3D Data and Model Management for the Geosciences with Particular Emphasis on Topology and Time

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS

von der Fakultät für

Bauingenieur-, Geo- und Umweltwissenschaften

des Karlsruher Instituts für Technologie (KIT)

genehmigte

DISSERTATION

von

Dipl.-Geogr. Edgar Butwilowski

aus Novosibirsk

Tag der mündlichen
Prüfung: 16. Juli 2015

Referent: Prof. Dr. Martin Breunig, Karlsruher Institut für Technologie

Korreferent: Prof. Dr. Mulhim Al-Doori, American University in Dubai

Karlsruhe 2015

Abstract of the Doctoral Thesis

The doctoral thesis deals with the examination of methods for topological data handling, as well as designing and implementing a toolkit for the geosciences for management of topological and temporal 3D data and models in geo-database architectures. The conceptual work and realisation process takes into account the management of multi LoD since it is considered to be an integral part of geo-modelling. The thesis presents the current state of processes, modelling tools, and database management systems in application-oriented use cases. An insight into the topics of model integration, abstraction of geodata and spatio-temporal modelling is given, including references to early pioneering, basic, and contemporary literature.

It is shown that geoscientific data is characterized by a heterogeneity of models and applications. The integration of multiple models and applications into common architectures is a relevant goal of the international scientific community. The topic of multi-representation is deepened by discussing the concepts of hierarchical digital landscape models, progressive mesh representation, multiple topological representations and others. The topic of spatio-temporal modelling is tackled through the discussion of such concepts as TimeStep, temporal point tube model and others.

The design of a Topology Module for the modelling of spatio-temporal topological objects is explained in detail. A concept and implementation for modelling and handling the topology of the cells of a complex geo-object is presented. Since the concept differentiates into net level and object level, it allows to create “big cells” that in turn comprise an arbitrary number of cells. The implementation of the model provides an iterator framework that allows for a flexible navigation on the topological structure. Editing methods for the topological structure are introduced and discussed. The concept of net level and object level is a sound preparation for the management of LoD of cell net components. Since net level and object level already exist, there is an architectural foundation which can be extended by additional detail levels. The spatio-temporal model of the Topology Module builds on the foundation of the Temporal Joint Model. Continuous geometric change can be processed through the Temporal Joint Model, whereas discrete topological change of “big cells” is modelled by the spatio-temporal model of the Topology Module. Finally, different aspects of runtime and memory performance of the Topology Module are evaluated.

Zusammenfassung

Die Dissertation untersucht Methoden zur topologischen Datenhaltung und beschäftigt sich mit der Konzeption und Umsetzung eines Toolkits für die Geowissenschaften zur Verwaltung von topologischen und zeitlichen 3D-Daten und Modellen in Geodatenbank-Architekturen. Die konzeptionelle Arbeit und die Realisierung berücksichtigt die Verwaltung hierarchischer Datenstrukturen, da diese als integraler Bestandteil der Geomodellierung angesehen werden. Die Arbeit legt den aktuellen Stand der Modellierungstools, Datenbankmanagement-Systeme und Methoden anhand anwendungsnaher Beispiele dar. Ferner wird ein Einblick in die Themen Modellintegration, Abstraktion von Geodaten und räumlich-zeitliche Modellierung, einschließlich umfangreicher Verweise auf grundlegende und zeitgenössische Literatur gegeben.

Des Weiteren wird gezeigt, dass geowissenschaftliche Daten durch die Heterogenität der Modelle und Anwendungen gekennzeichnet sind. Die Integration mehrerer Modelle und Anwendungen in gemeinsame Architekturen ist ein Ziel dieser Arbeit. Das Thema Multi-Representation wird durch die Erörterung der Konzepte der hierarchischen Digitalen Landschaftsmodelle, Progressive Mesh Representation, Multiple Topological Representations u.a. vertieft. Die räumlich-zeitliche Modellierung wird durch eine Diskussion über Konzepte wie TimeStep, Temporal Point Tube Model u.a. dargestellt.

Der Entwurf eines Topologie-Moduls zur Modellierung raum-zeitlicher topologischer Objekte wird im Detail erläutert. Das Konzept und die Implementierung des Moduls zur Modellierung und Handhabung der Topologie von Zellen komplexer Geo-Objekte wird vorgestellt. Da das Konzept nach Netz- und Objektebene differenziert, ermöglicht es die Modellierung „großer Zellen“, die wiederum eine beliebige Anzahl von Zellen umfassen können. Die Implementierung des Modells liefert ein Iterator-Framework, das eine flexible Navigation auf der topologischen Struktur ermöglicht. Bearbeitungsmethoden für die topologische Struktur werden vorgestellt und diskutiert. Das Konzept der Netz- und Objektebene ist eine solide Grundlage für die Verwaltung von Levels of Detail in Zellnetzkomponenten. Das raum-zeitliche Modell des Topologie-Moduls baut auf dem Temporal Joint Model von DB4GeO auf, einer service-basierten Geodatenbank-Architektur. Kontinuierliche geometrische Veränderungen können durch das Temporal Joint Modell verarbeitet werden, während diskrete topologische Änderungen der „großen Zellen“ durch das Raum-Zeit-Modell des Topologie-Moduls modelliert werden. Schließlich werden Programm-Laufzeiten und des Speicherverbrauch des Topologie-Moduls untersucht.

Hiermit erkläre ich, dass ich diese Arbeit, mit Ausnahme der hier vollständig und genau bezeichneten Hilfsmittel, selbständig verfasst und die Grundsätze des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

_____

Edgar Butwilowski

Karlsruhe, 9.4.2015

3D Data and Model Management for the Geosciences with Particular Emphasis on Topology and Time

PhD Thesis

at the Chair of Geoinformatics, Geodetic Institute
Department of Civil Engineering, Geo and Environmental Sciences of
Karlsruhe Institute of Technology

Prof. Dr. rer. nat. Martin Breunig
Prof. Dr. Mulhim Al-Doori

Edgar Butwilowski
Tel.: +49(0)721/608-42309
E-Mail: edgar.butwilowski@kit.edu

Table of Contents

List of Figures

List of Tables

List of Source Code Samples

Glossary

BMBF	Bundesministerium für Bildung und Forschung (German Ministry of Education and Research)
CAD	Computer Aided Design
cf.	compare
CGI	Computer Generated Imagery
DB	Database
DBMS	Database Management System
DEM	Digital Elevation Model
DFG	Deutsche Forschungsgemeinschaft (German Research Foundation)
DLM	Digital Landscape Model
E&P	Exploration and Production Industry
EEA	EEA Earth Energy Analytics & Development GmbH
et seq; et seqq.	and the following; and those that follow
EWKB/EWKT	Extended Well Known Byte/Extended Well Known Text
Geo-DBA	Geodatabase architecture
Geo-DBMS	Geodatabase management system
G-Map	Generalized Map
G-Map-DB	Database that contains G-Maps data
GI	Geoinformation
GIS	Geoinformationsystem
GML	Geography Markup Language
HTTP	Hyper Text Transfer Protocol
ID/id	Identification Number
Inv-ID	Identification Number of an Adjacent Cell-tuple
LHC	Largest Homogeneous Cell
LoD	Level of Detail
MRDB	Multiple Representation Database System
MTR	Multiple Topological Representation
OGC	Open Geospatial Consortium
OODBMS	Object-oriented Database Management System
OOM	Object-oriented modelling
PM	Progressive Mesh Representation
SIC	Laboratoire Signal, Image, Communications
SQL	Structured Query Language
STDBMS	Spatio-Temporal Database Management System
STO	Spatio-Temporal Object
VRML	Virtual Reality Modelling Language
WEDS	Winged-Edge Data Structure
XML	eXtensible Mark-up Language
XSLT	eXtensible Stylesheet Language Transformation

Directory of Utilized Software Tools

ArgoUML	Tool for the modelling of UML diagrams (Acquisition source: http://argouml.tigris.org/)
Autodesk® LandXplorer	3D city modelling software by Autodesk that can read and display CityGML XML format files.
db3dcore	The DB4GeO/DB3D Kernel is a completely in Java written library of 3D geometric data types (and its methods) and it even provides the possibility to build topologically defined nets (for example triangle nets) as well as access to a fast spatial index on the geometry data. (Acquisition source: http://github.com/geodb/db3dcore)
db3d	The db3d module extends db3dcore with complex operations, a temporal component, management of versions of geomodels etc.
Db3dRestModule	Db3dRestModule is the server module of DB4GeO/DB3D. It provides the functionality to expose geodata to a network.
DB4GeORestAdmin	Tool for the administration of DB4GeO/DB3D databases (Acquisition source: http://scc-bilbo.scc.kit.edu:8080/db4geotest/)
Eclipse EE IDE	Eclipse Enterprise Edition is an integrated development environment, used for the development and validation of Java programs and XML schemata. (Acquisition source: http://www.eclipse.org/downloads/)
Mirarco ParaViewGeo	Tool for visualization of geoscientific data, uses the Visualization Toolkit (VTK), based on ParaView (Acquisition source: http://paraviewgeo.mirarco.org/)
Paradigm™ Gocad	Software Suite for the visualization, creation and manipulation of geoscientific data (Acquisition source: http://www.pdgm.com/)
ESRI ArcGIS/ArcScene	A widely used Geoinformationsystem that is capable of visualisation and conversion of meshed surface models (Acquisition source: http://www.esri.com/jumppages/buttons/arcview_eval.html)
Microsoft Office Visio	Software is used for the development of various diagrams (Acquisition source: http://office.microsoft.com/de-de/visio/)
SIC moKa	The „Modeleur de Cartes“ is a CAD that has been developed at the Laboratoire SIC (Signal, Image, Communications) at the French Université de Poitiers. moKa internally exploits a graph based model of G-Maps (Acquisition source: http://www.sic.sp2mi.univ-poitiers.fr/moka/)
Sun Java SE SDK	Development package for the programming language Java (Acquisition source: http://java.sun.com/javase/downloads/)

1 Introduction and Related Work

1.1 Scope, Style, and Outline

The subject of this doctoral thesis is a direct outcome of the homonymous research project of the *German Research Foundation* (DFG) on which a small research group worked during the last years.[1] Since several years, research has been done in the *Geodatabases Working Group* at the University of Osnabrück and at the Chair of Geoinformatics in the Geodetic Institute of KIT on the technology of 3D data and model management for the geosciences. During these years, a prototype of a geodatabase architecture (Geo-DBA) known as DB4GeO/*DB3D*[2] has been developed (Bär 2007; Breunig, Schilberg, et al. 2009; Breunig, Butwilowski, Kuper, et al. 2013). My work bases on DB4GeO, and the results are reintegrated into the main development tracks.

As the title of the research project/thesis mentions, this work will focus on topological and temporal modelling. The DB4GeO Geo-DBA has some insufficiencies in these domains. The Working Group[3] already added much relevant work on this topic (Thomsen et al. 2008; Thomsen and Breunig 2007). I've already worked on this subject during my diploma thesis (Butwilowski 2007), so I'm glad to be able to continue this work on the development of my Ph.D. thesis. Now I aim to contribute to a more comprehensive model/architecture of DB4GeO. The employed work items will include i.a. the design of models for the

1 Grant no. BR 2128/9-1, official German title is “3D Daten- und Modellmanagement für die Geowissenschaften unter besonderer Berücksichtigung von Topologie und Zeit.” Group members were my colleagues Dipl.-Math. Andreas Thomsen, Dipl.-Geogr. M.Sc. Geoinf. Daria Golovko, and others.

2 DB4GeO is the newer external/marketing name; the older and still internally (in source code) used name is and will remain DB3D. In the following only DB4GeO is used as the term for DB4GeO/DB3D, though DB3D may appear in source code excerpts at times.

3 Especially A. Thomsen

management of topology of 3D/4D geological data, the development of suitable algorithms, runtime experiments and performance testing.

Generally, DB4GeO is developed with the ideal to be an “all-purpose” Geo-DBA at its core. This means that the core geometry model of DB4GeO has not been developed to serve any special application purpose but has been kept sufficiently generic to serve many different geometry modelling/computing applications (even those that are not common to the geosciences). Though, DB4GeO shall stay a multi-purpose Geo-DBA, it is important to keep in mind concrete application scenarios when enhancing the system with new functionalities in order to stay close to real-world requirements. As a consequence, the *EEA Earth Energy Analytics & Development GmbH*[4] was found as a partner of practice. The EEA is a company that plans to operate on deep geothermal energy projects and thus inherently has to be capable of managing and understanding a huge amount of *geoscientific data*.[5] As a company in a highly competitive market their focus lies on effectiveness and cost reduction of the applied processes and thus they are able to provide me with application requirements which I will mention especially in the introductory parts. The integrated concept of the EEA is subdivided into six modules (see Fig. 1).

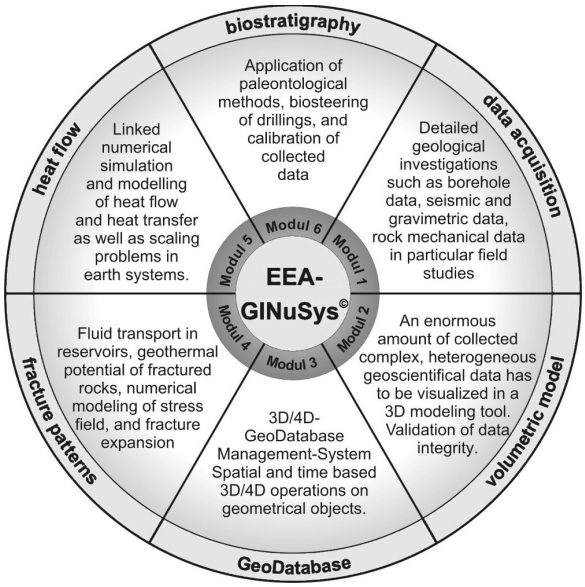


Fig. 1: Integrated concept GINuSys©
Source: (EEA GmbH 2012)

This service portfolio has been termed GINuSys© which is a system that employs knowledge from the scientific disciplines geology, computer science and numerical analysis. It is envisioned to deploy DB4GeO as the 3D/4D geodatabase management system for spatial and temporal (3D/4D) operations on geometric objects in module 3 of

4 Company website: http://beyondwind.net/

5 The essential question that such a business venture aims to answer is whether it is possible to produce enough hot water from a certain drill-hole to install an economically successful project. This set of problems is seen as the biggest barrier to investment in geothermal energy production (so-called Exploration Risk, see (Münchener Rück 2004, 44)).

GINuSys. As such it would be the central depository for the data exchange needs of the other modules.

This thesis consists of four main parts. It starts with a motivation for dealing with this topic; this is especially done by giving an inside view into the current application requirements of the geological modelling community, the relevant techniques that are used, the technical limitations of current data modelling and management systems and a number of use cases. This is followed by an overview of the relevant scientific literature in the field of geometric, topological, hierarchical, and temporal modelling for geoinformation systems. The third part describes details of the implementation of the spatio-temporal topology component that are developed in this dissertation. Finally the last part gives an outlook onto possible further workings/open issues.

The targeted audiences are spatial information technologists, especially those that are concerned with geological modelling, e.g. personnel of geological surveys or company employees in the *Exploration and Production* (E&P) industry, as well as GIS experts.

The style of writing generally follows the *Chicago Manual of Style* guide (University of Chicago Press 2010). The employed citation style follows the guidelines of the Chicago Manual of Style in the *author-date system*. If adopted figures, tables, content etc. from other authors were modified, then the indication of source are preluded by “cf.” Figuratively used terms are set in quotation marks only on the first appearance. In the then following sentences, they are “used normally”. Only when a longer segment lies between two such terms, quotes are used again. Upper/lower case spelling of adopted technical terms is aligned on the spelling in the respective source literature.

1.2 Motivation

1.2.1 Geological Modelling in Practice

Industries that typically benefit from geoinformation are defence and intelligence, business administration, education, government, health and human services, mapping and charting, utilities and communication, transportation, and public safety, as well as natural resources.[6] A subset of geoinformation is the information on geological features. The modelling of geological objects is performed especially in the various fields of the natural resources branch as in geomorphology, geophysics, nature conservancy or environmental management.

Geological models are used for education and in business consulting (whether done by national government agencies for geology or by private stakeholders). Geological models are a crucial factor for cost reduction in various geoscientific applications in the E&P industry, such as searching for oil, gas or geothermal fields. In E&P industry there generally is a high financial risk of finding the right subsurface spots of natural resources.

6 For a comprehensive list, see (ESRI 2010)

A better knowledge of the engineering and management of subsurface assets help to lower the uncertainty of E&P operations. Furthermore, geometric models are also essential to compute mechanical, hydrological or e.g. subsurface temperature models in order to make predictions about possible processes under the earth's surface.

To create a consistent geological model of an earth's crust sector, it is inevitable to “look below the surface of the earth” (i.e. to collect extensive data of the subsurface). There is a large number of methods to collect data for subsurface models most of which are based on the interpretation of rock exposures, remote sensing imaging, drillings[7] etc. Geophysical methods are applied in seismic measurements[8], electrical resistivity imaging (ERI) or gravimetry (Götze and Lahmeyer 1988).

Following the terminology of Schaeben et al. (2003, 174), there is a distinction between conventional *geographic data* that is modelled in “traditional” *Geographic Information Systems* (GIS), and geoscientific data, which is used in fields of the natural resources branch. Geoscientific data shows a higher complexity in terms of *dimensionality*, *mobility*, and *impreciseness*, as well as a greater *diversity* of its types. According to Schaeben, commercial GIS (that typically process 2-dimensional, static data) are not capable of dealing with geoscientific data adequately. Hence, geoscience specialists[9] that construct geomodels (e.g. geologists) use versatile specialized software suites[10] additionally to GIS to process subsurface data.

The subsurface data that is gathered through the above-mentioned collection methods (or measurements), is often of *considerable size*, it is *raw*, and it has to be refined, i.e. reduced and converted into a format that is processable by geomodelling software. Then the refined data is interpreted by the specialists with the help of geomodelling tools and transformed into a sensible model of the subsurface. The data formats that have to be managed by the specialists in day-to-day workload range from typical geoscientific data such as drill-hole files, point sets, geo-referenced remote sensing images, seismic images, Esri shapefiles, GOCAD ASCII files etc. to “common” data such as texts in various formats[11], pictures, videos etc. A typical geomodel that emerges as a final result of a modelling process is depicted in Fig. 2.

---

7 Here a lot of information/material is accumulated that can be used for interpretation such as borehole imaging, geoelectrical imaging or the drilling mud.

8 Seismic measurements are carried out by so-called geophones during (natural or man-made) seismic events and in principle base on the usage of the reflection and refraction of seismic waves for the interpretation of subsurface structures.

9 Simply “specialists” hereinafter

10 Some of the well known 3D modelling tools in the E&P industry are e.g. Paradigm GOCAD, Schlumberger PETREL, Halliburton Landmark, Seismic Micro-Technology KINGDOM or Maptek Vulcan.

11 E.g. office suite documents and Adobe Portable Document Format

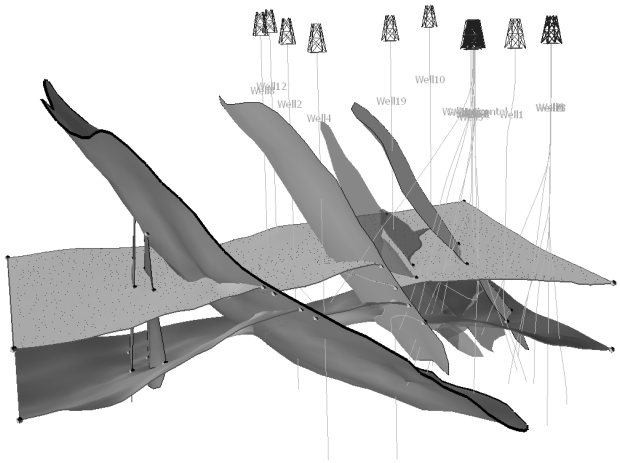


Fig. 2: Typical subsurface model with two strata boundary layers, seven fault layers and visualized well data Source:Visualization using GOCAD software, data from GOCAD workshop 2009

Typically, the modelled region is a delimited sector of the earth's crust with an extension of a few kilometres in width and depth (delimited by a *bounding box of interest*). Such models normally show the distribution of rock in the subsurface as well as the location and orientation of discontinuities in the rock. Usually, the strata solids themselves are not modelled but only their boundary and fault layers (as seen in Fig. 2 where two strata boundary and seven fault layers are depicted).[12]

In advanced projects the specialists produce volumetric strata models (in an additional step). An example of a volumetric data block is depicted in Fig. 3.

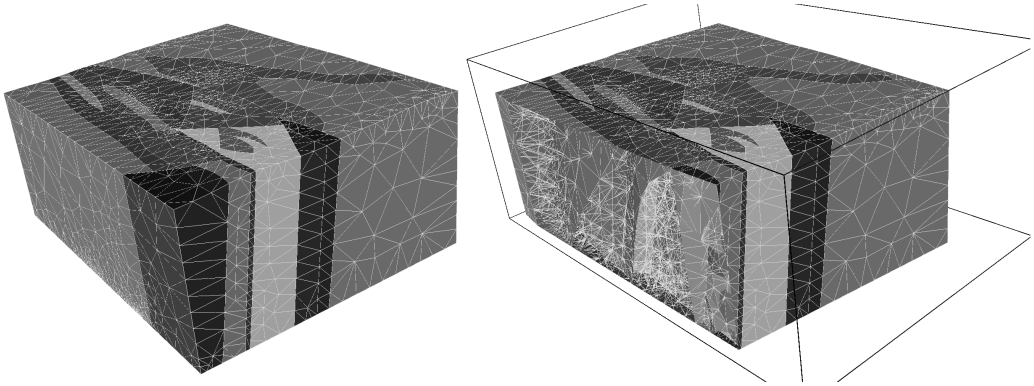


Fig. 3: Exterior of the Porcupine volumetric data block (left fig.) and a "look inside" the block (right fig.). Some layers have been removed in the foreground of right fig. to gain insight. Source: Visualization using GOCAD software, data by (Pouliot and Fallara 2007)

12 The geomodel in Fig. 2 also consists of well markers (point geometry data) and drilling trajectories (line segment geometry data); the boundary surfaces are internally structured as nets of triangles.

The figure shows a geomodel of the Porcupine-Destor zone, which is a one km deep earth solid under a ~ 45 km² wide area in the Abitibi subprovince of Quebec, Canada.[13] Albeit, it is not common yet to produce consistent volumetric strata models, since the construction of strata solids sets high demands on the quality of the refined data. But it is likely to become more common in the future as data quality and the tools' quality improve. Volumetric data is more intuitive and, what is more important, it provides the means for improved analytical processing of the data (see Ch. 2.1).

As a result of conversations with personnel of the *Landesamt für Bergbau, Energie und Geologie* (LBEG)[14] and the EEA, one of the more pressing problems of geological modelling in practice could be identified, which is the handling of large amounts of data files (of the models) that occur during the modelling process. These large amounts of data files emerge because often one geomodel is modelled by multiple specialists that work on their own files and additionally backup significant design steps in separate files. Furthermore, a geomodel can comprise changes of the model in time (also managed in separate files). And of course, most organizations have to manage multiple models in different spatial areas. The models may also be scattered over multiple organizational domains/boundaries. In addition, the data is kept in a proprietary data format, what often results in the necessity to acquire software only from one vendor (so called “data/vendor lock-in”).

