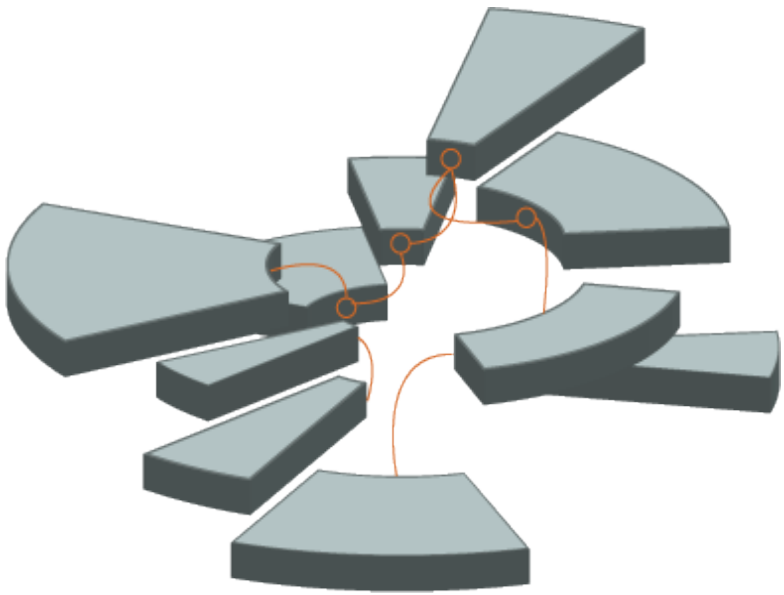
In einem nächsten Schritt werden mit den erarbeiteten Parametern erste Skizzen und Visualisierungen erstellt. Ziel ist es mit den Skizzen schon in der Recherche erörterte Fragen zu beantworten.

An dieser Stelle sind nur Skizzen ausgewählt, deren Charakteristik im späteren Verlauf des Projekts wieder aufgegriffen werden.

Idee: Ganz links ist eine Skala mit der Metrik Lines of Code zu sehen. Daneben baut sich ein «Software-Turm« auf, welcher in verschiedene weitere Einheiten unterteilt ist (Module, Klassen). Weiter ablesbare Metrik sind die Calls: Je mehr Ausschlag die Klassen nach links bzw. rechts haben, desto mehr incoming (rechts) bzw. outgoing (links) Calls hat die jeweilige Klasse.

Weiter zeigt die Skizze folgende Interaktionsidee: Durch Drag and Drop kann eine einzelne Klasse aus dem Stapel herausgezogen werden. Je nach Lage verfügt der Betrachter nun über verschiedene Aufsichten auf die Klasse und kann verschiedenartige Schlüsse daraus ziehen.





Kreisförmige Anordnung verschiedener Klassen im selben Package. Auch hier wird mit einem 3-D Modell experimentiert, wobei je nach Aufsicht verschiedene Informationen aus dem Schema herauslesbar sind.

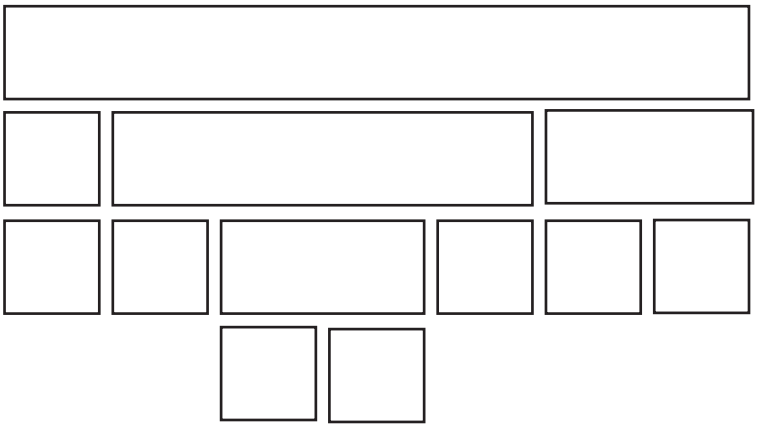
Im Beispiel schematisch dargestellt sind folgende Metriken:

Length of Code in Form der Kreissegmentlänge.