1.2.2 Introducing DBMS for the Management of Geoscientific Data

The above mentioned insufficiencies in day-to-day work lead to a demand for a database (DB) server system that provides a remote access service for the management and synchronization of geologically modelled data in an organized fashion (so that teams are enabled to work on the geodata separated but in cooperation). A DB server acts as a central data hub in a client-server system. In such a setting, an operator who wants to contribute to a certain geological model connects his workstation to a central repository of geological models over the internet or an intranet, fetches the model he wants to alter and transmits the changes he made back to the central repository.[15] It can be assumed that more users and editors will gain access to information systems in the future, increasing the amount of data accesses and the diversification of structured and unstructured data that have to be managed. Under such conditions, DBMS become inevitable data integrators to keep control on high amounts of data.

According to Brinkhoff, a geodatabase management system (Geo-DBMS) has to fulfil the following requirements (Brinkhoff 2008, 25):

---

13 Dataset source is (Pouliot and Fallara 2007), the content of the figures is described later in more detail

14 The LBEG is the geological survey of the federal-state of Lower Saxony. The conversations took place during a visit at the LBEG offices in 2009.

15 A detailed description of such a typical application session with a particular focus on DB navigation and querying, using the predecessor of DB4GeO (which was GeoToolKit/Corba Adapter), is given by (Shumilov et al. 2002, 120 et seqq.)

- provide geometric datatypes
- provide geometric functions for the geometric datatypes
- provide spatial access methods and spatial indices
- provide interfaces for spatial data interoperability

Topology in Industry-Scale Geo-DBMS

Geo-DBMS are an established topic in the information technology industry. The most prevalent industry-scale Geo-DBMS are Oracle Spatial (Ravada and Sharma 1999), Microsoft SQL Server (Fang et al. 2008) and Refractions Research PostGIS (Blasby 2001). These Geo-DBMS are conceptionally similar since they internally use the object-relational model for geodata management. Furthermore, their geometry models are based on the *Simple Feature Model*[16] of the *Open Geospatial Consortium* (OGC®/OpenGIS®).[17]

The topological functionalities in these DBMS are still under conceptual development or in an early stage of implementation. The PostGIS and Oracle Spatial topology modules aim to implement the guidelines of the ISO SQL/MM part 3 ("Spatial") standard (ISO 2011). In the following, the topology model of PostGIS is presented. An entity relationship diagram of the conceptual PostGIS topology model is shown in Fig. 4.

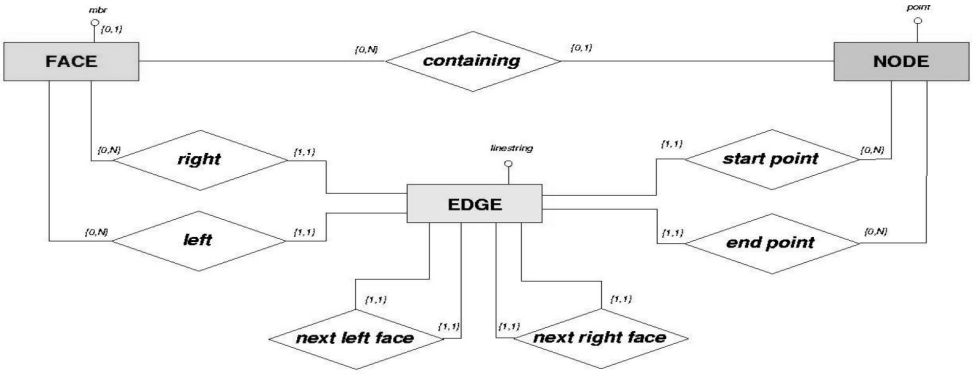


Fig. 4: Conceptual PostGIS topology model
Source: (Santilli 2011, 19)

The key element of the depicted topology model is the *EDGE*. Every edge is in relation to its *start point* and its *end point*, as well as to its *right* and *left face*, and to its *next left* and *next right face*. BRUGMAN (Brugman 2010, 17 et seq.) highlights that this model is equivalent to the *Winged Edge Data Structure* (WEDS) by BAUMGART (Baumgart 1975).

PICAVET points out the practical benefits of topologically enabled models in comparison to *spaghetti structures*[18] in PostGIS (Picavet 2010, 14 et seqq.). The primary aim of the

16 Which is specified in (Open Geospatial Consortium 2011).

17 For more information on simple feature support of the Geo-DBMS see (Patenge 2010) for Oracle Spatial and (Refractions Research 2012) for PostGIS.

18 Unstructured point sequences

development of a PostGIS topology module is to improve data quality. The topology module

- ensures that the occurrence of duplicate boundary elements is avoided wherever possible,
- it eases algorithmic navigation on parts of geo-objects, and
- reduces the need for geometric calculations.

However, the key assumption of WEDS is that an edge has always exactly two incident faces. This is only true in a 2D manifold setting, not for volumetric 3D geomodels.[19] Thus, WEDS and the PostGIS topology module only work in 2D application settings. Also the PostGIS topology module does not consider two and three dimensional net objects such as triangle nets and tetrahedral nets that are essential for the management of complex geoscientific objects as they were previously described.

PostgreSQL Topology Add-In for the Management of Geological Data

Therefore, members of the *Chair of Geoscience Mathematics and Informatics* of the *TU Bergakademie Freiberg* develop a net topology module for the PostgreSQL DBMS. It is capable of importing and managing geodata that has been created with Paradigm GOCAD. The geodata will be exposed through a *Web Feature Service* (WFS) in the data formats *Geography Markup Language* (GML) and *GeoScience Markup Language* (GeoSciML) (Gabriel, Gietzel, and Schaeben 2010).

The geometry kernel of the PostgreSQL net topology module of Gabriel, Gietzel, and Schaeben is termed *Geoscience spatial and temporal data* (GST). GST is capable of managing point sets, multiline sets, triangle nets, and tetrahedral nets (Gabriel et al. 2011, 4). Having a closer look at the geometry kernel, it becomes apparent that the kernel comprises runtime issues in certain application scenarios. The relational data model of GST for the management of *triangulated irregular network* (TIN) objects is shown in Fig. 5.

19 In a 3D setting, multiple faces may be aligned along an edge which leads to the Radial Edge Data Structure by Weiler (1988).

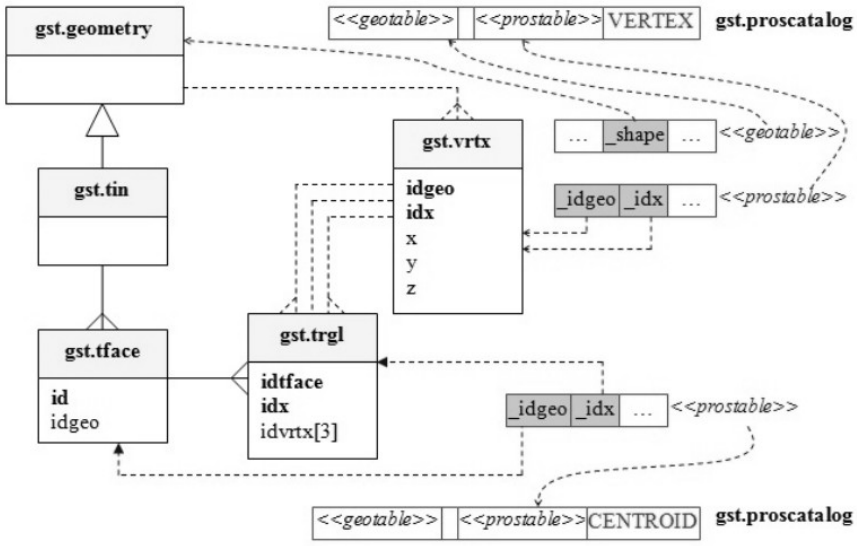


Fig. 5: TIN part of the relational data model of GST
Source: (Gabriel et al. 2011, 5)

In GST, a TIN (`gst.tin`) is modelled as a collection of faces (`gst.tface`) which in turn are modelled as a collection of triangles (`gst.trgl`). Every `gst.trgl` has a relation to three vertices (`gst.vrtx`). Also there are some back-relations from objects of a lower hierarchy level to the ones of a higher hierarchy level (like from every triangle to the face it belongs to). Though, this model is more powerful than the model of the PostGIS topology module, with such a model it still becomes complicated to navigate on the geometry net (e.g. to compute shortest paths between two nodes of a net), especially to navigate along the exterior of a net. For example, since node adjacencies are modelled only indirectly, to gain node adjacencies, it is necessary to traverse the structure from node to the triangles of the node and than from the triangles level back to the incident nodes. Similar inconveniences are expected for the tetrahedral net model.

DB4GeO for the Management of Geoscientific Data

The above-mentioned requirements stated by Brinkhoff are also fulfilled by the geodatabase management system DB4GeO[20]. DB4GeO provides 3D geometric data types, functions for these data types, and spatial access methods, as well as spatial indices in its core module, which is the *DB4GeO Core API*.[21] Some more complex operations (such as the *cross-section operation*) that are built as a composite of basic geometric functions, are gathered in a separate module which is termed just *db3d*[22]. This module also provides the means for project management, a basic handling of thematically defined data (Breunig,

20 DB4GeO is completely developed in Java.

21 The name of the development project of the DB4GeO/DB3D core API is db3dcore. db3dcore has been published as an opensource library under a GPL-like licence on the git hosting service “github”, available at the project's address http://github.com/geodb/db3dcore. For the sake of simplicity, the DB4GeO Core API is termed DB4GeO kernel or Core API or db3dcore hereinafter.

22 db3d - in contrast to db3dcore - is not published as open source.

Schilberg, et al. 2009, 49 te seq.), and an implementation of a model for spatio-temporal (4D) data, as well as the interfaces for spatial data interoperability through its I/O classes. DB4GeO exports the file types *DB3D X*ML (which is the proprietary interchange format of DB4GeO), GOCAD (VSet, PLine, TSurf, TSolid, SGrid[23]), GML, JML, VRML, and X3D and imports DB3D XML, GOCAD, JML, and Abaqus.[24]

Historically, DB4GeO is a successor to *GeoStore* and the *GeoToolKit* (Balovnev et al. 2004, 10 et seqq.). GeoStore is a Geo-DBMS that has mainly been developed in the 90s at the collaborative research centre SFB 350 of the University of Bonn. It has been developed with the C++ programming language and is an application-specific DBMS that focuses on the management of geological objects. Later, members of the research groups of SFB 350 realized that many of the geometric functions of GeoStore were useful in other application domains than geological modelling as well (Cremers et al. 2000, 4). Thus, a more generalised geometric kernel has been extracted from GeoStore and termed GeoToolKit. GeoToolKit is a library of geometric data types, functions, and spatial access methods. The foundations of the geometry model that was applied in GeoToolKit have been described in detail in (Breunig 2001). GeoStore itself has been redesigned to use GeoToolKit as its kernel for geometric computation. A custom desktop client has been developed for GeoStore that was able to load and visualize geodata from a GeoStore DB. In the following, other application modules such as GeoWeb (for the access on geological models via the web) have also been developed on top of the GeoToolKit library. GeoWeb had already been extended with such advanced functionalities as mesh decimation for a progressive transmission of the geodata to lower the usage of bandwidth for data transmission (Shumilov et al. 2002, 117). Also the Geo-DBMS had a spatio-temporal model and was able to process time-dependent queries (Siebeck 2003, 41 et seqq.). The resulting static or temporal geo-objects were visualized in a VRML browser that was extended by the Java applet Cortona™ (Shumilov et al. 2002, 116). After the 90s, the project “Advancement of Geoservices”[25] gave project team member Wolfgang Bär the chance to take GeoToolKit and all the lessons learned from its development as the inspirational basis to design and implement the next stage of the software in Java programming language (Bär 2007); it was termed DB4GeO.

The architecture of the network interface of the DB4GeO server changed several times during its development stages: in the first approach, when DB4GeO was developed in C++, the geometry objects where exposed to the net by the *remote invocation framework*

23 Support for SGrid (Stratigraphic Grids; also unstructured, irregular grids or “non-uniform” structured grids) is very limited

24 An exhaustive description of the I/O interfaces of DB4GeO can be found in (Rolfs 2005, 44). The supported file formats for import/export show that one of the main aims of DB4GeO is to bridge the gap between geoscientific IS and geographic IS.

25 The project with the German title “Weiterentwicklung von Geodiensten” was part of the special program GEOTECHNOLOGIEN of the Federal Ministry of Education and Research (BMBF) and the DFG. Detailed information on the project can be found on http://www.planeterde.de.

CORBA[26]. Later, when DB4GeO was ported to Java, the service architecture was changed to the distributed systems network architecture *Jini* (and remote invocation framework changed to RMI[27]). In the last step, the service architecture has been changed to RESTful HTTP.

The DB4GeO REST module[28] allows to expose the main functionality of the DB4GeO DBMS through a network as a service. The module implements the geodatabase server as a RESTful HTTP web server, which means that the operations of the DB4GeO DBMS can simply be accessed through URL requests, using the Representational State Transfer[29] (REST) architectural style and REST verbs – for call examples see (Breunig, Broscheit, et al. 2009, 104).

The RESTful interface supports the objective of DB4GeO to operate as a distributed spatial DBMS (or even more pointedly as a distributed GIS) to provide data selection, retrieval and operations on complex large scale geoscientific models. DB4GeO with the RESTful interface eases the distribution of geoscientific data across multiple virtual and real servers. Since RESTful HTTP is stateless, individual instances of DB4GeO servers do not have to synchronize states (cf. (Fielding and Taylor 2002, 119)).

DB4GeO has already been deployed in several research-centered application scenarios like for the early warning of landslides (Breunig et al. 2010, 84 et seq.), and it is intended to use DB4GeO as the central part of a distributed data system for the geosciences where data from different geoscientific fields shall be integrated into a network-based, metadata-driven system (Breunig et al. 2011, 15 et seq.).

Some application scenarios for DB4GeO have also been determined by the industry. For example, the EEA has identified some functionalities of a geodatabase as vital for their operational business. These were e.g. a “discrete access on geo-objects like aquifer, faults, and temporally moving crevasses, and the storage, access and export of virtual (i.e. computed) 2D profiles of 3D volume models, as well as the possibility for spatio-temporal querying and geometric 3D/4D operations on volume models” (EEA GmbH 2012).

Typical queries and editing operations on DB4GeO (and spatio-temporal databases in general) in application scenarios might be:

- Which stratigraphic horizons and faults are penetrated on a given drilling path (geometric query)?
- What is the volume size of a given block of rock (geometric query)?
- What are the hydrological parameters of the rock in a given spot (parameter/thematic query)?

26 Common Object Request Broker Architecture

27 Remote Method Invocation (Java's type of Remote Procedure Call)

28 The internal name of the service infrastructure project is RestDb3dModule.

29 For more information on REST architecture, see (Fielding and Taylor 2002)

- Remove a certain component of a horizon surface from the whole geo-object (editing operation).
- Did a given fault ever cut through a given rock solid (temporal geometric query)?
- What was the average speed of a given moving crevasse during the last two decades (temporal geometric query)?

Due to the limitations of its kernel geo-object model, DB4GeO is not able or has performance issues (i.e. is unacceptably slow) at responding to queries such as:

- What is the boundary polyline of a given fault?
- Show the polyhedral boundary surface of a rock solid.
- What is the shortest path between two drilling points on the mesh of a given stratigraphic horizon surface?
- On a digital geologic profile section: which is the stratigraphic layer on top of the given stratigraphic layer?

All of these queries require a powerful underlying topological model for geo-objects in the Geo-DBA kernel.

Hitherto, the implementation state of the topology model of the DB4GeO kernel limited its capabilities for the integration of heterogeneous spatial data models. DB4GeO is specialized on the management of Simplicial Complexes like triangle and tetrahedral nets. On the other hand, “classical/on-surface” GIS data utilizes a more unstructured approach to geometric modelling – e.g. *cadastre parcels* are typically modelled as polygons with an arbitrary number of support vertices.

1.3 A Generalized Geo-Model for the Integrated Modelling of Geoscientific and GIS Data

The integration of geoscientific subsurface data and “on-surface” (cultural) GIS data, such as 2D cadastre parcel features or 3D city models, brings forth valuable applications, cf. (Krämer et al. 2010; IAI (Institut für Angewandte Informatik Karlsruhe) 2011). An example of such an application is the integrated analysis of data of a geothermal field (subsurface) and 3D city model data (cultural) to study the profitability of a geothermal project, taking into account numbers, volumes and locations of potential customers' buildings as well as considering the costs of competing teleheating pipe pathways and the productivity of geothermal fields. A (simulated) visualisation example of a setting with two subsurface stratigraphic horizon boundaries, some well paths, a *digital terrain model* (DTM), combined with on-surface features like trees, drilling rigs and a city is shown on the left side of Fig. 6.

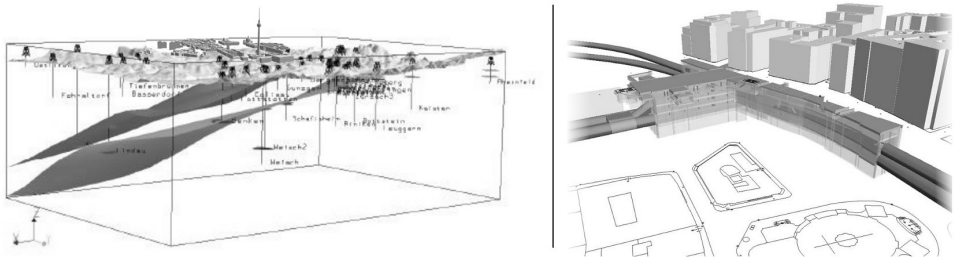


Fig. 6: Left: Subsurface geodata combined with 3D city model (collage/not related to reality); Right: Subsurface infrastructure planning
Source: left: collage from the figures of (Andenmatten and Kohl 2002) and City of Berlin dataset screenshot (© City of Berlin; citygml.org); right: (Dorffner, Ludwig, and Forkert 2006)

Another major application scenario for the combined management and analysis of geoscientific data and cultural geodata is the planning of subsurface infrastructure, which is depicted on the right side of Fig. 6, where a 3D city model and the model of a planned subway track are shown. In such planning scenarios, it is useful to seamlessly integrate geological subsurface data into the planning process in order to investigate the subsurface stability on the basis of rock material properties directly in one common system.

A well-known data exchange format for 3D city models is *CityGML*, which has been introduced by KOLBE and GRÖGER (Kolbe and Gröger 2003). CityGML since has become an OGC standard (Open Geospatial Consortium 2008). It is likely that the standard will be supported by several visualization clients in the future (cf. Fig. 7 for an example of a CityGML 3D city model, rendered with *Autodesk® LandXplorer*).

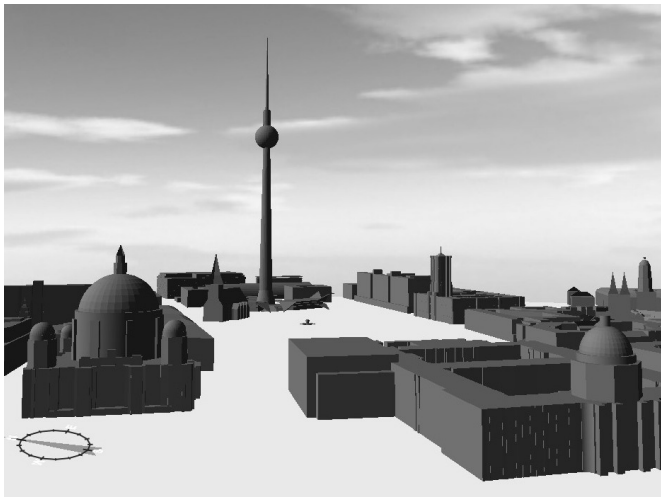


Fig. 7: CityGML model of Alexanderplatz, Berlin
Source: Autodesk® LandXplorer screenshot; data: © City of Berlin; citygml.org

The fundamental standard for CityGML and other OGC as well as ISO/TC211 standards is *ISO 19107* "Geographic information - Spatial schema" (ISO 2003). The geometry/topology model laid out in ISO 19107 specifies that OGC-compliant GIS/Geo-DBMS may support both, the modelling of geo-objects with vaulted, polyhedral surfaces (class

`GM_PolyhedralSurface`, constructed e.g. as triangle nets) as well as with arbitrary planar polygons (class `GM_Polygon`) in *boundary representation* (B-Rep) (Andrae 2008, 117 et seqq.).

CityGML uses B-Rep for the modelling of man-made objects. CityGML buildings for example are modelled as sets of polygons that define the buildings' boundaries. The B-Rep of CityGML requires bounding lines of building walls to be straight lines and bounding surfaces of building solids to be planar polygons (Open Geospatial Consortium 2008, 23). CityGML also explicitly encourages the usage of topological data structures, for example to model the joint use of geometries.[30] To efficiently handle the B-Rep of CityGML, a geometry/topology module has to be able to model topologically connected geometries with arbitrary flat boundary shapes (polygons) and to efficiently retrieve boundary geometries of geo-objects (e.g. the bounding polygons of a solid or the bounding line segments of a polygon). Furthermore, efficient topological navigation between CityGML geometry entities is needed (e.g. navigation from one building solid to its adjacent building). These requirements hitherto were not satisfactorily met by the DB4GeO kernel. This is because the concept of B-Rep is not directly consistent with the Simplicial Complex model of the DB4GeO kernel (more on this in Ch. 2.1). To meet these requirements, the geometry kernel needs to be extended by a topology module that is capable of handling both, objects that are described by net representation, and objects that are described by B-Rep, in one model.

However, the focus of this work will not be to design and implement data exchange interfaces in DB4GeO for CityGML (or OGC geodata models in general). Nonetheless, the topology module for DB4GeO shall enable the DBMS to internally handle such data. On top of the topology module, specialized models and import/export interfaces for CityGML data may be implemented (see Ch. 5.2.1 for more details).

1.4 Abstraction of Geodata

One of the key techniques of the human mind to make reality intellectually graspable is *abstraction*. Humans abstract continuously in various situations. This is a necessary consequence out of the fact that reality is too complex to be recognized in its entirety. Thus, the human mind generates models of reduced complexity of the reality in order to process it. This reasoning is fundamental, especially to the art of cartography and to geoinformatics in general. Cartography and geoinformatics deal with the creation of world models (2D maps or 3D representations) that are better graspable if provided on multiple levels of abstraction.

However, multi-scale modelling is not only an important concept for the human mind, but equally important for machine based processing. In fact, multi-scale modelling also is a fundamental subject in general computer science. The reason for the wide dissemination of

30 CityGML topology is formulated with XLinks. An examination of the topic of modelling CityGML topology with XLinks can be found in (Krimmelbein 2011).

the topic is due to the fact that in various problem definitions of computer science, it often is useful for performance reasons to initially compute an approximate solution on a coarse level, and then to seek more accurate solutions in levels of increasing detail. From this perspective, even search trees like *B-trees* (Bayer and McCreight 1972) can also be regarded as methods of multi-scale modelling.

In geoinformatics and cartography literature, there are several terms for the concept of abstraction with a similar meaning, as e.g. *generalization*, *multiple representation*, *multi resolution*, *hierarchy management*, *plurality of scales*, or *levels of detail* (LoD). With regard to some terms in this field, there is a confusion of tongues. In particular, the usage of *higher* and *lower* LoD or *large* and *small* scale of a map is often ambiguous/inverse. To avoid such confusion in this work, these terms shall explicitly be used in the following way: a *higher* LoD e.g. of a city model means that there is *more detail* in such a representation. For example: at a *lower* LoD, a city model may be missing all windows in the buildings, whereas these are added in a *higher* level. A similar concept applies for map scales: a large scale means that a map is very much “zoomed in” (much detail), a small scale “zooms out” (less detail).