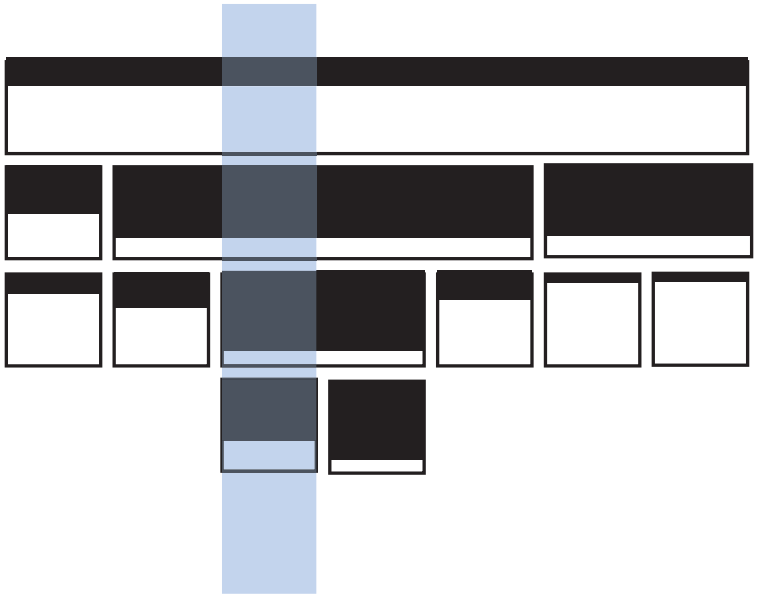
Hierarchie (Vererbung in der 3-D Ansicht).

02.02 SKIZZEN DIGITAL

Erste digitale Skizzen die versuchen die Ideen aus der analogen Skizzenserie aufzugreifen und zu verbessern.



Das Prinzip des Icicle Trees: Der Tree ist von oben nach unten linear lesbar. Das unterste Element (in diesem Fall ein Quadrat) ist die kleinste Einheit des Baumes. Jede Verästelung endet in einem Quadrat. Je nach Komplexität ist dies im Baum früher oder später der Fall. Das oberste Element ist immer 100 %.



Der obige Icicle Tree zeigt die Hierarchie eines Method Calls auf (=> die direkten Outgoing Calls einer Mehode).

Die Idee ist es nun verschiedene zusätzliche Metriken im Baum darzustellen. Im Beispiel nebenan werden noch die Incoming Calls einer Methode dargestell (Strichdicke).

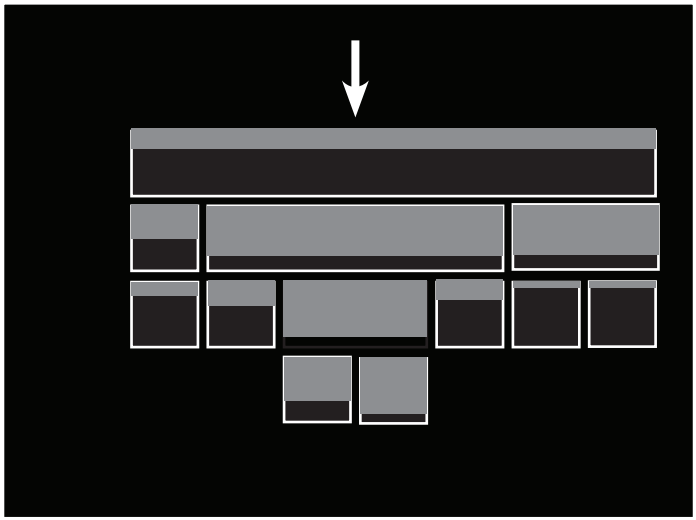
03.01 ICICLE TREE

Um der Komplexität von Softwarevisualisierung gerecht zu werden, müssen wir die Parameter, welche wir darstellen wollen einschränken. Das Ziel ist es Method Calls und deren Verhältnis In-Coming Calls (FAN IN), Outgoing Calls (FAN OUT) darzustellen und zu Vergleichen.

Eine weitere Einschränkung ist, dass wir immer nur einen Method Call bzw. einen Use Case, aufs mal anschauen. Zum Beispiel: öffnen einer Webseite im Browser.