Due to the broadness of the topic, a definition of a clean taxonomy of all concepts of hierarchy management is difficult. In geoinformatics, the topic of abstraction focuses on geometric, attributive, and topological abstraction. In geometric modelling, there are methods that allow for a continuous change of detail of a geo-object, like the *progressive mesh* method (which is explained in Ch. 2.4). However, in many applications of hierarchy management, it is useful to formulate certain definitions of fixed level of detail. These level definitions can only be formulated depending on certain applications. The definition of a certain detail level includes which types of geo-objects are assigned to that level. The assignment of a geo-object type to a certain LoD typically not only depends on the average size of the geo-objects of that type but also on attributive data. For example: in small scale waterways mapping, only features of large size, like a broad channel, are of interest. However, although waterway signs would be classified as to small in size on that LoD, they still would be included since they thematically are of special interest in this context.

1.4.1 Early Research on Abstraction of Geodata

Research efforts in machine based generalization were already conducted as early as in the 70s, e.g. by David William Rhind (1973). The research of Rhind focused on abstraction of geometry objects with fixed detail level definitions. Later, a greater research initiative called “Multiple Representations” on *multiple representation databases* (MRDB) was carried out by the National Center for Geographic Information and Analysis (NCGIA) in 1988. The closing report of the initiative (Buttenfield 1993) comprises an extensive collection of literature on the topic, some of which is discussed in the theory-part of this work.

MRDB are defined as spatial databases that are capable of maintaining several geometric representations with different levels of detail of the same real-world phenomenon.

Research on MRDB for a great part refers to the management and provision of 2-dimensional maps on multiple LoD (cf. Fig. 8).

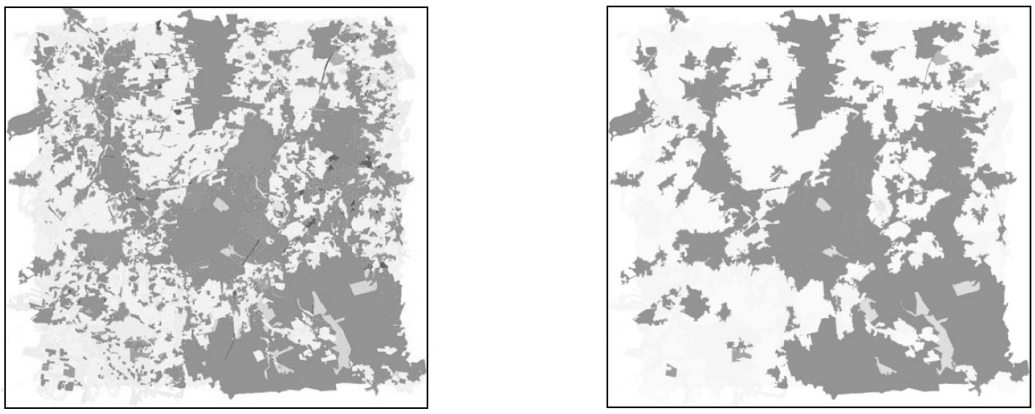


Fig. 8: Digital landscape model two different scales (left higher scale, right lower scale) Source: (Haunert and Sester 2005, 14)

Fig. 8 shows a map excerpt from an MRDB on landuse at three different scales. The linestrings depict boundaries of areas of different landuse. From left to right, the average sizes of the landuse areas are increasing. With changes of the area sizes, also the assigned land use classes change from more detailed to more general. Thus, any more generally classified landuse area can be defined as a collection of several more precisely classified landuse areas.

Many traditional paper maps already are published in multiple scales. In the design process of the maps, the lower scales have to manually be derived from maps of larger scale by techniques of generalization. While the generalization process for paper maps is solely targeted to the visualisation, one of the major specific tasks of an MRDB is also the management of the relations between multiple representations of a spatial object on different scales. This allows for cross-scale navigation and cross-scale analysis on geodata.

1.4.2 Difficulties in Automated Geodata Abstraction

Hierarchy management of geodata is a broad topic. The cartographic generalisation processes cannot easily be structured. For example, in some applications of hierarchy management, there is a demand that a geo-object even has to change its dimensionality on different LoD. This can be the case e.g. for a digital city model, where the city can be modelled as a point feature at small map scales, as a planar (polygonal) feature at medium map scales and as a volumetric buildings model at large map scales. Such cases make it difficult to find correlations between the different LoD and complicate a sound automated algorithmic treatment.[31]

31 A similar issue is found in multi-scale modelling of CityGML (Open Geospatial Consortium 2008, 9). CityGML supports the management of up to five LoD. Although it is possible e.g. to navigate between the LoD of a single whole building, it is not possible to navigate on parts of a building (e.g. a building wall) between their multiple versions on different LoD. In CityGML only a reduced set of relations is modelled between the LoD.

It is obvious that in such cases it is not only difficult to automatically find geometric interrelations between different LoD but that also issues arise when a geomodel is changed, i.e. when the geomodel is edited/updated by a specialist. To stick with the city model example given above, a multiple representation system has to incorporate rules for the adjustment of a lower LoD to changes in a higher LoD. An example would be the extension of a city model at its boundaries by the 3D building model of a spacious building complex on highest LoD. In such a case, an automatic update of all lower LoD might be necessary. On the polygon level, the base area polygon of the city might need to be expanded. On the point level, the representative point of the city (lowest LoD) might need to be displaced.

However, the adjustments are generally vague, since it depends on the application scenario and purpose, which rules apply. For example, there might be no need to displace the representative point if it is defined in a different way than as the *centre of gravity* of the urban extent, or if other rules interfere.

Often, automatic updates are not possible in the opposite direction at all. This can also clearly be shown by the above example: if the operator first manually displaces the representative point (due to the construction of a new building complex), the system of course cannot create a detailed building model by itself on a higher LoD. This is a general problem: on geometric modifications at a lower LoD, obviously the missing information of a higher LoD mostly cannot be synthesized automatically.[32]

The examples show that abstraction rules/mechanisms mostly have to be defined for certain narrow application cases and are not generally applicable. Hierarchy management systems for geodata are always restricted to one certain application or to a narrow application field. Thus, in industry and research, there are approaches that are specialized on clearly defined applications, or at least that strongly limit the application radius so that it is possible to develop applicable rules and systems that are valid in their respective field.

An example of a widely used hierarchy management system with a clearly defined narrow field of application is ATKIS[33] (Anders and Bobrich 2004), which is an MRDB of multiple *digital landscape models* (DLM) with a fixed amount of *four* LoD. ATKIS is a solely geometric MRDB, where the generalization information is maintained by links between the geo-objects of a larger scale and their representative of a lower scale.

This chapter explained the importance of the abstraction subject area in geoinformatics and outlined some general concepts and issues of geometric abstraction. However, in Ch. 2.4 not only the introduced reasoning further will be further elaborated (i.e. specific methods of geometric hierarchy management in 2D and 3D applications that are state of research are discussed) but also a shift in focus will lead to topological modelling of multiple representations.

---

32 Though, its not impossible in all cases. Synthesization can be achieved through the application of comprehensive sets of rules.

33 German Authoritative Topographic-Cartographic Information System

1.5 Spatio-Temporal Geodata

Spatio-temporal geodata is data that is defined with respect to space and time.[34] In several application domains, the extension of spatial data with a temporal component enables many additional useful types of analyses that static geodata alone cannot provide. Consequently, industries start adopting temporal geodata in their day-to-day business. In fact, temporal geodata can be employed virtually anywhere wherever geodata is already in use, e.g. in business management, social sciences, environmental research, geoscientific modelling, or city planning. For example, in the environmental sciences, spatio-temporal analysis of daily rainfall data that is measured at multiple locations, helps to understand the processes of climate change.

1.5.1 An Example of Spatio-Temporal Modelling in City Planning

City planning places particularly high demands on the capabilities of spatio-temporal models. Example cases of the application of spatio-temporal data in city planing are illustrated in Fig. 9 and Fig. 10.

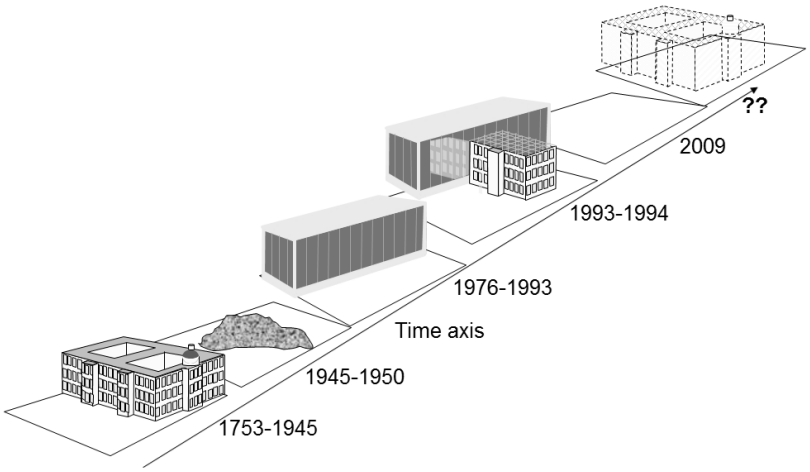


Fig. 9: Application example for the evolution of a building site over time (Berlin City Palace/Palace of the Republic), source: presentation of Andreas Thomsen at DFG project "Abstraction of GeoInformation" meeting in Gengenbach 2008

Fig. 9 shows state changes of a building (here: Palace of the Republic) over its lifetime (from 1753 to the future). A spatio-temporal DBMS that is capable of managing the chronological sequence of a building needs to handle various states and transitions. In the example, the Berlin City Palace was constructed in 1753. In 1945 the building was completely destroyed. Only debris remained on the lot for five years. The debris was

34 Definition according to the entry "Spatio-temporale Daten" in the GI-Lexikon of the Geoinformatik-Service of the University of Rostock (entry link: http://www.geoinformatik.uni-rostock.de/einzel.asp?ID=1981)

removed and the successor to the Berlin City Palace, the Palace of the Republic was build on the same lot in 1976. The Palace of the Republic remained externally unaltered for almost two decades, until it was constructionally extended in 1993. In 2009 the Palace of the Republic has completely been removed, and only an empty lot remained. At present, a reconstruction of the Berlin City Palace is under way.

This kind of complex setting is a typical case that occurs in urban planning. In the presented setting, multiple different types of temporal changes take effect. The schematic drawing of Fig. 10 more clearly delineates, abstracts and classifies the different change types.

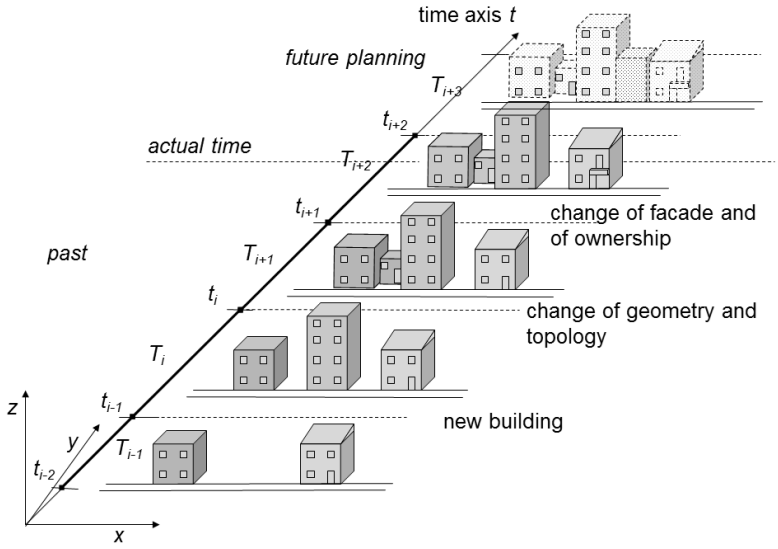


Fig. 10: Application example for geometric, attributive and topological changes of a 3D city model, source: presentation of Andreas Thomsen at DFG project "Abstraction of GeoInformation" meeting in Gengenbach 2008

In the example of Fig. 10, at time step T_{i-1}, the model consists of two disjoint buildings. A new building is added to the model at time step T_i. This changes the overall geometry of the model but not the geometries of the existing buildings. At time step T_{i+1} another building is added to the model that connects two of the existing buildings. This not only changes the overall geometry of the model but also the geometry of the two existing buildings that are affected by the constructional expansion. In the same temporal step, the overall topology also changes, since the two buildings, which were disjoint before, now get connected through the newly introduced central block. In time step T_{i+2} (which is marked as *current time*), there are three different types of changes that take place at the same time step. There is a change of ownership, which is an attributive change. This is not a geometric change, it only changes the character sequence entry of the ownership property of the building. Furthermore, in the same step, the facade of a building is expanded by a balcony. This induces a change of the building's own geometry and topology. In the future, the construction of a fourth building is planned for time step T_{i+3}, but this is only one

possible version of what really might be implemented when the time step becomes the present.

This was only one example of an application domain and its adoption of the temporal component. With an increasing number of application domains that need to adopt temporal geodata, also the claim to science to elaborate sophisticated spatio-temporal models and a clean structuring of the processes and problems increases.

1.5.2 Early Research on Spatio-Temporal Modelling and its Objectives

The management and storage of spatio-temporal data is a strong observed field of research in the geoinformation sciences already for several years. In 1990, LESTER compiled a comprehensive overview of (still mostly valid) research problems in spatio-temporal modelling, cf. (Lester 1990), some of which are:[35]

- the understanding of time,
- temporal logic,
- architecture of temporal GIS, and
- how to deal with alternative representations.

In this context, this thesis is primarily concerned with the issue of the *architecture of temporal GIS*.

SHOHAM and GOYAL identify four different *reasoning tasks* that can be supported by temporal GIS[36], which are: *prediction*, *explanation*, *learning new rules*, and *planning* (Shoham and Goyal 1988, 419 et seq.). Whereas *prediction* uses a set of existing rules and a model of the present state to predict a future state, *explanation* uses the set of rules to explain a former state from the present. In the task of *learning new rules*, two recorded states of two different points in time are used to deduce the affecting rules. In *planning*, a model of a present state, a model of a desired future state, and a set of rules are used to deduce activity guidelines to achieve that state.

At first glance, the extension of a GIS to a *time-integrative GIS* (Ott and Swiaczny 2001) seems straightforward, by introducing an additional time variable to the already existing spatial variables. But the problem is more delicate, since a temporal information system needs to hold and arrange copies of old recorded states (its *versions*). Thereby, questions concerning the level of *version tracking* arise.[37] Topologically regarded, similar rules apply for time as for space (Langran and Chrisman 1988): neighbourhoods can be modelled explicitly, with similar benefits as there are for the explicit modelling of topology in space.

---

35 This chapter presents some basic ideas of the topic in order to provide a rough overview. However, not all presented aspects will further be covered in the thesis.

36 Real-life application examples for the usage of temporal GIS are be given by (Worboys, M.-F. 1994).

37 The topics of temporal data and version control are closely related and should be considered conjointly.

1.5.3 Basic Considerations on Spatio-Temporal Modelling

WACHOWICZ describes two traditional, fundamental approaches regarding version control in temporal GIS (Wachowicz 1999, 3 et seqq.): The idea of "organising space over time" is closely related to the layer concept of GIS. Here, every step in time renders a completely new data layer (Fig. 11).

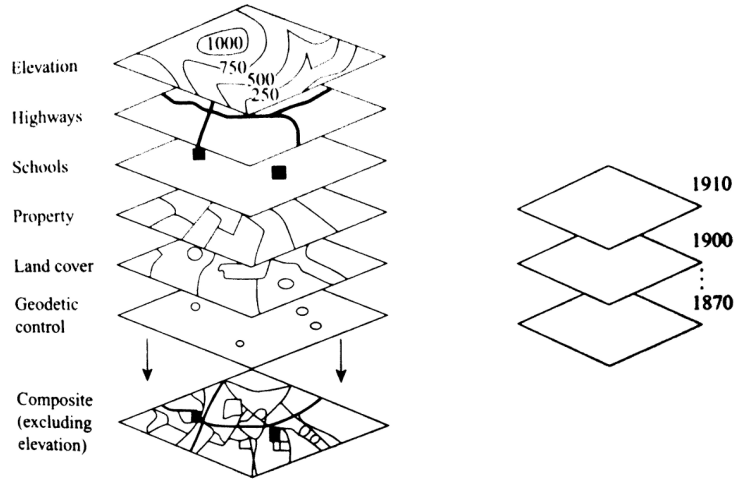


Fig. 11: Layer concept in GIS (left) and spatio-temporal layers (right)
Source: (Wachowicz 1999, 4)

On the other side, the idea of "representing a real-world phenomenon in space and time" (ibid.) means to append time to the spatial unit that is indivisible in the respective application and to perform version control on that element. This approach is more closely related to the object-oriented perspective. An indivisible unit can be – depending on the application – a whole geo-object (*object level*) or parts of the object, e.g. an object attribute (*attribute level*).[38] Version changes may imply a change of an attribute value, as well as changes in the object's geometry, or in its topological configuration (more on this topic in Ch. 2.5), or even in the schema of the object.

In temporal modelling of geo-objects, *version control* plays an important role. Version control aims at the *incremental update* of geo-objects or their parts. With incremental updates, versions are calculated by a combination/addition of other available versions of a geo-object. This raises new issues that can be summarized under the "space vs. runtime" problem field. DADAM et al. detected two types of strategies for incremental update (Dadam, Lum, and Werner 1984), which are *forward oriented versioning* (*non-accumulative* and *accumulative*), and *backward oriented versioning* (*non-accumulative* and *accumulative*). In *forward oriented versioning*, the oldest object that has been added to the data set, is taken as the basis on which the newer versions of the object are created. Newer objects are thus constructed on basis of the older objects by only recording the changes to the older objects. In contrast, in *backward oriented versioning*, the object that was added

38 Generally, the consideration of time aspects under the object-oriented perspective seems to be particularly fruitful. This has especially been pointed out by Wachowitz and others active in the field.

last, is taken as the basis on which the older versions of the object are redefined. By the *non-accumulative* strategy, the derivative versions of an object are always deduced from the one base version (oldest/youngest object), whereas by the *accumulative strategy*, each derivative version is incrementally deduced from its predecessor in the version history. Each approach has its advantages and disadvantages in terms of runtime and memory usage in certain applications. In research on spatio-temporal models, a recurring issue is which parts of a geo-object shall transfer from an existing to a new version, respectively what proportion of an object shall become part of its copy. Such problems are closely related to the question, in which cases an old object ceases to exist and when a new one begins.

In the attempt to capture temporal information in databases, it soon turned out that it is useful to categorize the recorded time into two types. SNODGRAS and AHN framed various approaches that were developed at that time by several research groups, under the terms *valid time* and *transaction time* (Snodgrass and Ahn 1985). While valid time designates the time in which an event occurs in the reality, transaction time indicates the time instant at which the same event is recorded in the database. Based on this classification, SNODGRAS and AHN distinguish four kinds of (chronological) databases, which are: *snapshot*, *rollback*, *historical* and *temporal databases*. While *snapshot databases* do not support any concept of time (do not store temporal data), *rollback databases* are capable of storing several versions of a data unit along with its *transaction time*. With the help of this meta information, previous states of the database can be restored (*rollback database*). In contrast, *historical databases* have no rollback functionality. Instead they include *valid time* to each data unit version. By this, it becomes possible to consider the real temporal evolution of an observed item. Finally, a *temporal database* supports *valid time* and *transaction time* and thus makes it possible to study the historical evolution of an object as it happened in (or was planned for) reality, as well as the evolution of its representation in the database (combination of *rollback* and *historical database*).

Furthermore, *valid time* can be modelled in a *longitudinal configuration* or in a *branching configuration* (Lester 1990, 12) (see Fig. 12).

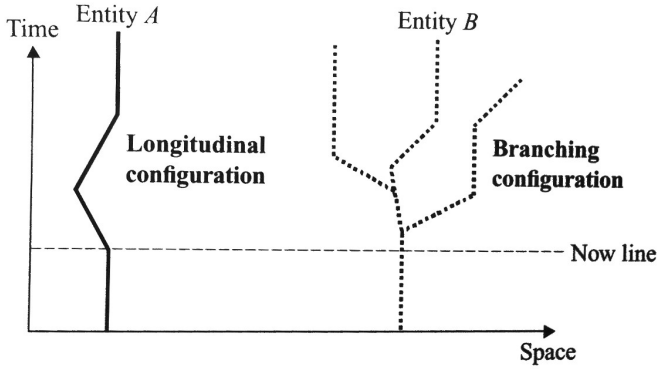


Fig. 12: Longitudinal and branching time
Source: (Wachowicz 1999, 21)

The figure shows the *time dimension* plotted against one *space dimension*. The space dimension in this figure is a representation of several space dimensions (for the sake of simplicity) and symbolizes the physical movement of a geo-object through space. In the *longitudinal configuration* (left), entity A takes only one path through time, so that at any time instant, there is always exactly one version of the object. In the *branching configuration* (right), first there is also exactly one version of entity B in the example, but only until "present time" (cf. "Now line"). However, after this, it splits into multiple simultaneously existing versions. Thus, at specific times, there are several versions of entity B that differ in their attributive and/or spatial configuration (position, shape).[39]

In the 70s, HÄGERSTRAND proposed the usage of space-time trajectories of spatial objects for problem solving in the geographic information domain (Hägerstrand 1975).[40] Space-time paths describe the lines that a geo-object generates in space-time over a given period through its spatial movement. Each spatial feature moves on its own spatio-temporal path (see Fig. 13).

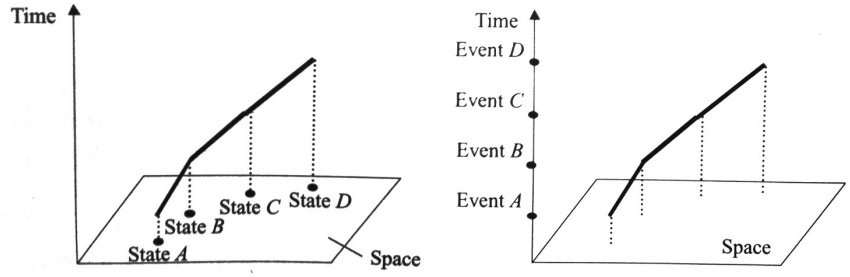


Fig. 13: States (left) and events (right) of a geo-object during its lifetime
Source: (Wachowicz 1999, 22 et seq.)

During its lifetime on the space-time path, a geo-object can repeatedly change the characteristics of its movement or of its other entity properties. Such time instants can be considered as *states* (see left illustration in Fig. 13). States are triggered by *events* in time (Fig. 13, right). An event can be for example an update process on the data. In information systems, events can typically be used to store explicit versions of the geo-object (more on this in Ch. 2.5.2).

The periods of time between states can be described by different approaches, such as by *kinematic* or *dynamic* descriptions. Kinematic models describe the change of an object by comparably simple rules that only consider change of the external appearance of the object.[41] Dynamic models instead describe the change of the external appearance of an object as a result of physical forces acting on the object.

---

39 A well-crafted application example of 5D models that combines both, valid branching time as well as transaction time in a realistic spatial database query, can be found in (Schaeben et al. 2003, 178).

40 Hägerstrand presented his ideas on the basis of an application-oriented example concerning the human behaviour in urban surroundings.