Die Darstellungsform eines solchen Method-Calls wollen wir mit einem bewährten Baumsystem bewerkstelligen. Da ein Baum, wie ein Method Call auch, ein Anfang und ein Ende hat und auf seinem Weg beliebig viele Verästelungen machen kann.

Da wir in der Thematik Method Calls sehr viele Varianten von Baumstrukturen haben, haben wir uns für eine Form von Baum entschieden, welche diese Anforderung am optimalsten abdeckt: Der Icicle Tree.



Die ortografische Seitenansicht des Baums ergibt dieses Bild. Durch die Überlagerung der verschiedenen Methoden im Baum wird das Bild lesbar.

Der Blick per Vogelperspektive auf den Baum ergibt durch die Überlagerung und die Transparenz der Baumelemente folgendes lesbares Bild:

03.02 BAUMSTRUKTUR UND 3-D

Mit dem Hinzufügen der dritten Dimension gewinnt die Baumstruktur an Informationsraum. Dieser neu gewonnene Raum wird durch Interaktion dem potentiellen Benutzer durch Interaktion erschliessbar gemacht.
Unsere Konzeption sieht vor, dass der Baum drehbar wird und von verschiedenen Seiten betrachtet werden kann. Die einzelnen Baum-Elemente stellen nun durch Grösse und Farbe verschiedene Metriken dar (In unserem Beispiel steht die Farbe für die Incoming Calls bei Runtime, die Grösse für die generellen Global Incoming Calls). Die Elemente sind transparent, wodurch Überlagerungen entstehen, welche in orthografischer, sowie perspektivischer Ansicht, vielsagende Bilder generieren.

Eine weitere Idee ist, in der Z-Achse des 3-D Raums eine Versionierung des Method-Calls aufzuzeichnen, womit in den verschiedenen Ansichten Vergleiche über die Evolution gemacht werden können.

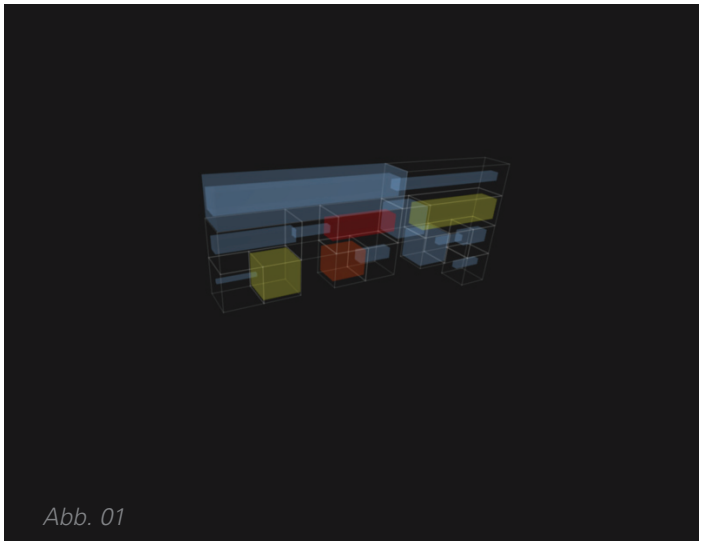


Abb. 01

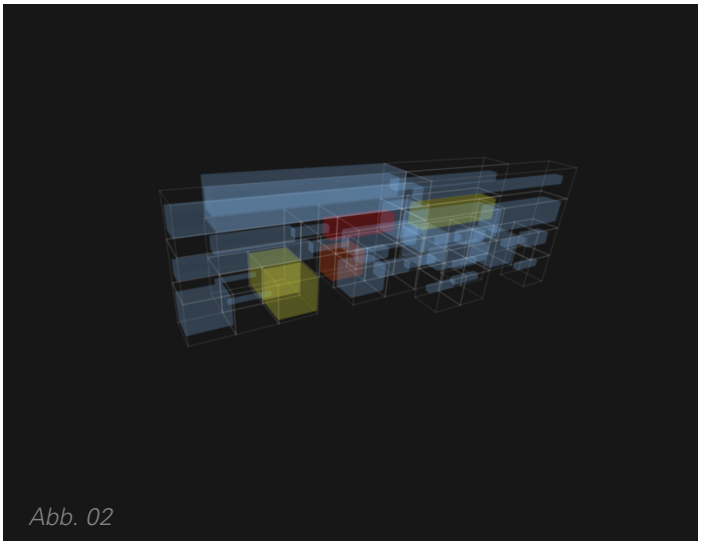


Abb. 02

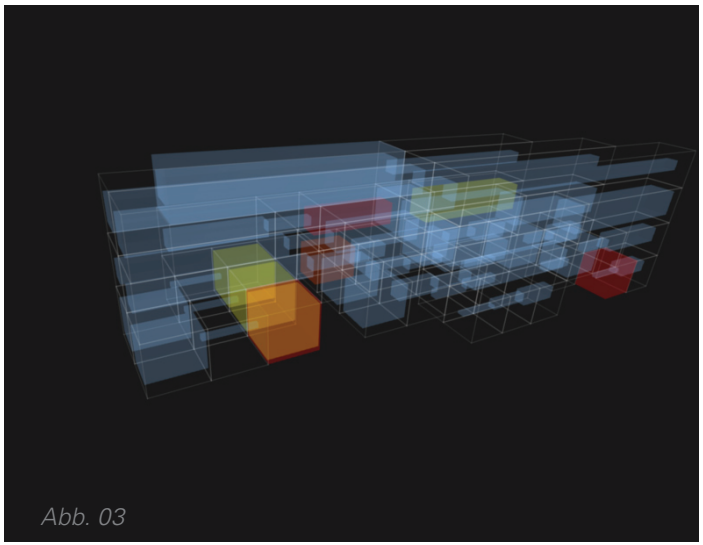


Abb. 03

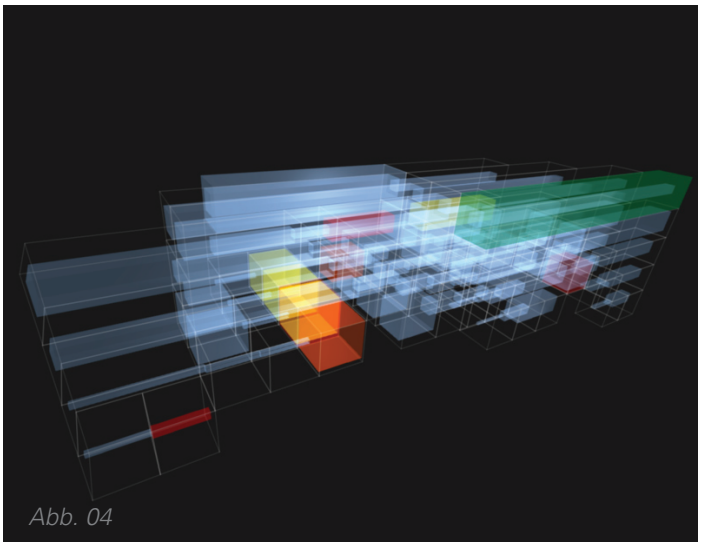


Abb. 04

04.01 VERSIONIERUNG

Anschauungsbeispiel zur Versionierung. Mit zunehmender Version wird die Methode komplexer. Sogenannte Hot-Spots (Methoden die oft aufgerufen werden) entstehen, werden in späteren Versionen «korrigiert», neue entstehen.