41 (Alms et al. 1998, 255 et seqq.) delineates the architecture of a class library that facilitates the modelling of change of the external appearance of geological objects.

Based on these elementary considerations of spatio-temporal reasoning, several models for the handling of spatio-temporal data have been developed in the last decades. As we will see, the models focus on specific application requirements. Temporal geodata becomes increasingly valuable in a growing number of application fields. For example, temporal geoscientific data is recorded when an oil reservoir is tapped. As long as the oil field is in production, the reservoir is monitored and temporal geometry models of the reservoir solid are constantly generated. The temporal models are needed to detect changes in the reservoir volume early in order to be able to react properly in time. Another application field where temporal geodata becomes increasingly useful is city planning, where temporal geodata can help to better understand the mechanics of the city's evolution.

Applications with temporal geodata put high demands on the underlying temporal geometry model. Chapter 2.5 will present well-known spatio-temporal models and show that they have deficiencies in modelling topology that changes in time. The extension of the DB4GeO topology module with the capability of handling temporal data (Ch. 3.6) can contribute to an advanced usage of topology in temporal data.

1.6 Remarks on Suitable Spatio-Temporal Testdata

Spatio-temporal 2.5D and 3D data are still rare. A remarkable spatio-temporal dataset (Lautenbach and Berlekamp 2002) that has been compiled and developed by LAUTENBACH in his diploma thesis, could be obtained. It is used in tests of concept implementations in this thesis (two snapshots of the dataset are illustrated in Fig. 14 for a first impression).

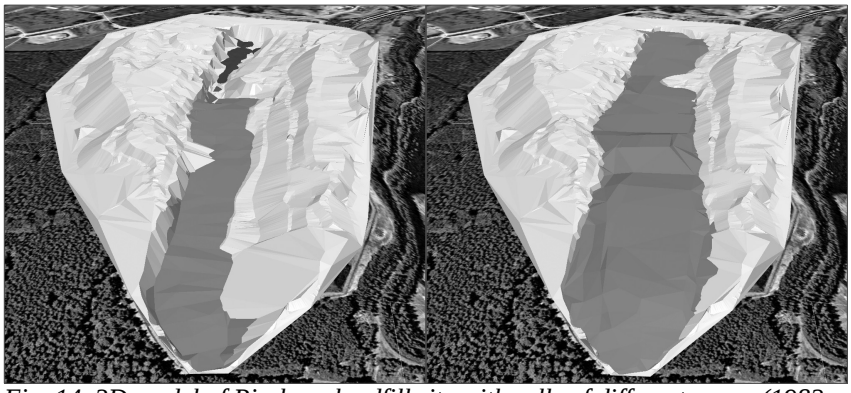


Fig. 14. 3D model of Piesberg landfill site with cells of different usage (1982 and 1993).
Background image: ©2010 GeoContent, ©2009 Tele Atlas, ©2009 Google

Multiple datasets and informations from different sources were integrated to produce the resulting dataset at hand. The Piesberg dataset is a combination of clipping of *digital elevation model* (DEM) of the city of Osnabrück[42], SICAD® drawings, and cross-section drawings. The SICAD drawings are a *top view line drawings* of the *breaking edges* of the

42 The DEM for Osnabrück is provided by the LGN (Ordnance Survey + Geoinformation Lower Saxony), the product is called DGM5 with a grid expanse of 10 m

dumpsite. Cross-section drawings are vertical “cuts” through the dumpsite that illustrate the local height profile. These were first only available as paper drawings and had to be manually digitized. All drawings are available for multiple years. The “flat” SICAD drawings were combined with the height profiles of the cross-section drawings. In this process, the heights from the cross-section drawings have been transferred to the SICAD drawings to generate several 2.5D geomodels. The *Triangulated Irregular Network* (TIN) of the geomodel has been computed by the *TIN module* of *Arc/Info*. Finally, the DEM has been used to expand the geomodel to its sides. After all, the whole temporal model consists of 12 TINs – screenshots of the individual geo-objects, denoting the year of valid time of the respective object, are depicted in Fig. 15.

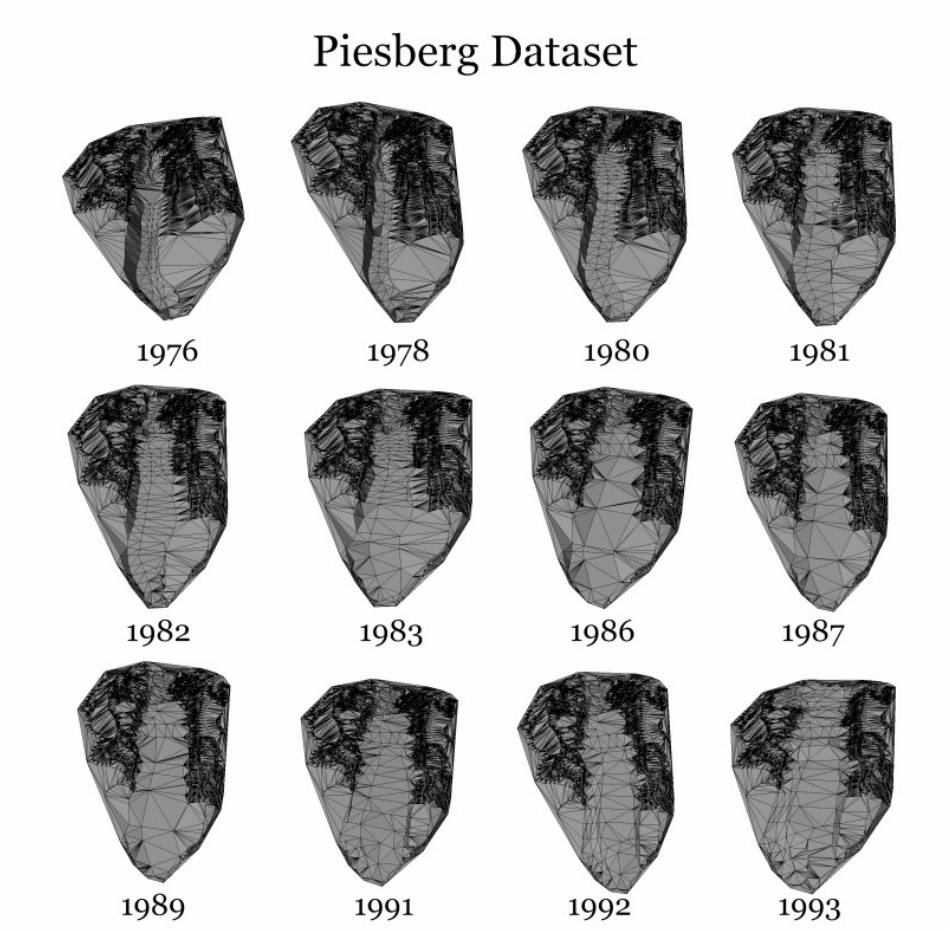


Fig. 15. Complete Piesberg dataset, years 1976 - 1993

The final dataset consists of twelve files (the twelve points in time). Originally LAUTENBACH stored the data in ADF format[43] that can be visualized and converted with *ESRI ArcGIS 3D-Analyst*. In ArcGIS 3D-Analyst the data has been converted to the more open/better

43 Arc/Info Binary Grid format; a binary format developed by ESRI for storing raster data

readable VRML format. In VRML format the complete dataset takes about 67 MB[44]. In a point in time, the model consists of about 30K triangles (so about 300K triangles at large). As it can be seen in Fig. 15, the configuration of the TIN changes from year to year at large. There are parts that nearly stay unchanged (especially at the borders) and parts that greatly change (in the centre). The amount of triangles that build up the TIN also differs from year to year (cf. Table Table 1).

Year	**1976**	**1978**	**1980**	**1981**	**1982**	**1983**
Iteration	*+0*	*+2*	*+4*	*+5*	*+6*	*+7*
Triangles	30,144	30,092	28,742	26,978	26,582	25,688
Year	**1986**	**1987**	**1989**	**1991**	**1992**	**1993**
Iteration	*+10*	*+11*	*+13*	*+15*	*+16*	*+17*
Triangles	24,436	24,474	24,026	24,056	24,004	19,128

Table 1. *Number of triangles in the Piesberg dataset*

In general, the amount of triangles decreases at every time step (with some minor exceptions). For a better overview, the data history are clearly presented in Fig. 16.

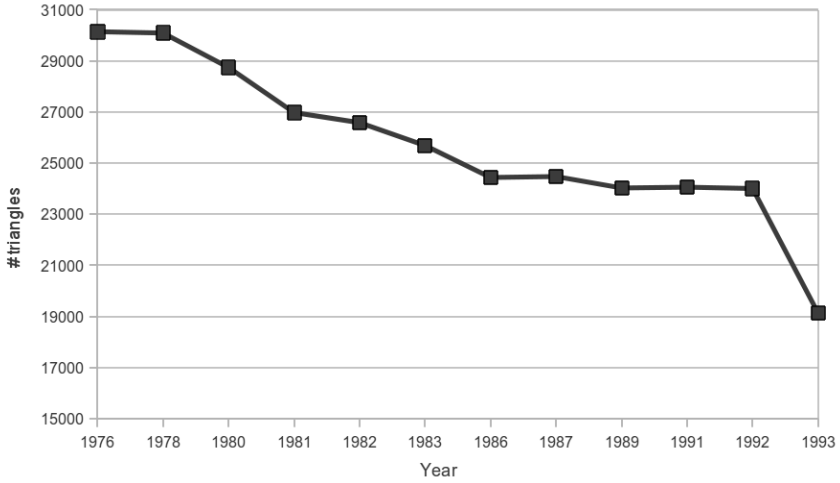


Fig. 16: Number of triangles in the Piesberg dataset

The decrease in the number of triangles is because the model of the landfill itself has a lower resolution (is represented by fewer triangles) than the DEM of Osnabrück, in which it is embedded. Since the landfill continues to expand, more and more surface of the DEM is “covered” by the landfill model, and thus the number of triangles in total is continuously reduced.

The meshing of the triangle model is realised with continuously fewer triangles and subject to strong changes with each time step. Due to this dynamic properties, the data set provides a major challenge to temporally model and keep track its geometry and topology.

44 In comparison ADF format consumes ~ 10 MB

2 Topological Concepts of Spatio-Temporal Data Modelling

As stated in section 1.2.1 (Geological Modelling in Practice), geoinformation occurs in diverse applications in a variety of ways. The underlying models that are used to manage geodata are also diverse, starting with a simple model as *spaghetti structure* that is used as the simplest, unstructured way to gather geoinformation in common GIS (Peuquet 1984, 76 et seq.) to the complex models that are needed to represent temporal 3D objects. This chapter gives a summary of the common models that are most important for sophisticated spatio-temporal modelling and outlines the current state of international research in the field of spatio-temporal model-building for geoscientific data.

2.1 Geometry Model as Basis for the Topological Model

Within spatio-temporal data models, topological information should be supplemented by a geometry model representing the location of geo-objects in space and time. As an example, the *db3dcore* geometry model is an implementation of the *Simplicial Complex* model (Alexandroff and Hopf 1935, 45:158 et seqq.). In the model of *Simplicial Complex*, the considered space is completely subdivided into connected *simplices*, thus it is a specific *cell decomposition model*, cf. (Mäntylä 1988, 72 et seqq.). The intersections between d-dimensional simplices are the (d-1-dimensional) simplices that constitute the boundary of the d-dimensional simplices (see Fig. 17).

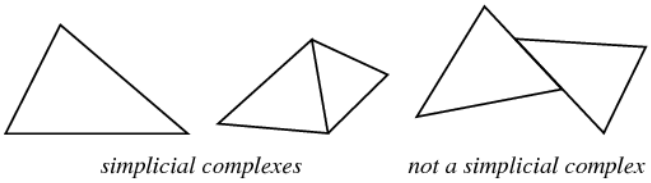


Fig. 17: Example and non-example of Simplicial Complex
Source: (Weisstein 2010c)

The DB4GeO/DB3D Core API implements the Simplicial Complex model for the *spatial part* of its *3D object model*, see UML class diagram in (Bär 2007, 65). The Core API defines a *3D object* to be an object in 3D space that has a spatial part which can be a *point sample*, a *curve*, a *surface* or a *volume*. These abstract geometry concepts are specified by concrete *geo-objects* as follows:

- *point sample* as *point net*,
- *curve* as *line segment net*,
- *surface* as *triangle net*, and
- *volume* as *tetrahedron net*.

The triangle net is a construct that is used in application to model e.g. the strata boundary layers of subsurface models, as depicted in Fig. 2. Tetrahedron nets can be the substructure of blocks of volumetric geomodels, as depicted in Fig. 3. It is obvious that volumetric objects enable more analytical evaluation in application than plain 2D objects, since e.g. the volume size of rock blocks can only be computed with volumes that are represented by closed surfaces. If a volume is subdivided into multiple tetrahedra, the calculation problem for the whole geo-object can be distributed to the single tetrahedra of the net. The calculation of the volume size of a single tetrahedron is algorithmically easier than the calculation of an arbitrarily shaped geometric object. After calculating the volumes of the single tetrahedra, the individual intermediate results can be added to obtain the overall volume size. The concept of splitting a complex geometric object into multiple simple geometric objects is applied in several methods of the API with the aim to ease code complexity and maintenance.

All the aforementioned nets are subdivided into non-overlapping *net components*. A net component itself consists of connected simplices. By the means of the Core API, it is possible to navigate on top of net components by iterating over the explicitly stored *neighbourhood relations* between simplices.[45]

45 This structure can be seen as the implicit topology model of the DB4GeO/DB3D Core API

2.2 Limitations of Navigation on the Geometry Model

The explicitly stored neighbourhood relations between single simplices can be visualized in an *incidence graph*. Fig. 18 shows the incidence graph of the Simplicial Complex model as it is implemented in *db3dcore*.

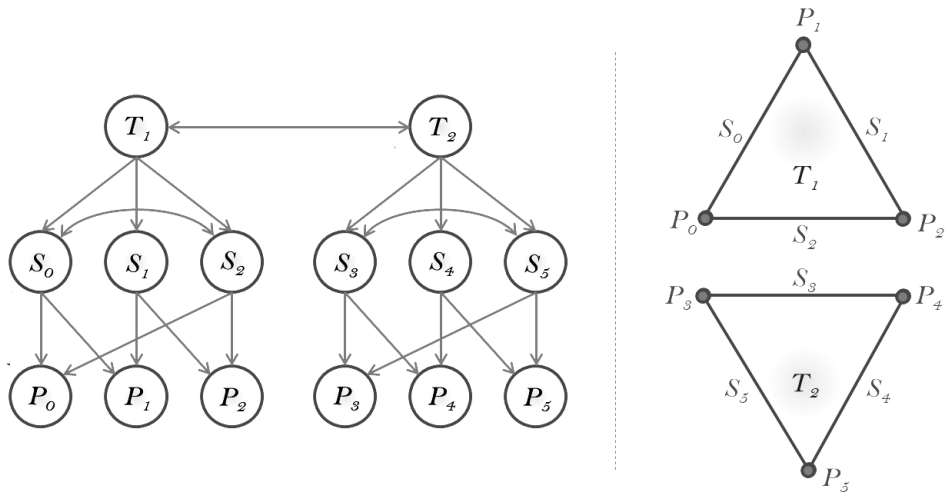


Fig. 18: Incidence graph of Simplicial Complex model of the DB4GeO kernel (T: triangle, S: line segment, P: point)

The arrows (left side) represent the connections between the simplices. Depicted is an example of two triangles that are adjacent through the line segments S_2, S_3 (see right side). There are directed top-down incidence relations from triangles to segments to points, as well as "next to"-connections between multiple triangles (of a triangle net) and equally between multiple segments (of a segment net). This incidence graph is quite usual for geometry modelling systems (B. Lévy and Mallet 1999, 3). Obviously, there are also some insufficiencies concerning the navigational properties of this structure. For example, there are no back references from lower to higher dimension simplices as well as there is e.g. no direct connection between S_2 and S_3, what makes navigation quite difficult. There are cases that force a traversal of the whole structure to do only one step in navigation.[46] Lienhardt (1989) and Brisson (1989) proposed explicit generic topology models that address such problems.

46 For example if it is necessary to find all neighbouring line segments to a given point.

2.3 Cell-Tuple Structure and Generalized Maps

The way to the invention of the *cell-tuple structure* and the *Generalized Maps* was paved by prior topological models that have widely been used in CAD industry. These prior models have been BAUMGART's *winged edge representation* (Baumgart 1975), WEILER's *half-edge structure* (Weiler 1985), and the *radial edge representation* (Weiler 1988). The *cell-tuple structure* has finally been proposed by BRISSON (1989).

2.3.1 Cell-Tuple Structure and Adjacencies

In the *cell-tuple structure*, first, a geo-object of dimension *N* is completely divided into arbitrarily shaped cells of dimensions $N, N-1, \ldots, 1, 0$. Second, *cell-tuples* are defined as ordered sequences of *cells* (c_n) of decreasing dimension. A cell-tuple is denoted as:

$$C(c_N, c_{N-1}, \ldots, c_1, c_0).$$

From another perspective, a cell-tuple corresponds to a path in the aforementioned incidence graph.[47] A cell-tuple structure is a set of cell-tuples that represent all possible paths in the incidence graph. In the set, all cell-tuples are *unique* by their tuple elements.

All cell-tuples of a set are "connected" through the concept of *adjacency* that is inherent to the cell-tuple structure:

$$C\, A_i\, C' \Leftrightarrow \forall\, 0 \le j \ne i \le N, c_j = c'_j.$$[48]

Two cell-tuples C and C' are called i*-adjacent* (A_i) if exactly one cell, viz. the cell of dimension i of the cell-tuple is exchanged (*switch* operation) so that another unique tuple of the set of valid cell-tuples is obtained in return.[49] This structure can easily be mapped to a *relational database*. Once the cell-tuples are stored in a database table, adjacencies can be computed elegantly through SQL statements such as:

```
SELECT * FROM celltuples ORDER BY Node-ID, Edge-ID
```

Listing 1: SQL statement to generate a result that pairwise lists cell-tuples that are 2-adjacent

Listing 1 exemplary shows an SQL statement that can be used to pairwise list all 2-adjacencies (A_2) of a set of cell-tuples. This method can be used as a basis to find any desirable adjacency.[50] Such representations are discussed in more detail in Ch. 2.5.5 in the context of spatio-temporal modelling.

---

47 For example, for a 2-dimensional structure we can note (F_1, E_1, N_1), with face F_1, edge E_1 and node N_1. On terminology of geometry and topology primitives, cf. (Butwilowski 2007, 18 et seq.)

48 Source of this denotation is (B. Lévy and Mallet 1999, 3)

49 For instance, the following two cell-tuples are 0-adjacent: (F_1, E_1, V_1) and (F_1, E_1, V_3)

50 More detail and more examples on the topic of the relational representation, particularly its combinatorial properties, can be found in (Butwilowski 2007, 47 et seqq.)

In an actual implementation, for convenience and faster processing, the cell-tuples can be explicitly linked (see Fig. 19[51]).

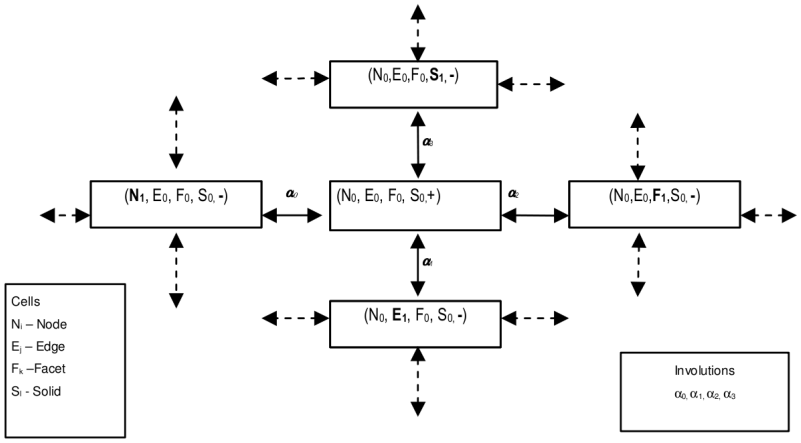


Fig. 19: Graph representation of an oriented 3-G-Map
Source: (Thomsen et al. 2008)

In Fig. 19, an implementation example is sketched, where the cell-tuples are realised as nodes of a graph (depicted as rectangles) and are connected by bidirectional pointers (arrows) that pre-calculate and make the switch operations persistent.

2.3.2 Generalized Maps and Involutions

LIENHARDT (1989) proposed the more abstract model of *Generalized Maps*[52]. A G-Map (G) of dimension N is defined as a pair that consists of a set of *darts* (D) and of a set of operations that are defined on the darts, called *involutions* (α_i):

$$N\text{-}\,G(D, \alpha_0, \ldots, \alpha_{N-1}, \alpha_N)\,.$$

The involutions have to satisfy LIENHARDT's axioms $\alpha_i(\alpha_i(d)) = d$ and $\forall\, 0 \le i < i+2 \le j \le N\,, \alpha_i \circ \alpha_j$ (B. Lévy and Mallet 1999, 4).

A dart is an abstract construct and the involutions are defined as abstract transitions between darts, not specifying the actual realisation of a transition (for a visual representation of an example 2-G-Map, see Fig. 20).

51 In the fig. the order of the sequence of cells is inverse, compared to the definition of Brisson.

52 Abbreviated as G-Maps or GMaps

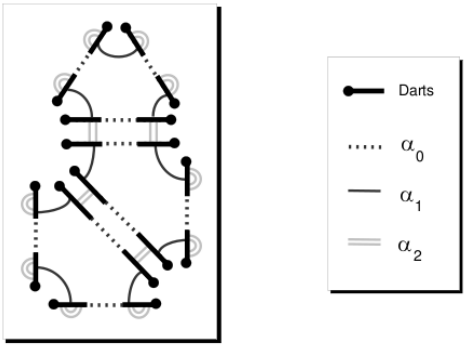


Fig. 20: 0-2-involutions of a 2-G-Map
Source: (B. Lévy and Mallet 1999, 4)

The dimension of a G-Map (*N*) equals the number of involutions in the involutions set. For example, a 2-G-Map (*2- G*) consists of the involutions $\alpha_0, \alpha_1, \alpha_2$.

Cell-tuples are a possible realisation of darts and switch operations are a possible realisation of involutions. In such a case, the cell-tuple structure is a realisation of G-Map. Another viable realisation of G-Map could be to model the darts as shallow objects and the involutions as direct references between the darts (this can also be interpreted as *graph representation* of G-Maps).

Particular attention should be paid to the darts at the *boundaries* of the example 2-G-Map that is depicted in Fig. 20. Since the 2-G-Map "ends" at the boundary, a special handling of the α_2 -involutions is needed: also exemplary depicted in Fig. 20, the α_2 -involution of any dart d_b at the boundary is defined as $d_b = \alpha_2(d_b)$ (i.e. the dart is *self-referencing* for α_2). Outer darts, lying in the 2D universe (face universe) or *outer void*, are not defined in this case, but they can be added easily.

The realisation of G-Maps by Lévy and Mallet (1999) is capable of modelling all kinds of manifold geometric set-ups in 2D and 3D. Though, G-Maps are generally not capable of modelling non-manifold situations as visualized in Fig. 21, they still are capable of modelling some non-manifold objects that are the so-called *Cellular Quasi-Manifolds*[53] cf. (B. Lévy and Mallet 1999, 2)).