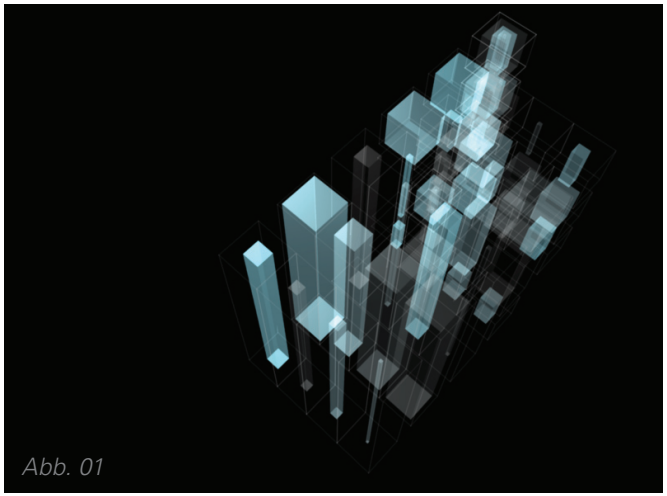


Abb. 01

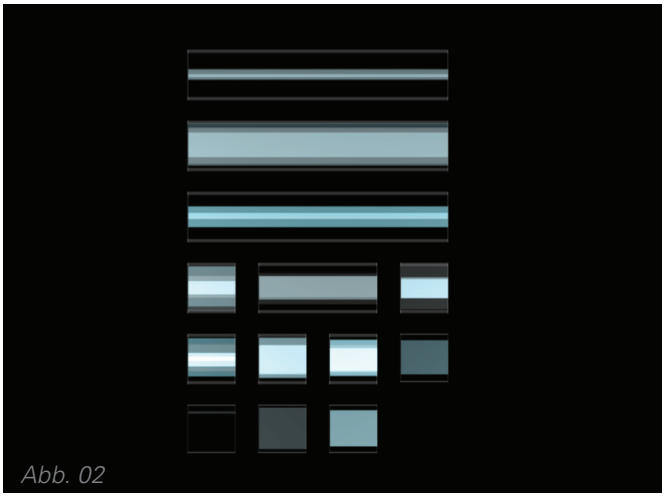


Abb. 02

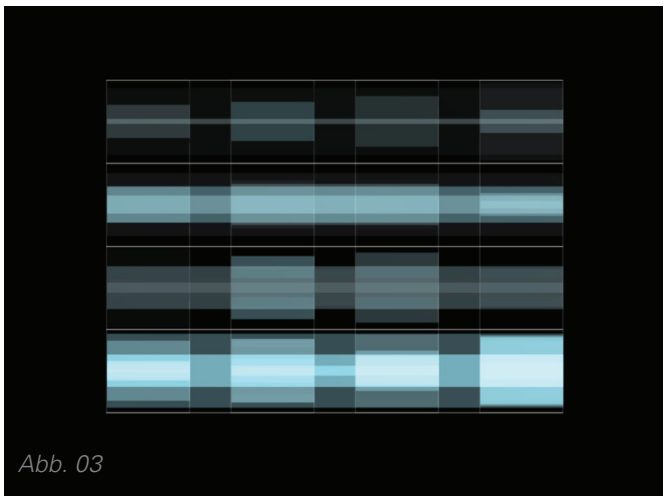


Abb. 03

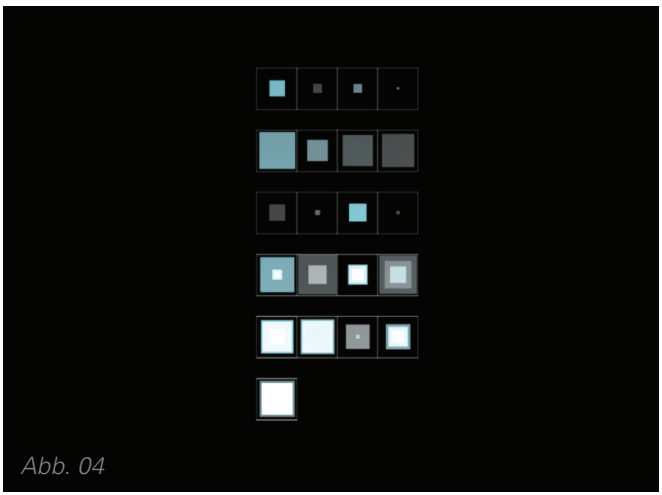


Abb. 04

04.02 ORTHOGRAFISCHE VIEWS

In Abb.01 ist das perspektivische 3-D Modell, mit welchem der User interagieren kann.

Abb.02 zeigt den Baum von vorne, in dieser Ansicht lässt sich sehr gut die Hierarchie des Method-Calls betrachten. Auch die Global Incoming Calls (Höhe des Baum-Element Inhalts) sind gut lesbar.

Abb. 03 und 04 zeigen Seitenansicht und Aufsicht auf den Baum. In der Z-Achse sind nun noch die Versionierungen zu sehen. Die Bilder werden vergleichbar.

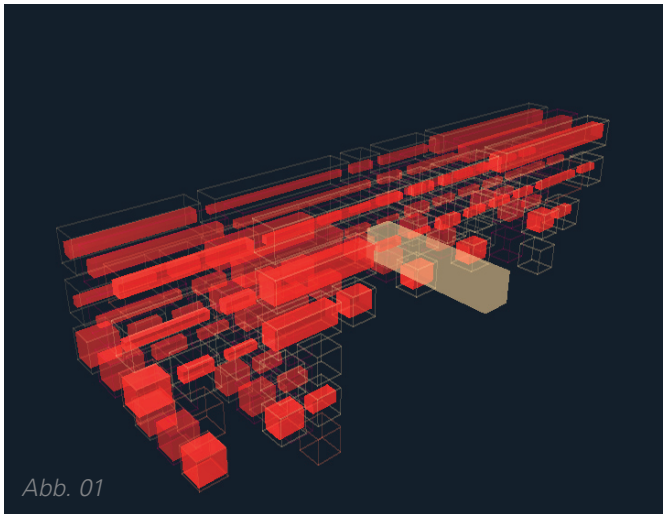


Abb. 01

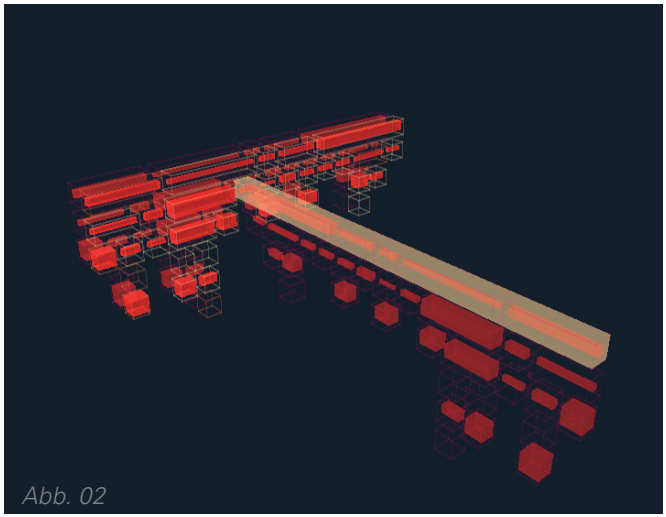


Abb. 02

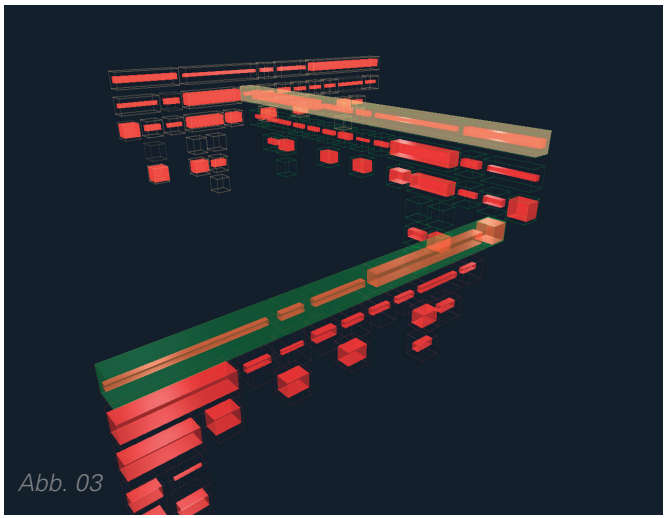


Abb. 03

04.03 MIKRO-MAKRO INTERAKTION

Das Interaktionsmodell ist hier ausgearbeitet. In Abb.01 sind Packages die bei einem bestimmten Use-Case aufgerufen werde in der Baumform dargestellt. Der Betrachter kann nun an einem bestimmten Package ziehen wodurch die im Package aufgerufenen Klassen in Baumform dargestellt werden. Auf der Klassen-Ebene kann nun der Benutzer in die unterste Ebene wechseln und eine Methode aus dem Baum herausziehen (Abb.03).