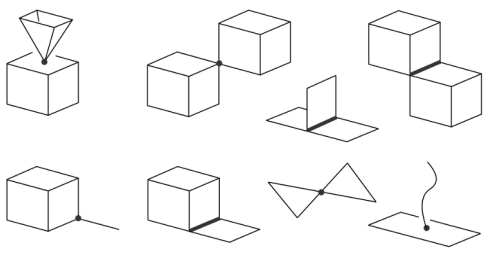


Fig. 21: Examples of non-manifolds
Source: (B. Lévy and Mallet 1999, 2)

53 This class of objects is discussed in more detail in Ch. 3.1.5.

Manifolds are a class of geometric objects that are locally homeomorphic to a disc in 2D or a ball in 3D.[54]

2.3.3 Involution Sequences Forming Orbits

A series of single involution steps can be combined to *orbits*.[55] An orbit is defined as a subset of darts and is denoted as $\langle \alpha_{i_1}, \alpha_{i_2}, \ldots, \alpha_{i_k} \rangle(d_s)$ (B. Lévy and Mallet 1999, 4), where the part in angle brackets ($\langle \ldots \rangle$) is a list of involutions (*involution sequence of the orbit*). Only these involutions are allowed to be traversed. The involutions in the list can generally be traversed in an arbitrary sequence, but they have to obey the preconditions stated in Ch. 2.3. However, at least the orbits $\langle \alpha_i, \alpha_{i+1} \rangle(d_s)$ can also be traversed in an ordered sequence (B. Lévy and Mallet 1999, 5). An example of such an ordered orbit is $\langle \alpha_0, \alpha_1 \rangle(d_s)$ that can be used e.g. to return all vertices of a face in an ordered sequence.

With an orbit, it is possible to traverse all darts that belong to an *i*-cell of arbitrary dimension. This is done by defining an orbit that consists of all involutions but the involution of dimension *i*. Such an orbit is denoted as $\langle \not{\alpha}_i \rangle(d_s)$ [56] or as *i-orbit* (an orbit of dimension *i*), see also (Butwilowski 2007, 66 et seq.), where *i* determines the dimension of the cell that is traversed by the orbit. This is the cell that is completely described by the orbit. For example: a *0-orbit* always performs a repeating sequence of $\alpha_1 - \alpha_2$ -involutions until it reaches the start dart, thus collecting all darts of a 0-cell (node).

Lévy and Mallet present an algorithm to traverse darts of cells of any dimension in (B. Lévy and Mallet 1999, 5):

```
1.  traverse(start: Dart, α_{i_1},α_{i_2},...,α_{i_k} : int)
2.    S : Stack;
3.    mark(start);
4.    push(S, start);
5.    while not empty(S)
6.      Dart d = pop(S);
7.      DO_IT(d);
8.      for j = 1 to k
9.        if not marked( α_{i_j}(d) )
10.         mark( α_{i_j}(d) )
11.         push(S, α_{i_j}(d) )
12.       end if
13.     end for
```

54 For a comprehensive definition, see (Remmert 1964).

55 Orbits are discussed in greater detail in (Butwilowski 2007, 35 et seq.)

56 e.g. the orbit $\langle \alpha_{0,} \alpha_{1,} \alpha_2 \rangle(d_s)$ on a 3-G-Map traverses all cell-tuples of a 3-cell (solid) since the only missing involution in the orbit is α_3, thus $\langle \not{\alpha}_3 \rangle(d_s)$

```
14.   end while
15. end traverse
```

Listing 2: Dimension independent orbit traversal algorithm (pseudo code); source: (B. Lévy and Mallet 1999, 3)

Fig. 22 more clearly shows the execution sequence of the introduced orbit traversal algorithm in a flow chart diagram.

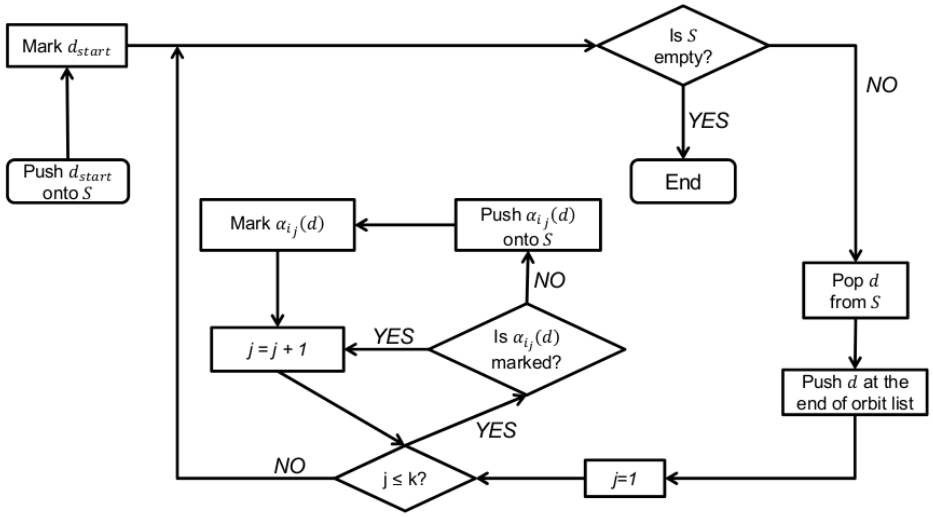


Fig. 22: Flow chart diagram of dimension independent orbit traversal algorithm

To start the process, the method user passes two parameters to the orbit traversal method:

- a start dart (d_{start}) and
- a list of involutions $\alpha_{i_1}, \alpha_{i_2}, \ldots, \alpha_{i_k}$ of size *k* to be applied to form the orbit.

d_{start} is the dart with which the orbit shall start and end. First, d_{start} is marked as "visited" and pushed onto a stack of darts (*S*) that yet have to be processed. If *S* is empty, then the algorithm terminates. *S* is not empty in first step, since d_{start} is already in *S*. As long as *S* is not empty, the top dart *d* is popped from *S* and processed by the API user. The type of processing depends on the case of the user and can e.g. simply be a push of *d* onto the end of a *resulting orbit list*. This arbitrary, case dependent processing is paraphrased by the DO_IT function call in line 7.

Then *j* (which is the index number of the involutions list) is set to *1*. Then, in a `for`-loop, all involutions of the *involutions list* are applied on *d*. Every d' ($d' = \alpha_{i_j}(d)$) is checked whether it is already *marked* (has already been visited). If it has not been visited, it is pushed onto *S* and marked. Then the next dart *d* is popped from *S*. Since, in this way, it can be guaranteed that any dart of an *N*-G-Map is visited at least once, this algorithm is dimension independent.

G-Maps offer a way of flexible navigation on cells. The type of manageable cells is not restricted to simplices but they may be all kinds of *connected cell subdivisions*, thus this model has a higher *expressive power*. For example, the G-Map representation may also be seen as a generalisation of B-Rep (B. Lévy and Mallet 1999, 2), and therefore is also capable of modelling B-Rep. The model facilitates simple, short algorithms, since it is dimension neutral (though in their presentation of the model in 1999, LÉVY and MALLET give visual examples and some details only on 2-G-Map). Due to their flexibility and modelling power, G-Maps are used as a topological toolbox by the widely-used geomodelling software GOCAD, as indicated in (Royer 2004, 4).

2.4 Managing Geomodels with Multiple Levels of Detail

While Ch. 1.4 introduces the basic ideas of generalization/abstraction of geodata, this chapter will detail the concepts and present specific common methods of geometric and topological generalization that have been developed in previous research efforts.

2.4.1 Hierarchy Relationships as Links Between LoD

In their description of ATKIS (a four LoD DLM, cf. Ch. 1.4), ANDERS and BOBRICH outline three ways to generate the links between objects of different levels. First, by *manual linking* where an operator manually and interactively sets the links between multiple representations of the same geo-object on different scales. No automation algorithms are involved in this process.

Second, *linking by matching* which is a semi-automated process that identifies geometry objects on different scales that possibly represent the same geo-object. The matching algorithms analyse different aspects of a geo-object which are the geo-object's geometry, topology and semantics. More elaborate algorithms incorporate all three aspects in integrated *relational matching procedures*. Still, every LoD comprises its own separate dataset that is maintained independently of the other LoD. The process of manually maintaining every LoD separately is error prone and cost intensive.

Therefore, a third approach is to automate or at least to semi-automate the process of generalization in an MRDB by predictable/deterministic and thus repeatable algorithms.[57] By this approach, only the dataset with the highest LoD is explicitly modelled by the user.[58] All other LoD are automatically derived from the highest LoD/largest scale (this becomes the *base dataset*). The links between the geo-objects of different LoD are generated automatically during generation and later with every editing process on the base dataset.

57 In fact, Fig. 8 shows the results of a semi-automated generalization process for the abstraction of polygon features.

58 Though, in different contexts, e.g. in building planning, a top-down approach is more suitable for the workflow, since architects design simple building models first and then add more detail to their models step by step.

This is only possible if the generalization rules are sophisticated enough so that it is always known which geo-objects of a larger map scale have to be condensed to which geo-objects of a lower map scale.

Subsequent research on MRDB dealt with topics such as the optimization of updates on already existing data sets of multiple scales (Haunert and Sester 2005), which was also termed *incremental generalization*. The (semi-)automated generation of small-scale maps from larger scales is computationally intensive. The aim of the research of HAUNERT and SESTER was to identify methods that allowed for local modifications in the base dataset without the necessity to fully recalculate all map areas of all smaller scales.

2.4.2 Progressive Abstraction/Reduction of Geometry

As indicated in the introductory chapter, the topic of geodata generalization is not limited to "traditional" 2D map data but encompasses other geoinformatics disciplines and geospatial data models, such as 2.5D meshed surfaces, 3D vector data, or georaster data (Sester et al. 2008). For example, the technique of *progressive mesh representation* (PM) which has been introduced in (Hoppe 1996), is regarded as an abstraction technique (see Fig. 23).

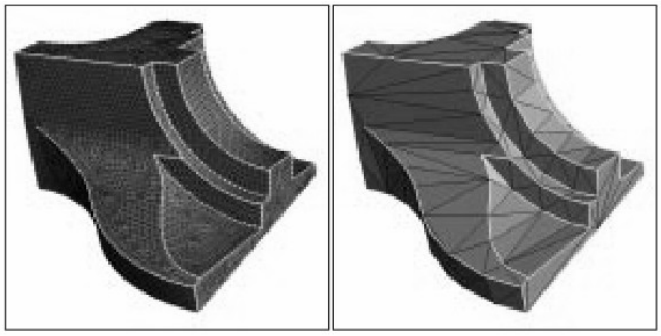


Fig. 23: Application of the progressive mesh algorithm (left: a geometric object in full detail; right: the same object with reduced detail)
Source: (Hoppe 1996, 108)

Fig. 23 shows an application of a PM algorithm on a geometric object that is described by a triangulated mesh surface. The left figure shows the object in full detail, while the right figure shows the same object with a mesh of reduced detail.

PM provides the means to represent a spatial object that is modelled as a triangulated surface in different degrees of detail, from the least detailed mesh that still preserves some of the main geometric characteristics of a geo-object (M^0) to the most detailed geo-object (M^n). This is achieved in two steps: first, M^n is taken as a basis on which elementary operations are applied to reduce the detail of the geo-object step-by-step by reducing the amount of triangles (to "thin out" the geo-objects) and therefore to lower the descriptive details of the geo-objects. The operation that is used to reduce the detail is the *edge collapse* transformation that merges two vertices and two triangles and deletes the edge

between the two vertices. The edge collapse operations are conducted in such a way that the main geometric characteristics of the geo-object are preserved. This is made possible through the inclusion of attribute information of the geo-object into the PM algorithm. Attribute information could comprise for example *breaklines*.

At the end of the PM reduction, M^0 remains. Only M^0 and all detail reduction operations are recorded. Intermediate complete versions of the mesh are not stored. This saves a great amount of storage space, since only the differences (deltas) between each detail representation but not the bulky detail representations themselves are stored. Finally, all intermediate versions of the mesh and the most detailed geo-object ($M^1, M^2, \ldots, M^n$) can be restored by applying the inverse operation of each recorded edge collapse operation step-by-step on M^0. The inverse operation of the edge collapse operation is the *vertex split* operation. The vertex split operation splits one vertex into two vertices and connects them by a new edge (which necessarily leads to the creation of two new triangles).

One of the great advantages of PM is that any reduced geometric model always remains to be a triangulation, which is important for 3D graphics processing, since 3D graphics hardware is optimized on efficiently handling triangle meshes. A PM algorithm has been implemented in DB4GeO by Kuper (2010). This implementation focused on reducing the complexity of geometric models for computational purposes. A geo-object that is reduced in detail can more easily (faster) be processed, uses less memory and thus needs less bandwidth when it is transmitted through a computer network.

2.4.3 Generalized Topological Approach on Multiple Representation

While the MRDB approach presented in Ch. 1.4 only aimed at managing links between geometries of one type on different LoD (e.g. only between polygons), Bruegger and Kuhn (1991) integrated the cell concept and cell connectivity (i.e. topological representation) into their considerations. One of the early attempts to structure the subject matter of hierarchical management of topological representations – so called *multiple topological representations* (MTR) – was developed by them. Bruegger and Kuhn argue that MTR are inevitable for multiple LoD of geodata to avoid severe performance problems in processing topological queries. They elaborate a GIS application example that demonstrates that unnecessary topological information is an obstacle to efficient algorithms for certain spatial queries that refer to particular (lower) LoD. Fig. 24 shows, for an example of a 2-dimensional cell decomposition, how detailed topological information can lead to information overload at lower map scales.

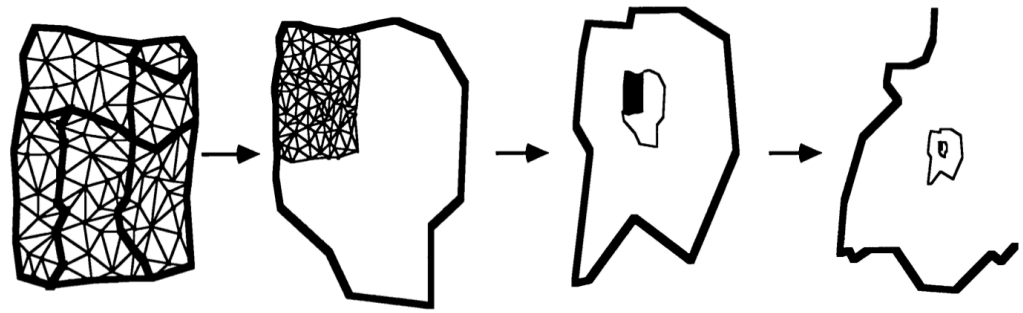


Fig. 24: Increasing number of topological cells at lower map scales
Source: (Bruegger and Kuhn 1991, 8)

Fig. 24 depicts demarcation of land area on multiple map scales. The map scale lowers from left to right. The thick lines in the leftmost depiction show the boundaries of property parcels of a city, the rightmost depiction shows a whole state. In this example, the parcels are internally modelled as TIN (indicated by thin lines in left figure). A single representation model would preserve all topological information throughout all map scales, leading to an extensive topological information overload at the lowest map scale. In such a case, a topological query such as "which states are neighbour states to a given state?" would involve an iteration over all outer boundary line segments of the outer property parcels, making this a computationally expensive query.

Assuming the usage of MTR in such applications, topological queries can be performed in constant time. BRUEGGER and KUHN elaborate a general concept on MTR, without restricting their approach to a specific cellular model. Their considerations apply to any cellular model of arbitrary dimension (whether Simplicial Complex or others). They introduce *largest homogeneous cells* (LHC) which are an abstraction of specific, implementable topological cell types.[59], and that are used as the basis of MTR. Cells that represent the same point set on different LoD are linked in both directions. In an MTR, a set of cells of one level is connected to a set of cells of another level. The inherent dimension of the cells does not matter: e.g., a 2-dimensional cell may be represented as a 1-dimensional cell at another LoD. All correspondence relations between all topological cells of all contained dimensions are modelled. Fig. 25 shows a diagram representation of an MTR with 2-dimensional LHC and all bidirectional links.

59 In Fig. 24, for example, LHC are realized as Simplicial Complex cells.

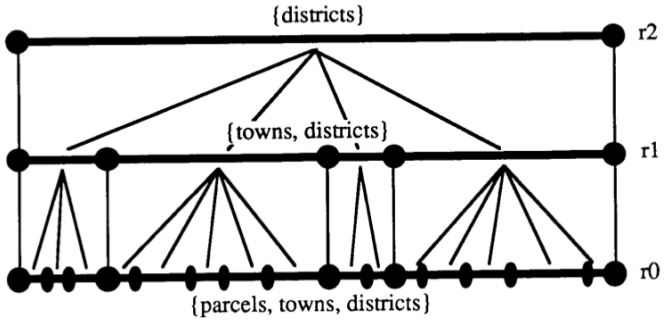


Fig. 25: Links between cells of multiple LoD
Source: (Bruegger and Kuhn 1991, 14)

The diagram presents a stylized profile view of the above introduced application example of land area demarcation on multiple map scales, with districts demarcation on lowest, towns demarcation on middle and parcels demarcation on highest map scale. All topological elements on different LoD that represent the same point set are linked. In the example, not only the demarcation surface elements of different LoD are linked but also all boundary edges that represent the same boundary at different LoD, and also all nodes. The diagram shows all links between surface elements and between node elements.[60]

Many links of a multi representation are not modelled explicitly but are given implicitly through the *transitivity rule*. Even if two levels of detail of a multi representation are not directly connected/adjoining, then nonetheless their geometries are linked indirectly. The transitivity rule states that if a subgeometry on a higher LoD is part of a parent geometry on a lower LoD, then all geometries that are part of the subgeometry on even higher LoD are also part of the parent geometry of the lower LoD. All indirect hierarchy relations can be derived by following all direct links from a lower LoD geometry to its higher LoD representations through all intervening hierarchy levels.

Bruegger and Kuhn identify two types of relations between cells of different LoD, which are *1:m*- and *m:n*-relations. *m:n*-relations allow for multiple linking in both directions of two adjoining LoD. This implies on the one side that a cell at lower LoD is represented by one or more cells at higher LoD. On the other side, a cell at higher LoD may partly represent not only one but also two or more cells of a lower LoD.[61] Such a constellation is not possible with *1:m*-relations, since these allow multiple linking only in one direction from lower to higher LoD. This implies that a cell at lower LoD is represented by one or more cells at higher LoD but that a cell at higher LoD always partly represents only exactly one cell of a lower LoD. If all links in an MTR are modelled as *1:m*-relations, then any higher LoD is said to be a *refinement* of all lower LoD. Only from refinement relations follows a clear hierarchy of abstraction. The diagram in Fig. 25 shows an example of an

60 Edges are also linked but they are not depicted only due to diagram type.

61 An example of an m:n-relation would be the hierarchical linking between countries and UTM zones. A UTM zone may include any number of countries. In the opposite direction, a country belongs to multiple UTM zones if the country is crossed by grid lines of the UTM system.

1:m-relational MTR. Finally, Bruegger and Kuhn show that only 1:m-relations preserve connectivity relations (i.e. do not lead to contradictory topological representations) of a cellular partition on different LoD. Therefore, in the implementation part of the thesis only refinement relations will further be dealt with.

2.4.4 Application of Multiple Topological Representation in Subsurface Modelling

Multiple topological representations can be applied in a variety of application cases, not only in demarcation of land area on multiple map scales but also e.g. in geological modelling. Fig. 26 shows an example of an MTR in geological modelling.

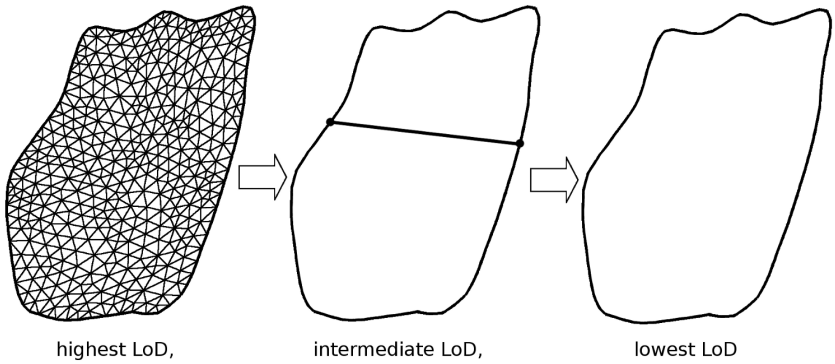


Fig. 26: A subsurface fault, depicted in three levels of detail

The figure shows a geometric model that represents a subsurface fault. The level of detail decreases in the figure from left to right. The left depiction shows a representation of the fault at highest level of detail as a TIN. It represents the direct interpretation of the collected raw data. Such grade of detail may comprise a large amount of data which might be unnecessary/overwhelming in some application cases.

One of the lowest possible LoD that still preserves the overall geometry is simply the boundary of the fault (see right depiction of Fig. 26). This “big cell” on lowest LoD is a simplified representation of the whole fault. Even such simple representation is already useful in some application cases, for example if only the general position and extend of a fault is of interest.

In between these both outer stages, any number of intermediate abstraction instances are feasible. For example, in the figure, one intermediate step of abstraction is presented in the central depiction. In the intermediate step, the whole fault object (at lowest LoD) is subdivided into two parts by a line segment that “cuts through” the surface and splits it into two new connected surfaces. In geological application, this is a useful subdivision, for example to indicate that a fault extends into two geological horizons on both sides. The cutting line segment indicates where the borderline between the geological horizons passes on the fault.

2.4.5 Using G-Maps for Multiple Topological Representations

An example of a concrete multiple topological representation specialized on the creation of architectural models on the basis of G-Maps has been presented by FRADIN, MENEVEAUX, and LIENHARDT in 2005. The definitions of G-Maps/cell-tuple structure presented in Ch. 2.3 facilitate only the modelling of geo-objects of one certain LoD. The definitions do not make any statements on how to model multiple LoD. However, (Fradin, Meneveaux, and Lienhardt 2005) extended the concept of G-Maps by the concept of *H-G-Maps*[62] to add the possibility of managing multiple LoD. Their approach is based on copies of G-Maps, where every LoD is represented by a complete G-Map. Thus, an H-G-Map is defined as a *sequence* of G-Maps:

$$G_i(D^i, \alpha_0^i, \ldots, \alpha_{N-1}^i, \alpha_N^i)_{i=0,m},$$

where i is a certain hierarchy level of an H-G-Map and m is the number of available levels. Fig. 27 illustrates the process of creating a new LoD of a geometric object from an existing LoD with H-G-Maps during an editing session.

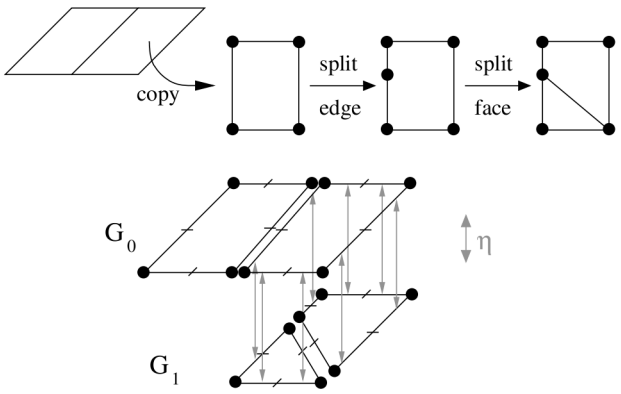


Fig. 27: Editing of an H-G-Map (visualized on two hierarchy levels)
Source: Fradin et al. 2005

The operator starts with a simple geometric object that consists of two rectangles (see top left depiction in Fig. 27). This object is considered to be the object with the lowest detail. The topology of the lowest detail object is internally modelled by a G-Map (G_0). The operator wants to edit one of the rectangles in order to add a detail. To add a detail means to establish a new hierarchy level. Thus, in advance to the actual editing process, the operator creates a copy of the rectangle, and the copy is labelled to be an object of a higher LoD. All darts of G_0 that model the polygon's topology are also copied. The copied darts produce the G-Map of a new hierarchy level (G_1). Consequently, the system consists of two topological LoD (G_0 and G_1).

62 Hierarchical G-Maps

A navigation between different LoD is made possible by explicit links. First, FRADIN et al. use a general, bidirectional link on whole G-Maps between G_i and G_{i+1}. Second, a finer graded connection between the two G-Map levels is established as a *bijection* (symbolised by an η) between the newly created darts of G_{i+1} and their representatives on G_i. This is visualized in the lower depiction of Fig. 27 where the light grey arrows symbolize bidirectional links between each dart that exists in G_0 and its copy in G_1. The bidirectional links allow to navigate between darts of different LoD. This leads to a flexible navigation on the hierarchy since it enables instant movement between all hierarchy representatives of any cell-type. For example if it is necessary to retrieve a certain edge at a higher detail, the system only needs to access an arbitrary dart of the edge and follow η one step up which will return the higher level representative of the given dart. Finally, the higher detail edge is accessed through the returned higher level dart.

After the whole copy has been created and the bijection completely established, the operator can start editing the object at higher LoD to add detail. In the example of Fig. 27, the operator first adds a node to an edge (which splits the edge into two) and then adds an edge between two nodes (which splits a face into two faces). In this process, new darts emerge in G_1 that all have no representatives in G_0 and thus, there are no additional links set between darts of both levels (see Fig. 27). Accordingly, there is no link between the newly added node and its less detailed representative. This is correct behaviour by the system, since this node is only available at higher detail and has no lower detail representative.

In (Fradin, Meneveaux, and Lienhardt 2005), H-G-Maps are used exclusively for architectural modelling. FRADIN et al. present application examples, where H-G-Maps make it possible to manage, render and visualize comparably large and detailed models on standard desktop hardware. They show examples of large 3D building models with several stories, indoor spaces and detailed interior decoration. In their concept, the least detailed model of a building (G_0) is a volume that represents only the exterior walls. At the next higher LoD (G_1), the building model is detailed by indoor walls that define indoor spaces. At highest LoD, the indoor spaces are increasingly detailed by elaborated models of the furnishing. Since H-G-Maps are used internally, the different LoD representations of the same volumetric object are connected by bidirectional links.

For example, an indoor room (G_1) is connected by bidirectional links to the exterior shell model of the building (G_0) to which it belongs. The number of steps an algorithm needs to navigate from higher LoD to lower LoD differ, depending on the spatial position of the dart that is used for hierarchy navigation. If the dart is positioned at a cell that is shared by both LoD (e.g. a face that represents an exterior wall), then only one step suffices. If the dart is positioned at a cell that only exists at the higher LoD (e.g. a face that represents an interior wall), then an algorithm first has to perform an orbit in order to find a dart that is shared by both LoD. Depending on the topological situation, the orbit that is needed to find a dart that links between LoDs, can be a 1-, 2-, or even a 3-orbit. There are even spatial

configurations, where no hierarchy link can be set. For example if a room of a building lies completely inside the building and does not share any wall with the shell of the building, then H-G-Maps do not establish a hierarchy link between the room and the building at all. This makes finding hierarchy relation still difficult in some special cases.

Following FRADIN et al., THOMSEN and BREUNIG proposed to use H-G-Maps for other application domains than architecture also. (Thomsen and Breunig 2007) specifically elaborated an example in which hierarchical 2-G-Maps (see Fig. 28) are used for the generalization of 2-dimensional land use maps.

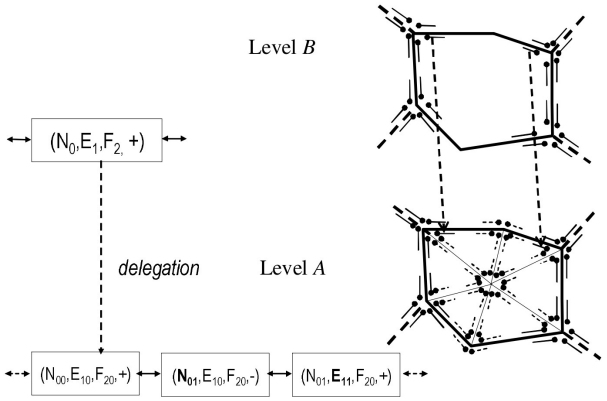


Fig. 28: Generalisation by aggregation in a hierarchical 2-G-Map. Source: (Thomsen and Breunig 2007, 248)

The application example builds on the use case of HAUNERT and SESTER that has been presented in Ch. 2.4.1. The depicted cells in Fig. 28 can be interpreted as land use areas at two hierarchy levels that are managed in an MRDB. For example, the area on the generalized level *B* could be general agricultural land. On level *A*, this area is subdivided into different kinds of agricultural land like wheat fields, corn fields and so on. The affiliated cells of the different levels are interconnected by bidirectional links between the cell-tuples that are appended to the cells.

The last section of the chapter presented an approach that uses G-Maps to model the hierarchy of the topological representation of geodata. The description of the concept and application examples showed the versatility of the introduced H-G-Maps approach. In Ch. 3.5, the presented H-G-Maps approach is used as a basis and expanded. First, it is necessary to adjust the model to the needs of the underlying DB4GeO architecture and the G-Maps kernel model for DB4GeO that is developed in Ch. 3.1. The adapted H-G-Maps for the DB4GeO approach will also deal with some disadvantages of the H-G-Maps that has been presented in this chapter. The disadvantages are discussed in Ch. 3.5 in detail.

2.5 Modelling the Temporality of Geoscientific Data

While Ch. 1.5 presented a general introduction to the topic of spatio-temporal models and clarified the meaning of essential terms of the subject area of temporal geodata, this section will introduce a number of specific spatio-temporal models that mostly focus on the management of large kinematic geo-models, and discuss their advantages and drawbacks. The interaction of topology, geometry and net meshing is of particular importance in this examination.

2.5.1 Concepts of Continuous and Discrete Temporality

While time is changing *continuously*, information on time can only be computed *discrete*.[63] In application, during data input, usually not every (minimal) change of a geo-object is stored, due to limited amount of storage space and limited amount of measured time steps (in the original data). But when it comes to data retrieval, it is valuable to access the geometry of a geo-object in a *high temporal resolution*. Thus, *interpolation* or *approximation* techniques have to be employed to gain synthetic time step data in-between the explicitly modelled time step data. This idea is conceptually illustrated in Fig. 29.

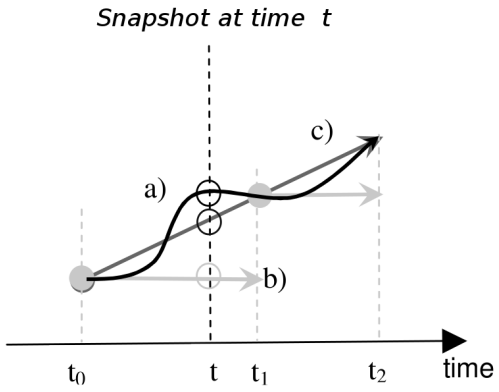


Fig. 29: Continuous vs. discrete modification of geometry in time.
Source: Drawing by Andreas Thomsen, KIT

In Fig. 29, curve *a)* represents the actual geometric alteration/movement of a real world object (the process to be represented). Computed models of the real object are stored at t_0, t_1 and t_2 in a database.

Curve *c)* indicates a *linear interpolation* between the explicitly stored temporal instances of the geo-object. This means that a linear interpolated instance of the geo-object can be obtained at any time instant from such a system, but the instance will probably have an

63 Since a computer is a finite-state machine, the maximum number of internally representable time steps (temporal resolution) is always limited. Thus, there must always be a “time leap” between time steps (discrete time computation).

offset compared to the real object (like at time step *t*). Other interpolation methods than linear interpolation can be implemented as Geo-DB methods.[64]

In contrast, curve *b)* represents a *stepwise approximation*, where the geometry is not interpolated but changed "abruptly" only at fixed time instants. Before the abrupt change, the object simply constantly adopts the geometry of its temporal predecessor.

DB4GeO, for example, uses *linear interpolation* between 4D vertices to create synthetic geometric representations of a geo-object in its *spatio-temporal modules* (see Ch. 2.5.3 and Ch. 2.5.4). However, though interpolation with a high temporal resolution between temporal instances of a geo-object is possible, the topology of a geo-object can only be changed in a discrete manner of clearly distinguishable states. Thus, in order to combine geometric and topological data in a temporal system, the underlying temporal models for geometric interpolation and topological representation have to be integrated.

2.5.2 TimeStep, an Adaptive Time-Dependent Discretization

Relevant work on the issue of integrating geometric and topological change in a temporal model has been done by POLTHIER and RUMPF. In (Polthier and Rumpf 1995) they propose the notion of `TimeStep`, an *adaptive time-dependent discretization* (see Fig. 30).

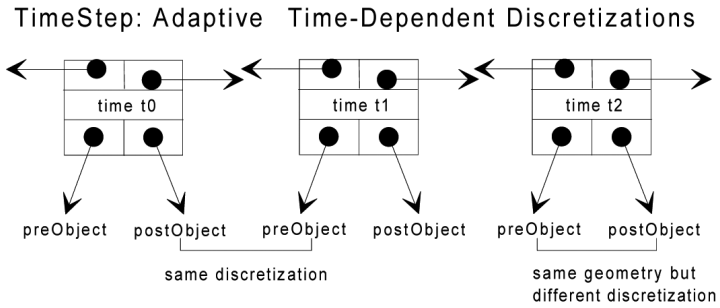


Fig. 30: The Class TimeStep
Source: (Polthier and Rumpf 1995)

The class `TimeStep` models the state of a geometry of a certain geo-object at a particular time step (time instant). The geometry of an object cannot change "inside" a time step – it is "frozen" in this state. But the topological configuration of an object (its *discretization*) may change (adapt) inside a time step, between the *pre-object* and the *post-object* of the same time step. This is indicated in the example set-up of Fig. 30, where every `TimeStep` object *t0*, *t1*, *t2* references a `preObject` and a `postObject` that represent the same geo-object at a certain time step with a similar[65] geometry but with a potentially different topological description.

64 For example: morphing

65 "Similar" in the sense that the overall geometric characteristics of the object are similar but the underlying geometry pattern that describes the object may vary.

An alteration of the geometry of an object is possible in the *time interval* in-between two time steps. The transition between two geometrically altered states of an object proceeds between the `postObject` of the preceding time step and the `preObject` of the succeeding time step. POLTHIER/RUMPF use *linear interpolation* to generate intermediate geometric instances between a `postObject` and the subsequent `preObject`. In this phase, the topological structure of the object must remain unaltered.

In applications that process temporal geo-data, the geo-objects are exposed to alterations of their geometries. If the geometric changes are extensive, it becomes necessary to change the discretization of the geo-object in order to better reflect the transformed geometry (for example to add or reduce mesh detail in certain parts of the object). With the introduction of `TimeStep` class, POLTHIER/RUMPF made it possible to model geometric change and to model changes of the object's discretization at fixed time steps in one system. This greatly improved the usability and applicability of the model in many application domains.

2.5.3 Temporal Point Tube Model of DB4GeO

The concept of time steps is also used by ROLFS and THOMSEN in the design of the *STO model* of DB4GeO[66] (Rolfs 2005) which became the first model for *spatio-temporal objects* (STO) of DB4GeO. The major element of this STO model of DB4GeO is the class of `SpaceTimeElement` (*ste*). An *ste* is an ordered pair of simplices (*start* s_1 and *end* s_2 simplex). The two simplices are extended with information on the time instant in which they exist ("are alive") and with rules that describe an interpolation between the *point tubes* of the two simplices. An *ste* describes the state of a simplex at two time steps (see lower part of UML class diagram in Fig. 31[67], and cf. the UML diagram in (Rolfs 2005, 68) for a comprehensive overview of the class structure of the space-time module).

66 Designed and implemented during the research project "Development of Component-Software for the Internet-Based Access to Geo-Database Services".

67 Singular simplices are of type "simple geo-object" (SimpleGeoObj) in DB4GeO/DB3D.

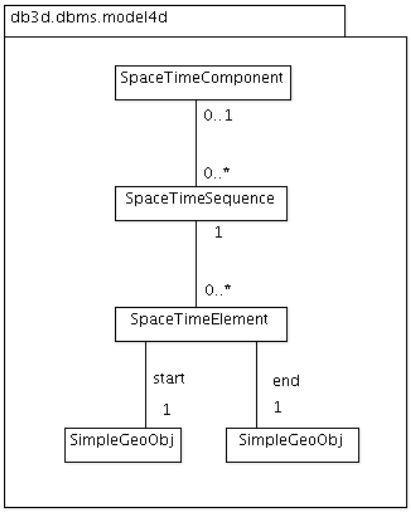


Fig. 31: Simplified UML diagram of the space-time model of DB4GeO/DB3D

A series of *ste* is combined to a `SpaceTimeSequence` (*seq*) in such a way that $s_2(seq_i) = s_1(seq_{i+1})$ [68], i.e. the *end* simplex of a preceding *ste* equals the *start* simplex of the succeeding *ste*. However, a *seq* can describe the geometric evolution of a simplex at any number of user-added time steps.

A set of spatio-temporally non-overlapping, adjacent *seq* constitutes a `SpaceTimeComponent` (*stcomp*). An *stcomp* does not only describe multiple user added states of a simplex but multiple states of a whole net component of simplices (see Fig. 32 for a schematic illustration of an *stcomp* with two *seq*; cf. illustration in (Rolfs 2005, 55) for a geometric example of this situation).

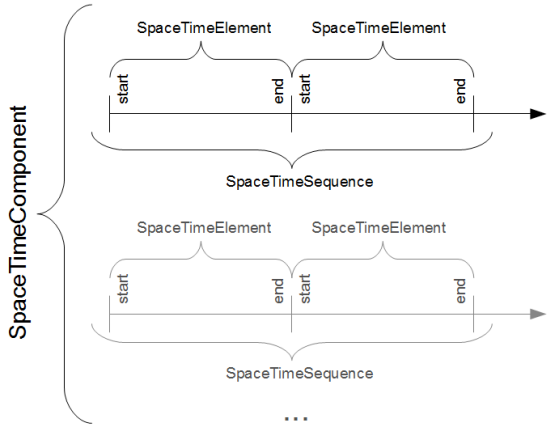


Fig. 32: Spatio-temporal component consisting of two spatio-temporal sequences.

68 Where *i* is the index of the time step

A notable restriction in this model is that the topology of the mesh of *stcomp* has to stay invariant throughout all time steps – only the geometry may change. Multiple *stcomps* with varying mesh topology can be assembled successively in order to create a temporal geo-object that changes its discretization through time. This is similar to the concept of POLTHIER/RUMPF, in the way that the process inside an *stcomp* is analogue to the process between two time steps in the model of POLTHIER/RUMPF. A valuable extension in comparison to the concept of POLTHIER/RUMPF is that an *stcomp* can have not only two but any number of user-added time steps.

The space-time module of DB4GeO is capable of computing spatio-temporal analytical information and spatio-temporal intersection queries, such as:

- "what is the average speed of a certain volume?" or
- "do the trajectories of a line segment intersect a certain rectangle?" or
- "is a segment contained in a tetrahedron at any time step?"

It is also capable of computing artificial 3D models at user-defined time steps (snapshots) and capable of using spatio-temporal access method (STAM).

In the taxonomy of reasoning tasks of SHOHAM and GOYAL (cf. Ch. 1.5), the point tube model of DB4GeO is best covered by the category *planning tool*, since it needs explicit geometric objects and a set of rules in order to operate properly. Furthermore, according to the taxonomy introduced in Ch. 1.5, the system can be classified as an *accumulative forward oriented versioning* system, since every new object that is added to the database is stored fully redundant, and the first object that is added to the database, is taken as the reference point for newer versions of the object. However, the model only provides management of *valid time* in a *longitudinal configuration* (*historical database*), since it does not additionally record transaction time nor does it support the branching of time.

2.5.4 Temporal Joint Model of DB4GeO

In order to overcome the issue of highly redundant geometry storage of the ROLFS/THOMSEN point tube model, KUPER and THOMSEN drafted in 2010 a new spatio-temporal model for DB4GeO. Like the model of ROLFS/THOMSEN, it also falls under the category *planning tool* with a *forward oriented versioning* of objects in *valid time* in a *longitudinal configuration*. The important difference is that the forward oriented versioning is implemented to be *non-accumulative*., i.e. only the differences of newer versions of a geo-object are stored explicitly; unchanged parts of an object are reused.

KUPER (2010) describes the implementation of the revised 4D model of DB4GeO. The new 4D module merges the concepts of *PointTube*, *delta storage* and POLTHIER/RUMPF in one joint model (further referred to as *Temporal Joint Model*). Each of the three concepts addresses a different issue of spatio-temporal modelling. The main objective for implementing *PointTube* model is to simplify data handling for interpolation and analysis calculation. Geometric calculations that integrate multiple time steps of the same geo-

object are algorithmically simpler if the passage of a net geometry through time is perceived as separate passages of the net's points through a tube that is stretched in time. The *delta storage* concept reduces the memory footprint in volatile and non-volatile memory, since it stores only the differences between the individual time steps of a geo-object. Finally, the model of POLTHIER/RUMPF facilitates the possibility to support temporal changes in the meshing of a Simplicial Complex.

KUPER extended the existing DB4GeO API by the Temporal Joint Model as depicted in the architecture overview diagram of Fig. 33.

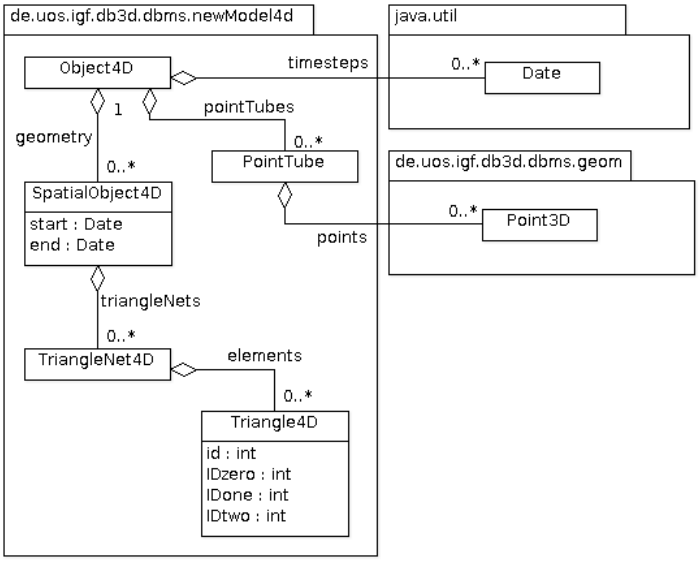


Fig. 33: Architecture overview diagram of Kuper's 4D model for DB4GeO

The general approach of the Temporal Joint Model is to define a temporal object (`Object4D`) on the basis of two parallel structures, the point tube and the spatial 4D object. While the point tube comprises the point geometry of the temporal object, the spatial 4D object models its net meshing. The API has to constantly maintain consistency between both branches.

On the most abstract architectural level, the Temporal Joint Model allows for the instantiation of an `Object4D`. An `Object4D` has to be populated with a list of time steps (`Date`[69] typed `timesteps` list class attribute of `Object4D` class). The `Date` objects define the time steps at which the temporal object provides explicitly modelled geo-object instances. Every temporal object additionally contains a list of spatial 4D objects (`SpatialObject4D`) that define the *meshing of the net* (`geometry`) of the 4D object.

69 The Date class of Java operates on the basis of Java data type long (2^{64} possible values) that stores milliseconds. This is sufficient for a description of time for ~ 300 million years into the past and into the future. This could already be to short for geological applications. However, since this is merely a prototype implementation, a Date object is accepted as adequate.

Each spatial 4D object has a `start` date and an `end` date of type `Date` and a set of temporal triangle nets (`TriangleNet4D`).[70] Each temporal triangle net owns a set of temporal triangles (`Triangle4D`). Each temporal triangle includes three *ID*s that identify the three points of the triangle. For each node/point ID of a Simplicial Complex, one point tube with the same ID is defined directly at the temporal object (`pointTubes`). The system has to maintain the consistency of the IDs internally. A point tube itself consists of a set of `points` of type `Point3D`. The points of a point tube represent the same point at several time steps; a point tube is the "spatio-temporal path" of a point.

This model allows for memory efficient management of temporal complex geo-objects such as triangle nets as depicted in Fig. 34.

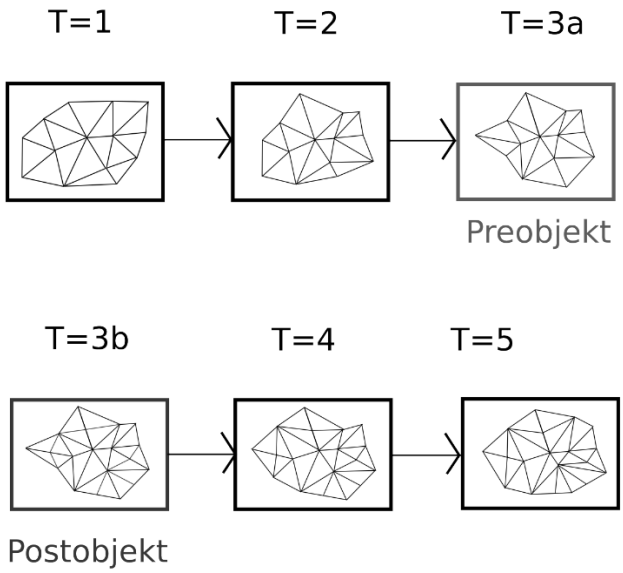


Fig. 34: Application example of the 4D model of Kuper
Source: (Kuper 2010, 42)

Fig. 34 shows an application example where a triangle net component moves through time (in five time steps). The meshing of the triangle net component stays constant until step *T=3a*. Until step *T=3a*, the triangle net component only changes the coordinates of some of its points (the point geometry) but not its meshing. In KUPER's 4D library for DB4GeO, this part is modelled as an `Object4D` with one `SpatialObject4D` with `start` value *T=1* and `end` value *T=3a*. Additionally, 14 `PointTubes` are created and also appended to `Object4D`. Every `PointTube` is populated with three points (one for each time step).

"Inside" step *3*, the overall geometry of the geo-object stays constant but the meshing changes between *T=3a* and *T=3b*. Subsequently, the `Object4D` has to be extended by a new `SpatialObject4D` with `start` value *T=3a* and `end` value *T=5*. If some support points of the triangle net component are identical between *T=3a* and *T=3b* (e.g. by their

70 The current implementation only provides support for temporal triangle nets, not for temporal tetrahedral nets.

geometry or by explicit IDs), then their point tubes can be reused for the points of the new `SpatialObject4D`. If we assume in this example that all point identities can be established, then the `Object4D` has to be extended by only four new `PointTubes`, since the post-object has four more points than the pre-object. All `PointTubes` are populated with three more points (one for each additional time step).