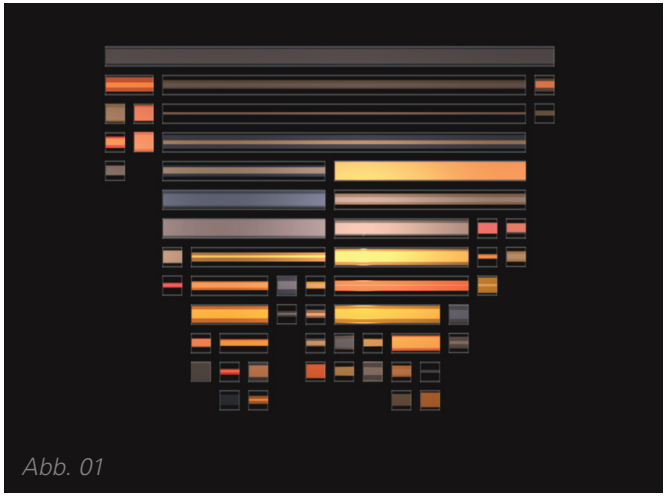


Abb. 01

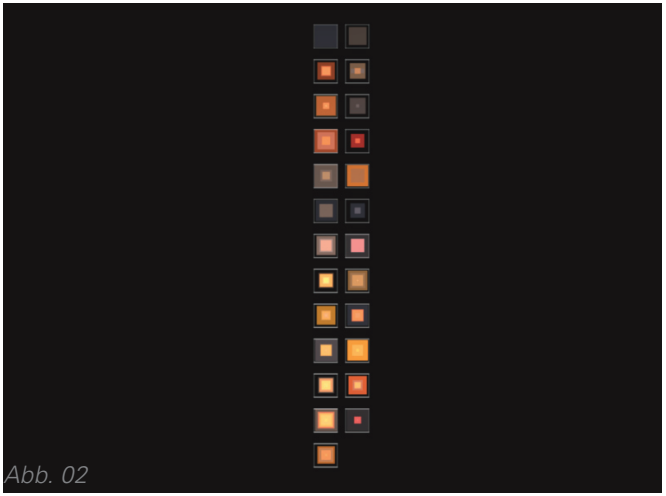


Abb. 02

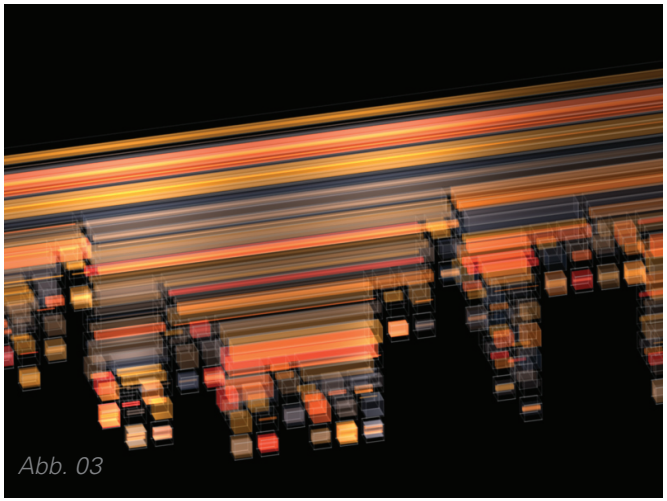


Abb. 03

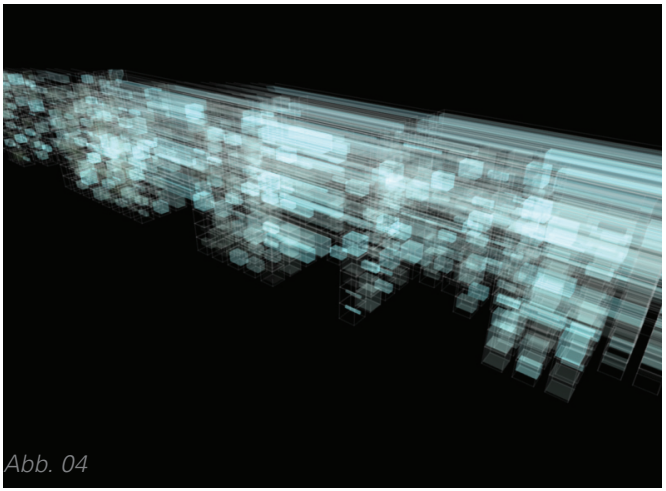


Abb. 04

04.04 AUGMENTED COMPLEXITY

An dieser Stelle stehen komplexe Bäume (allerdings keine Realdaten). Die Bilder werden je nach View nicht mehr im Detail lesbar und hinterlassen nur noch einen Grobeindruck.