The example shows how the Temporal Joint Model makes it possible to manage the geometry of a Simplicial Complex through time, even if the meshing of the object's net representation changes at certain time steps. Hitherto, the Temporal Joint Model has been implemented by KUPER in DB4GeO for the management of triangle meshes.

2.5.5 Temporal Cell-Tuple Model for Spatio-Temporal-Attribute-Objects

(Polthier and Rumpf 1995) left the question open on how to model the transition between the `preObject` and `postObject` of a `TimeStep`. In such a transition the overall geometry stays unchanged but the mesh configuration of the geometric net changes. Though, KUPER's Temporal Joint Model is capable of managing point identity even after a reconfiguration of the mesh, the identity of the topological objects (simplices) is lost in such a case. This becomes obvious in the diagram of Fig. 33 that indicates that every `SpatialObject4D` has its own `TriangleNet4Ds` with their own *new* `Triangle4Ds`.

However, in practice, it is of little value to track identity of the meshing, since it is most often used as a meta-structure that is transparent to the user. But the situation is quite different regarding the management of the topology characterized by “big cells” (cf. Ch. 2.4.4). Big cells are explicitly modelled by the user and provided with certain distinct properties[71]. The temporal change of big cells is of particular interest for analytical purposes. Thus, it is of interest to provide a model that is capable of managing the spatio-temporal change of the topology of big cells.

RAZA and KAINZ (1999) utilize the concepts of object-oriented modelling (OOM) to generate a model for the management of *Spatio-Temporal-Attribute Objects* (STAO) in generic *temporal GIS* (*TGIS*). STAO is an aggregation of three objects of the types `SpatioTemporal`, `Attribute` and `LinearTime` (cf. Fig. 35).

71 For example in the geosciences domain with rock density, rock type, etc.

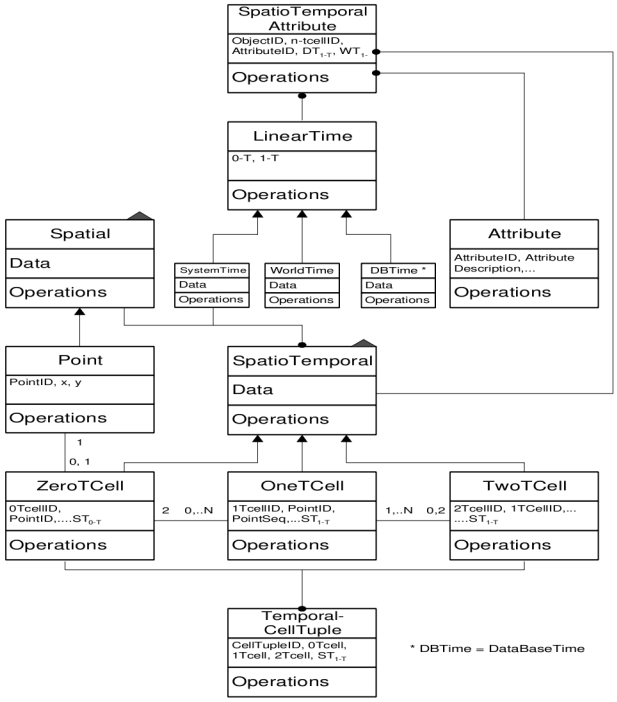


Fig. 35: Cell tuple based spatio-temporal data model
Source: (Raza and Kainz 1999, 21)

Spatio-temporal objects are represented by cell objects of dimension *0*, *1* and *2* (cf. with the classes `ZeroTCell`, `OneTCell` and `TwoTCell` in Fig. 35). For the modelling of the temporal topological relationships of cell objects, RAZA/KAINZ propose a *temporal cell-tuple structure* (cf. `TemporalCellTuple` class in Fig. 35). Therefore, they introduce the notion of *temporal cell-tuple*. First they develop a concept for the management of time of cell objects on the basis of type `SystemTime`[72] (*ST*). Two essential classes of type *ST* are `PointTime` (ST_{0-T}) and `IntervalTime` (ST_{1-T}) class (cf. Fig. 35). ST_{0-T} is a one-value point in time (value: `[TFrom]`), whereas ST_{1-T} is a two-value *time interval* (values: `[TFrom]` and `[TUntil]`). A `TemporalCellTuple` class is an aggregation of three cell objects (one for each dimension *0-2*) and of an object of type `IntervalTime` (cf. field ST_{1-T} of class `TemporalCellTuple` in Fig. 35). Thus, a temporal cell-tuple is defined as a cell-tuple with a *time interval*, i.e. a *life span*. Every temporal cell-tuple has a time instant when it is *born* and eventually a time instant when it *dies*. Furthermore, the model

72 Raza and Kainz use the terms world time (at which an event occurs in reality) and database time (at which an event is recorded in the database), which are equivalent to the terms valid time and transaction time that have been introduced by Snodgrass and Ahn (cf. Ch. 1.5). However, Raza and Kainz additionally introduce the notion of system time. System time is similar to database time with the difference that it does not model the time of the whole STAO but only of single geometric/topological objects.

of RAZA/KAINZ provides for every temporal cell-tuple an optional *child link* (to the temporal predecessor cell-tuple) and an optional *parent link* (to the temporal successor cell-tuple).

RAZA/KAINZ give an application example for their temporal cell-tuple structure (cf. Fig. 36).

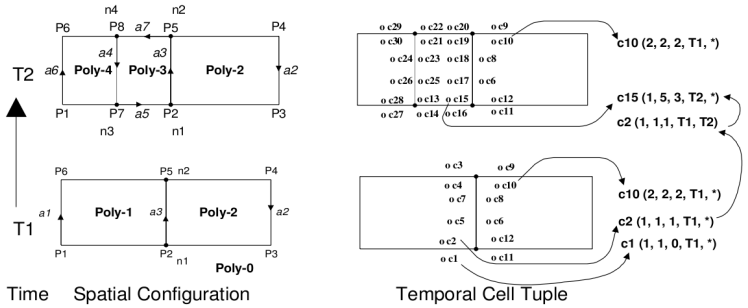


Fig. 36: Model of Temporal Cell Tuple by Raza and Kainz
Source: (Raza and Kainz 1999, 23)

The left side of Fig. 36 shows a *temporal cell complex*. Depicted are the polygons, arcs and nodes of the cell complex with their respective IDs. The cell complex makes a transition from *T1* to *T2*. Between *T1* and *T2*, new nodes *P7* and *P8* and a new arc *a4* are inserted that split the *Poly-1* polygon and the *a1* arc of *T1* into the two polygons *Poly-3* and *Poly-4* and the three arcs *a5*, *a6* and *a7* of *T2*. The life span of each cell in the cell complex is also recorded by system.

The right side of Fig. 36 illustrates the cell-tuple structure that describes the temporal cell complex that is depicted on the left side. Throughout the transition from *T1* to *T2*, the cell-tuples of *Poly-2* stay unchanged while the cell-tuples of *Poly-1* die and are replaced (`[TUntil]` of dead cell-tuples is set). Also new cell-tuples are added to the set (with existing `[TFrom]` value), and all required child and parent links are set.

Though, RAZA/KAINZ use an object-oriented approach to generate the STAO model, they use an SQL-database to make the STAO data persistent. Therefore, they finally transform the object-oriented model into a relational model for the actual processing. In the final relational model, the cell-tuples are stored as relational tuples, of which the combinatorial structure corresponds to BRISSON's model of cell-tuple structure (cf. Ch. 2.3).

The model is explicitly designed to manage geodata at a maximum of *2* dimensions of space. Hence, the concept is focused on polygons that are typical for traditional 2D GIS, not for geo-objects with a complex underlying network structure. Additionally, the concept makes no statements on the management of geometric change (interpolation) and on insertion of additional time steps. The insertion of additional time steps between existing time steps is theoretically possible in the model. However, RAZA/KAINZ do not define such an operation, thereby leaving open the questions of how to ensure consistency when inserting intermediate time steps. The concept of RAZA/KAINZ also does not support geometric holes in polygons. These issues are discussed in more detail and solutions are proposed in the process of designing a hierarchy-enabled spatio-temporal GMaps model for DB4GeO, beginning in the next chapter.

3 Design and Implementation of a Topology Module for the Modelling of Spatio-Temporal Objects

This section presents the design and implementation of a prototype *Topology Module* for the modelling of STO (with a G-Maps based kernel) in a Spatio-Temporal Database Architecture.[73] The module works as a plug-in on top of DB4GeO. This way, the classes of the Topology Module can reuse and extend the functionality of the already existing classes of DB4GeO. It is a recommendable and clean approach in order to reuse as much functionality as possible of an already application-proven API. This way, it can be avoided to "reinvent the wheel", and advantage can be taken of past and future bug fixes of the DB4GeO API. Additionally, the plug-in/module approach allows for experimental development of the G-Maps module without intervening into the already matured and well-tested source code of DB4GeO.

The architecture of the entire system is developed in compliance with principles of software engineering such as abstraction, decomposition, encapsulation, or modularity.[74] In a first step, a concept and implementation details for a module for the management of the topology of 3D geo-objects are elaborated. This is done with an emphasis on the question

73 The implementation of the module is realised in an individual, separated code trunk termed *GMapsDb3dModule*. The module is implemented on top of DB4GeO. Its name suffix "Db3dModule" is in compliance with the implementation guidelines for DB4GeO plug-ins. Only modules with such ending will automatically be identified as plug-ins by the DB4GeO server.

74 Principles as described in the SWEBOK (Abran and Moore 2001)

of how the architecture of DB4GeO can be extended by such a module. Second, it is evaluated how the 3D topology module can be extended to a system that is capable of modelling multi-representation and temporal topology of geo-objects.

The most of the presented and implemented source code is deliberately not optimized for runtime performance but for readability (and thus maintainability).

3.1 Basic Class Model

3.1.1 Overview of DB4GeO Kernel

In order to realise a "conceptual symmetry" between the Topology Module and DB4GeO (i.e. not to break already established concepts), it is inevitable to have a closer look at the design principles and the architecture of DB4GeO first. Following (Bär 2007), the architecture of DB4GeO, at its most abstract level, is a subdivision into three horizontal layers, see Fig. 37.

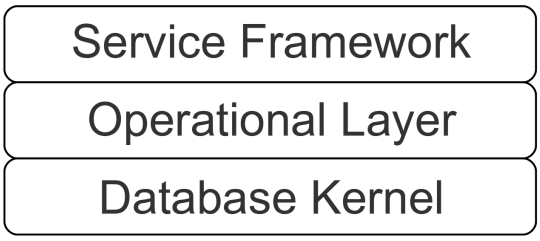


Fig. 37: Layers of software architecture of DB4GeO, according to (Bär 2007, 58)

The most basic (lowest) layer that is essential for all other layers is the *Database Kernel*, which includes a set of 3D geometric data types (and its methods) as well as topologically defined nets and a spatial index on the geometry data. The kernel provides only basic geometric operations. More complex operations that are built as compositions of basic operations are gathered in the *Operational Layer* (I/O operations are also part of the Operational Layer). Thus, the Operational Layer is based on the Database Kernel. On top, the *Service Framework* is based on the Operational Layer and exposes the complex operations to a communication network as a network service. The following section focuses on the Database Kernel layer.

As briefly mentioned in Ch. 2.1, the geo-object model of DB4GeO incorporates the notion of Simplicial Complex as a model for its geometric kernel. The *geometric primitives* of the Simplicial Complex model (for up to three dimensions) are *point*, *line segment*, *triangle* and *tetrahedron*, i.e. the 0-, 1-, 2-, 3-simplices. These geometric primitives are used as *simple geo-objects* in the DB4GeO kernel (see *geom* column in Fig. 38).

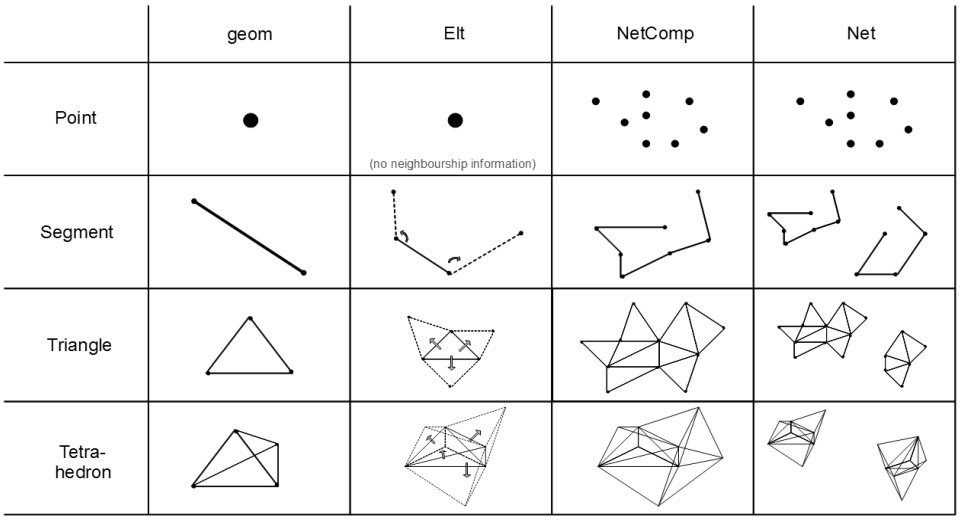


Fig. 38. *Geometry model of DB4GeO*

If a simple geo-object is part of a 3D object then it is a structural part of the 3D object's net and of one of its net components. As depicted in Fig. 38, for every primitive type, there also exists a corresponding net component type (see *NetComp* column) and a net type (see *Net* column).[75] If a simple geo-object is embedded into a net component, it also holds information on its direct neighbours. A simple geo-object enriched with neighbourhood information is a *net element* (see *Elt* column).[76]

3.1.2 Extended Module Functionality

While this model is suitable for many applications, it has also some shortcomings. For example, it is not possible to distinguish *regions* on a net component efficiently, i.e. it is not possible to distinguish (big) cells (e.g. faces/volumes) that are composed of multiple simple geo-objects (triangles/tetrahedra) but still smaller than a whole net component. From another point of view: it is not possible to define, which simple geo-objects (thematically) belong together within a net component; every simple geo-object yields its own separate cell (see Fig. 39, left of the arrow).

75 For example, an object of TriangleNet3D class has a reference to objects of TriangleNet3DComp class which in turn have references to objects of Triangle3D class.

76 For example, the net element of a Triangle3D object is a TriangleElt3D object.

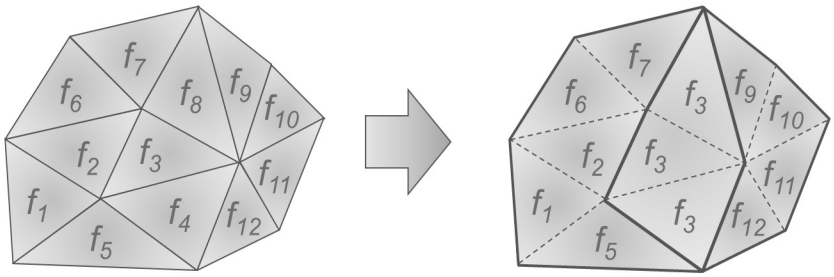


Fig. 39. *Left: simple geo-objects are separate cells; right: three simple geo-objects are combined to a cell*

The left side of Fig. 39 shows an example of the Simplicial Complex model, where, in a triangle net component, every simple geo-object is a distinct entity. As outlined in Ch. 1.3, many applications require a model that allows for managing distinct regions that consist of multiple simple geo-objects inside one net component, as depicted in the right side of Fig. 39. This applies also for volume models, using tetrahedral net components.

The inability to form regions is a general issue of the underlying model, predetermined by the navigational deficiencies of the employed Simplicial Complex model, as pointed out in Ch. 2.2. To support the creation/distinction of regions that cover multiple triangles or tetrahedra, following a naive approach, it would be sufficient to assign to every individual triangle or tetrahedron of the Simplicial Complex an attribute that determines to which region the respective simplex belongs. However, with this approach it would not be possible e.g. to navigate along the edge geometries of the regions efficiently (for example along the boundary surface between two volumes or along the boundary segment between two surfaces).

Right to the arrow of Fig. 39, the extended functionality of the G-Maps module for DB4GeO is shown, where a set of simple geo-objects can be combined to a cell. In the figure this is indicated by the common IDs of some simple geo-objects and by the tones of grey that related cells have in common.

The following list summarizes some basic design goals of the G-Maps module for DB4GeO (not considering the requirements for the management of hierarchy and temporality so far):

- It should be possible to generate the cell-tuple structure directly from the Simplicial Complexes of the DB4GeO kernel.
- The model requires to support *only 2D and 3D geometries*. A support of 0D is not necessary, since a net object consisting solely of disjoint/unconnected point geometries makes no sense in the considered application domains. Line segment nets of DB4GeO are also not supported by Topology Module since they are not in focus of this work.

- A pivotal functionality is the generation of a *boundary geometry*, i.e. a *polyline* (consisting of segments and points) in 2D (from a surface) and a *polyhedral surface* in 3D (from a volume) at any time.
- *Editing* of topological objects should be possible to some extent (e.g. inserting and deleting an edge in a face).
- The editing process (e.g. the insertion of an edge) shall be supported by the system with *traversal algorithms* on the object's net representation.[77]
- It should be possible to attach *properties* to any given (complex) cell.
- The editing of the underlying geometric objects of the triangular or tetrahedral net (through the shifting of point coordinates) that changes the topological configuration of the net needs not to be supported.
- The deletion of geometric objects of the net needs not to be supported.

In addition, it is a design goal to utilize the advantages of *information hiding* in the API layout. Classes shall be encapsulated in packages wherever possible by setting class constructors as package visible and by hiding the complex inner class structure to the API user. Ideally, the API user accesses only “high-level” classes on a higher model abstraction level, but has no contact with “low-level” classes that expose detail about the inner workings of the API. In particular in G-Maps programming, the inner structure can be notably fragile and needs some experience and extensive knowledge with Topology Module to handle it, so information hiding becomes particularly useful here.

In the chosen approach, the first step in designing the G-Maps module for DB4GeO is to define (on the most detailed architectural level) the mechanism by which the simple geo-objects of DB4GeO can be accessed and processed by the G-Maps module.

3.1.3 Spatial Cells as Wrappers for Simple Geo-Objects

The G-Maps module for the management of topology of 3D geo-objects relies on the notion of cells. As well known, cells are a means of describing a geo-object by decomposing it, using “other kinds of basic elements than just cubes”[78] (Mäntylä 1988, 72). By definition, a cell is a “finite regular polytope” (Weisstein 2014a). In our narrow perspective, a cell may be any spatial object that is composed of a set of connected simple geo-objects, e.g. a curved surface or a polyhedron. If the cell of a particular dimension is indicated, it is denoted as a *d-cell*, where *d* is the dimension. In the next sections, the following denominations of cells for dimensions *0-3* are used interchangeably:

- a *0-cell* shall be denoted as a *node*,

77 For example by finding the shortest path for an edge: if an edge is inserted, the shortest path should be found over the triangular or tetrahedral net. However, the concept of multi-level representations in the Topology Module is introduced in Ch. 3.1.8.

78 Meaning that for a cell any geometric form is possible that does not contain a hole.

- a *1-cell* as an *edge*,
- a *2-cell* as a *face*,
- and a *3-cell* as a *solid*.

The relations between the classes of simple geo-objects[79] and the cell classes[80] are depicted in Fig. 40.

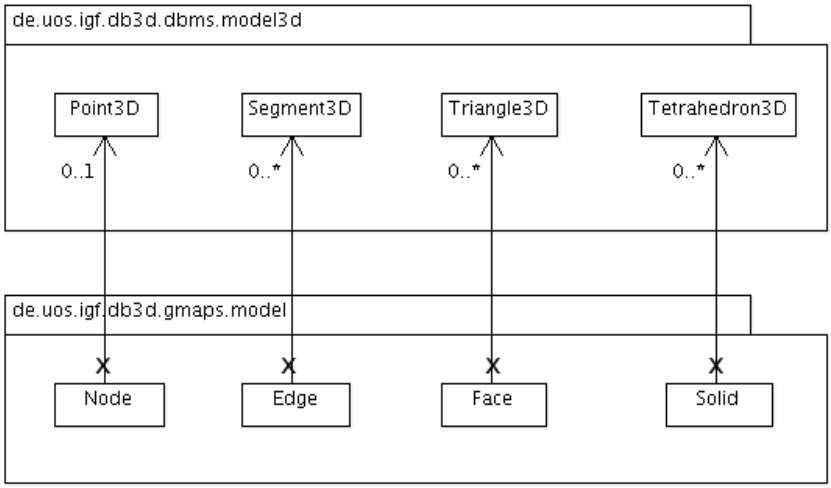


Fig. 40: Correlation between classes of simple geo-objects and cell classes

The associations between cells and simple geo-objects are directed from the classes of cells to the classes of simple geo-objects (unidirectional). The unidirectional associations are optional (see cardinalities in Fig. 40); the optionality theoretically[81] allows for the generation of an autonomous topological structure that exists independent of any geometry considerations.[82] Since the associations are unidirectional, there can only be references from cells to simple geo-objects but no references from simple geo-objects back to cells.[83]

The cardinality from any cell object towards the corresponding simple geo-objects is defined as *0* to *unlimited*, since a cell object can cover any number of simple geo-objects. A "big cell" solid can internally be composed of many tetrahedra, a face of many triangles and an edge of many segments. The only exception is node, where one node can

79 Implemented in the *db3dcore* API

80 Provided by the GMapsDb3dModule

81 "Theoretically", since the approach is not pursued further in this treatise.

82 The paper (Ellul and Haklay 2006) identifies the need for "geometry free" topology toolkits, as they are needed in chemistry for atomic field modelling, or in biology for protein modelling.

83 This is a necessary condition for the decoupling of *GMapsDb3dModule* from the *db3dcore* module, since this way, the classes of *db3dcore* are able to exist without "knowing" anything about the Topology Module.

geometrically only be described by no more than one point. The specific realisation of this concept in Java code is described in the subsequent section.

As the classes `Node`, `Edge`, `Face` and `Solid` are *cells*, they all realise a common interface `Cell` (see Fig. 41).

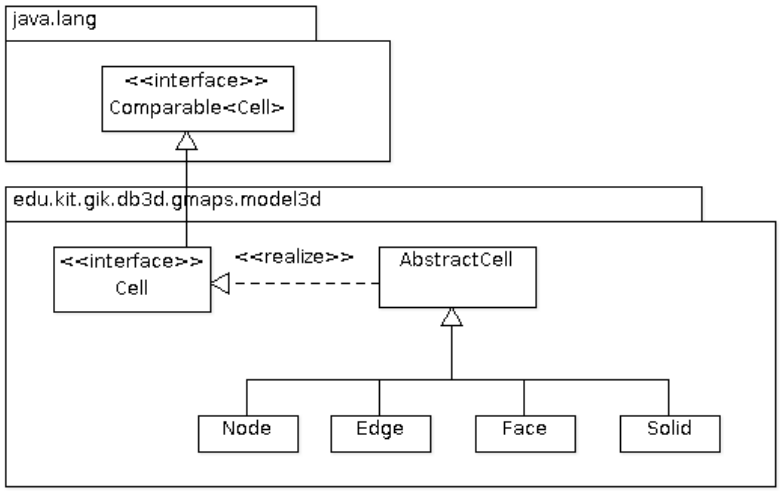


Fig. 41: Inheritance of cell classes

The `Cell` interface defines the methods that have to be provided by all implementing cell classes. The diagram shows that any cell has also to be a `Comparable` (`Comparable` interface is defined by the Java API, not by the G-Maps API). Since any `Cell` class is a `Comparable`, `Cell` objects may be stored in standard Java `Sets` or `Maps` and retrieved efficiently by their identifier.[84]

All methods of the `Cell` interface that use identical algorithms for all cell types (i.e. for the classes `Node`, `Edge`, `Face`, and `Solid`) are gathered (and pre-implemented) in the `AbstractCell` abstract class (which is a partial realisation of the `Cell` interface). Fig. 42 shows the methods that are defined by the `Cell` interface and the methods that are implemented by the `AbstractCell` class (to avoid double listing of the methods, the methods that are implemented by `AbstractCell` are not listed in the class diagram of `Cell` interface, though the `Cell` interface also requires the implementation of all the methods that are listed in the `AbstractCell` depiction).

84 More on the concept of the Java API interfaces Set and Map can be found in (Schildt 2011, 459 et seqq., 482 et seqq.).

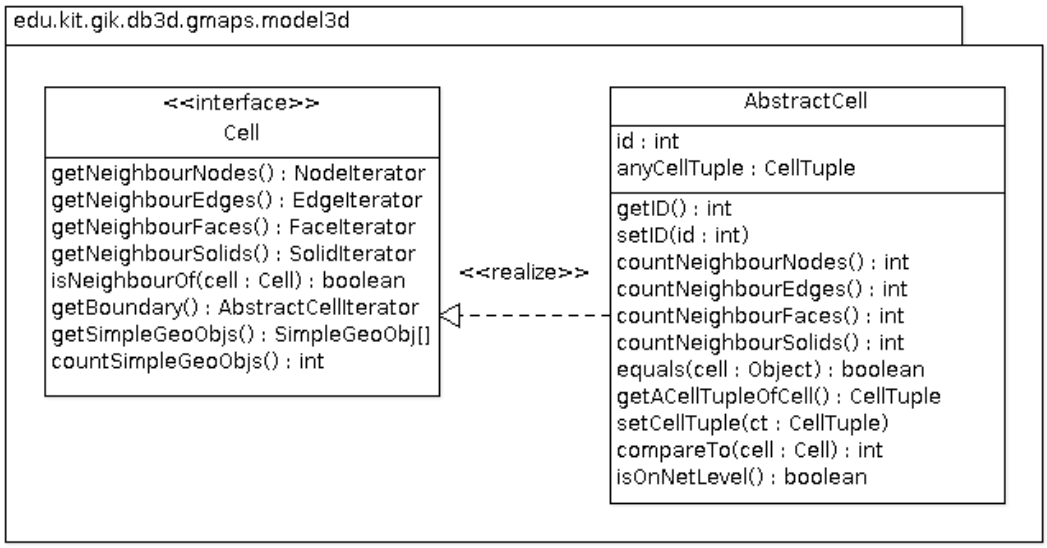


Fig. 42: Methods of cell interface and abstract cell

The `AbstractCell` class defines that all cells need to have an `ID` of type `int`. The `AbstractCell` class provides methods to read and to set/change the cell-ID as well as a method to compare two cells for equality (`equals`). The `equals` method first checks whether the object type is correct (type `Cell`). This is necessary since – as a requirement of the Java language – the parameter of `equals` method has to be of type `Object`). Second, the `ID`s of the cells are compared.[85] A `compareTo` method (which has to be provided by any cell as a requirement of the Java `Comparable` interface) is also implemented. In the implementation, only the cell-IDs are evaluated for comparison. Also, any `AbstractCell` can report whether it is on *net level* or on another level through its `isOnNetLevel` method. The specific concept of net level and of *hierarchy levels* that is used in Topology Module is covered in Ch. 3.1.8 and Ch. 3.5.

Furthermore, an `AbstractCell` provides methods to count the *number* of neighbour i-cells, with $\{i \in \mathbb{N} | 0 \leq i \leq 3\}$, by its `countNeighbour<cells>`[86] methods. We state that c is the cell for which to check the number of neighbouring cells and d is its dimension. A `countNeighbour<cells>` method returns the number of i-cells that are *incident* to c_d, except for $d \neq i$. The method returns the number of *adjacent* i-cells if $d = i$, instead.

All methods mentioned so far can be implemented uniformly for all cell classes (`Node`, `Edge` etc.) in the `AbstractCell` class.[87] Contrariwise, the methods listed in the diagram

85 Of course, technically, this is not a comparison for equality but for identity; but since it is guaranteed by the system that all cell-tuples inside the system are dissimilar, a real check for equality is not meaningful and thus can be implemented trivially by an identity check.

86 This “wildcard notation” shall summarize all methods that start their name with “countNeighbour”, regardless of which cell type completes the method's name (“Node”, “Edge”, “Face”, or “Solid”).

87 Thus simplifying maintaining and bugfixing efforts

of `Cell` interface have to be implemented individually by the `Cell` interface realisations (i.e. in `Node`, `Edge` etc.).

The `countSimpleGeoObjs` method returns the number of simple geo-objects that are *covered* by the cell. “Covered” comprises all simple geo-objects that form the geometric embedding of the boundary and the inside of a big cell. For example, a face cell returns the number of all triangles that are at the inner side of the boundary and all triangles that are inside the face. The same applies to all other cell types. In the case of a `Node` object, the returned number can only be *0* or *1*, in all other cases *0* to *unlimited* (cf. Fig. 40). The `getSimpleGeoObjs` method returns the simple geo-objects in an array of `SimpleGeoObj` class type.

Any `Cell`-implementing class has also to provide a `getBoundary` method. This method has to return an ordered list of cells that form the *boundary* of the cell c_d that the method is invoked on. The returned boundary i-cells are always of dimension $i = d-1$, except for $d = 0$ where always an empty set is returned. The return type of the `getBoundary` method is `AbstractCellIterator` (*cell iterators* are explained in Ch. 3.3.2 in detail).

Moreover, every implementation of the `Cell` interface has to supply `getNeighbour<cells>` methods that return for any cell a *list of neighbouring cells*. Analogue to the behaviour of the `countNeighbour<cells>` methods (see above), this methods return incident cells if $d \neq i$ and adjacent cells if $d = i$. As valid for the `getBoundary` method, the cells are returned in an *ordered sequence*. In fact, the `getBoundary` method is internally implemented as an invocation of the appropriate `getNeighbour<cells>` method, depending on the dimension of the respective cell:

- `Edge` class implements `getBoundary` as an invocation of `getNeighbourNodes` method,
- `Face` class implements `getBoundary` as an invocation of `getNeighbourEdges` method,
- `Solid` class implements `getBoundary` as an invocation of `getNeighbourFaces` method.

The `getNeighbour<cells>` methods return different types of customized *cell iterators*; these iterators will also be explained in Ch. 3.3.2.

Finally, the `isNeighbourOf` method returns a `boolean` value that indicates whether the given parameter of type `Cell` *is a neighbour* of the invoking cell object (actually this is also implemented through an invocation of the appropriate `getNeighbour<cells>` method and a subsequent check for equality).

3.1.4 Tuples of Spatial Cells

The advanced navigation requirements that are needed by the cell methods introduced in the last chapter can be satisfied by the cell-tuple structure, as considered in Ch. 2.3.

Therefore, it is necessary to establish a path from any cell to a cell-tuple/dart[88]. In order to provide a path from cell to cell-tuple, the `AbstractCell` class has a reference to a cell-tuple (cf. Fig. 42 and Fig. 43).

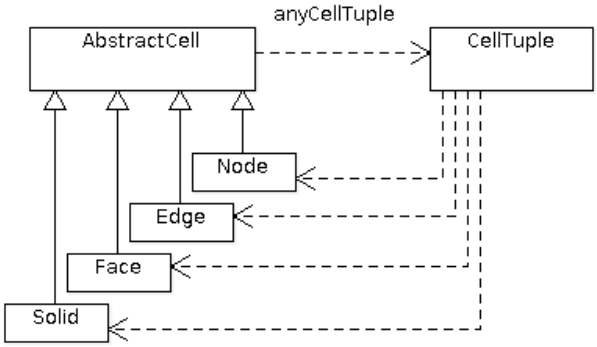


Fig. 43: References between abstract cell, cells and cell-tuples

The reference is realised as class attribute `anyCellTuple`, which represents an arbitrary cell-tuple of the cell ("of the cell" means that the respective cell-tuple *back-references* the cell). Thus, any cell (e.g. a node or an edge) is able to provide a valid cell-tuple that "belongs" to the cell. On the other hand, any cell-tuple has separate references to all its cells (cf. Fig. 43 and Fig. 44).

The class attribute `anyCellTuple` is of type `CellTuple`. Referring to the definitions of cell-tuple structure and G-Map of BRISSON and LIENHARDT, such as those indicated in Ch. 2.3, the class diagram of Fig. 44 serves as a basis for the `CellTuple` class model of the Topology Module.

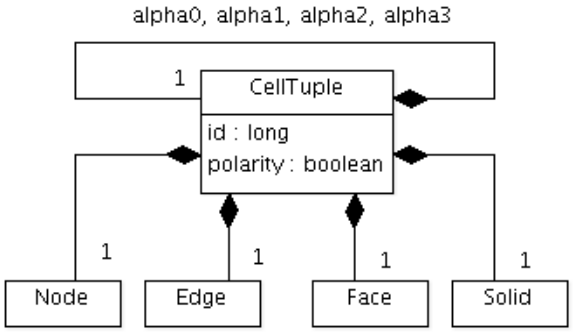


Fig. 44: Model of CellTuple class

An object of `CellTuple` class is a composition of incident i*-cell* objects of dimensions $i, \{i \in \mathbb{N} | 0 \leq d \leq 3\}$ [89]. A `CellTuple` can be interpreted as the instantiation of a path in an

88 For practical reasons, the terms "cell-tuple" and "dart" are identified and used interchangeably in the following.

89 i.e. of objects of the classes Node, Edge, Face and Solid

incidence graph of a certain cellular representation (cf. Ch. 2.2). A `CellTuple` is explicitly modelled as a *composition* of cells, thus it is only valid if it provides (*not*-`null`)-references to *all* its i*-cells*. This is a pivotal consistency requirement for the model, since following algorithms will rely on this assumption.

Following the definition of G-Map *darts* (cf. Ch. 2.3), a `CellTuple` object is also a composition of all its i*-adjacent* `CellTuple` objects. Thus, a `CellTuple` object is also not valid if it misses to provide a reference to any of its adjacent `CellTuple` objects. The i-adjacent `CellTuple` objects can be attained through its *adjacency* or *involution associations*, which are denoted as `alpha0`, `alpha1`, `alpha2`, and `alpha3` class fields (cf. top of Fig. 44).

The presented adjacency model of cell-tuples can be interpreted as a graph representation (cf. Ch. 2.2). However, a design goal is to ensure through the entire modelling process that a *transformation* into a *relational representation* is possible at any point in time without loss of consistency. In order to be able to create certain cell configurations in the relational representation, the introduction of a `polarity` and additional *consistency checks* on the overall cell-tuple structure are necessary, which is explained in more detail in Ch. 3.1.5 and (Butwilowski 2007, 71 et seq.). The consistency of both associated structures is respected at any given point in time, and thus, a model transformation can also be performed at any given time (a “switch” between the two concepts is possible).

The `CellTuple` class also includes an *identification number* (`id` class attribute)[90]. The IDs of cell-tuples are *unambiguous* throughout a net component. Conversely, cell-tuple IDs are ambiguous in a viewpoint across multiple net components. This design decision is based on the specification of `NetComp` which are always disjunct by definition. Thus, each net component can be described by a separate G-Map and each G-Map employs its own ID management. One 3-dimensional G-Map corresponds to one `NetComp` and vice versa (3-G-Map ↔ `NetComp`).

Furthermore, the `CellTuple` provides an `isAtFaceBoundary` method that indicates whether the cell-tuple (d_a) is a cell-tuple at the inner side of a 2-cell universe boundary. d_a is at the inner side of a 2-cell universe boundary if $\alpha_2(d_a)=d_U$, where d_U is a cell-tuple that belongs to a 2-cell universe. This method is extensively used by the `OrbitIterator` class (see Ch. 3.3.1).

Due to the principles of information hiding, the `CellTuple` class is encapsulated in the package by setting the class constructors and methods only as *package visible*, so that the inner cell-tuple structure is hidden to the API user. The API users are only able to access the various `Cell` classes but have no contact with the cell-tuples themselves since this is a fragile structure that needs to be protected by the API.

90 The ids of objects of CellTuple class are always greater than or equal to 0.

So far the class model of cells and cell-tuples, as it is used by the Topology Module, has been outlined. The next section will highlight, what specific properties of geometric space and model restrictions result from the adopted design.

3.1.5 Basic Properties of the Utilized G-Maps Approach

Properties of Utilized G-Maps Approach

The model of `CellTuple` class in the described configuration leads to notable properties of the utilized G-Maps approach:

- As the *d-cell* classes are modelled explicitly, the maximum processable dimension of a *d-cell*, provided by the API, is *3* (`Solid`).
- A valid `CellTuple` object has always to reference exactly four *d-cell* objects. Thus, the only utilized G-Map type, notwithstanding the dimensions of the managed cells, is always the *3-G-Map* (cf. Ch. 2.3).[91]
  - In the case that only e.g. 2-cells shall be managed in a certain application scenario, to keep compatibility,
    - these 2-cells will “forcefully” be embedded in a 3-G-Map. In this case, the solid cell that is associated to all of the cell-tuples, is defined as the *universe* solid (S_U).
    - The cell-tuples are not duplicated to generate mirroring cell-tuples for α_3 -involutions, instead
    - all α_3 -involutions for all darts of the 2-cell are *reflective*, i.e. $\alpha_3(d)=d$ ($\alpha_3(d)=id$).
    - Cell-tuples that are lying in the *outer void* of 2-cells (face universe: F_U) are also provided in the case that only 2-cells are modelled. Thus, for stand-alone 2-cells $\alpha_2(d_b)=d_b'$[92] is always *true* and $\alpha_2(d_b)=d_b$ is never the case, where d_b is a cell-tuple at the inner side and d_b' a cell-tuple at the outer side of the boundary of a face.
- If modelling 3-cells, a solid universe is attached to the outer void of the 3-cell. This means that at the boundary to the outer void, outer cell-tuples are provided, i.e. that in these cases, α_3 is always defined as $\alpha_3(d_b)=d_b'$, where d_b is a cell-tuple at the inner side and d_b' a cell-tuple at the outer side of the boundary of a solid.
- If a surface is used as a part of a solid, then
  - U_F cannot exist (in such cases, there is no face universe),

91 This keeps the overall model comparably simple, since only one type of G-Map has to be managed, thus less code has to be implemented and less exceptional states have to be considered.

92 With d_b being an inner cell-tuple at the boundary of a 2-cell

  - all cell-tuples of a face have to be duplicated to generate mirroring cell-tuples for α_3 -involutions.
- Since universe definitions are used for 2- and 3-dimensional cells, there are two "flavours" of `Face` and `Solid` class: faces and solids that are marked as *universe* and such that are not ("normal" cells).

The forceful embedding of 2-cells into a 3-G-Map is essential in order to simplify the class model and the processing algorithms. As a result, there are several cases where only one algorithm is needed instead of employing a case distinction for 2-G-Map algorithms and 3-G-Map algorithms. Some of these cases are presented below, when the algorithms of the Topology Module are examined in detail.

Cell Property Identifies Universe Cells

To mark faces and solids as cells that represent the universe, the classes `Face` and `Solid` obtain an additional `isUniverse` class property of `boolean` type (see Fig. 45).

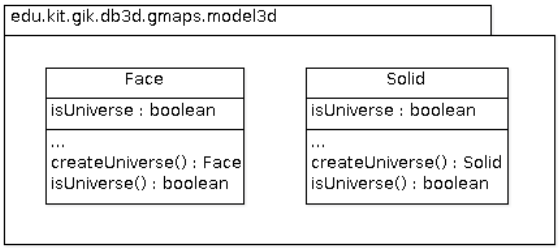


Fig. 45: Face and Solid class provide the possibility to create universe cells

The information, whether a given cell is a universe or not, can be accessed through the `isUniverse` method, which is also available for both classes (the method returns the value of the `isUniverse` instance property).

New universe cells can be created only through the `createUniverse` methods.[93] These methods are *object factories* that internally instantiate a new universe cell (i.e. a cell class with `isUniverse` class attribute pre-set to `true`) of the respective dimension and provide them as the return value.

The Topology Module allows to instantiate multiple universe cells in each dimension – i.e. multiple universe faces and multiple universe solids. The usefulness of multiple universe cells is discussed in a section below.

93 For the convenience of the class user, the *createUniverse* methods are *static* methods that follow the factory method pattern.

Non-Manifold Set-Ups

G-Maps generally model manifold geo-objects. As LÉVY points out in his thesis (Bruno Lévy 2000), G-Maps can also be used to model non-manifold situations in some special cases which are called *Cellular Quasi-Manifolds*. LÉVY gives some examples of such set-ups. For example, he shows that it is possible to "glue" three (or more) faces to a non-manifold 3D "fan" with G-Maps (Bruno Lévy 2000, 63 et seqq.). To do so, LÉVY first duplicates all cell-tuples of the faces that already exist in 2D to generate mirroring cell-tuples for α_3 -involutions and then assembles all faces to one face net. This set-up only works if the 2-cell universe is not modelled, i.e. if $\alpha_2(d_b)=id$. The explicit integration of a face universe (as realized in the Topology Module) prohibits the modelling of non-manifold fan set-ups. A problem is, for example, that this would lead to undefined intersections of the universe faces (see Fig. 46).

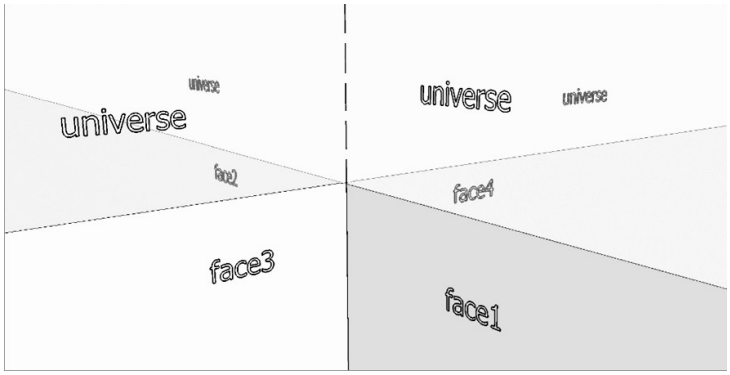


Fig. 46: Non-manifold face fan in 3D and the intersection of universe faces

The API has to prevent the user from constructing such a topological set-up (more details on this are given in later sections).

However, as long as the API user intends to utilize the Topology Module not in stand-alone mode but in combination with the DB4GeO Kernel geometry, this issue cannot ever occur.[94]

Minimal Cell Configurations

The minimal cell configuration that is possible in the presented model[95] is

94 The Topology Module constructors that base on the DB4GeO Kernel, will reuse the already existing construction algorithms of the DB4GeO Kernel (more on this in Ch. 3.2). Therefore, the constructors are limited by the constructing capabilities of the kernel algorithms. The model of and the construction algorithm for triangle nets in DB4GeO is not designed for non-manifold triangle set-ups and thus does not allow to construct such.

95 "Presented model" means particularly: with polarity; always 3-dimensional; mapping between graph and relational representation always possible

- a face with one edge and one node ("racket", cf. Fig. 47, left), embedded in F_U *and* S_U, for *2-cells*, and
- a solid with two faces, one edge and one node ("nutshell", cf. Fig. 47, right), embedded in S_U, for *3-cells*.

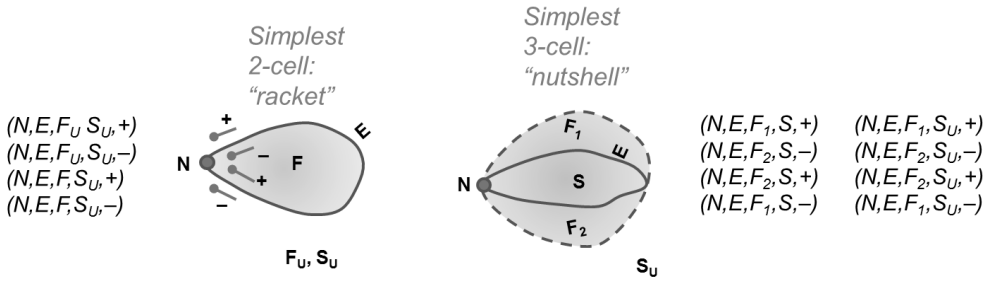


Fig. 47: 3-G-Map minimal cell configurations: 2-cell (left), 3-cell (right)

It is considered here as the *minimal configuration*, since a configuration with any cell less of any dimension would lead to an inconsistent set of cell-tuples in *relational representation*, i.e. a set where the cell-tuples are not *unique* by their tuple elements any more (cf. Ch. 2.3 and (Butwilowski 2007, 71 et seq.)).

In this context, the polarity is an important feature in this set-up to keep the cell-tuple structure consistent in relational representation. The switch operation in relational representation only works if the cell-tuples of a cell-tuple structure are unique throughout the structure. They would not be unique any more in the examples of Fig. 47 if the `polarity` property would be omitted. Then there would be multiple equal cell-tuples in the set which would break the switch operation. Such situations cannot only occur in this simple configuration but can also reoccur in particular set-ups of complex geo-objects.[96] Since the internal *graph representation* of the Topology Module should be able to produce a consistent external *relational representation* of the cell-tuple structure at any time instant, the `polarity` property has always to be maintained by the kernel of the Topology Module.

Though, the Topology Module can be used stand-alone to purely model the cell-tuple structure without a connection to a geometry, the common mode of operation is *in combination with the Geometry Model* of the DB4GeO Kernel. In that case, the Topology Module has to support the geometry data structure of the DB4GeO Kernel. Then, the actually minimal cell configuration is predetermined by triangle type for 2-cells and by tetrahedron type for 3-cells (see Fig. 48).

96 Examples of such set-ups can be found in (Butwilowski 2007, 71 et seq.)

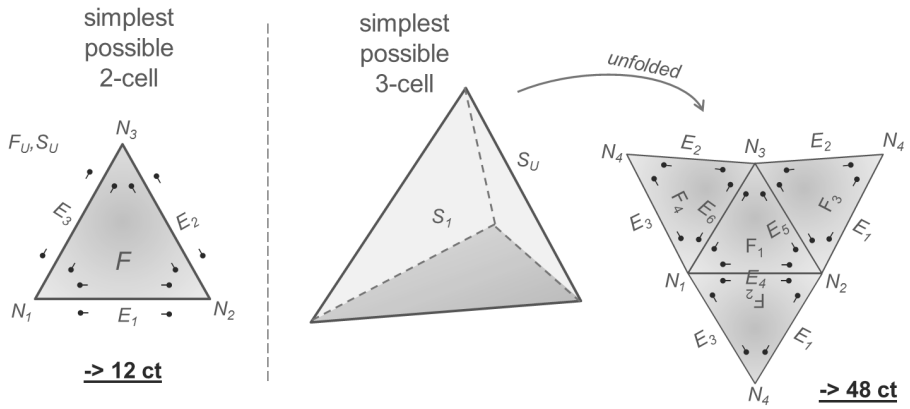


Fig. 48: Most simple possible cells in DB4GeO Topology Module: 2-cell (left), 3-cell (right)

In the minimal cell configuration for 2-cells, the G-Map consists of 12 cell-tuples (see left depiction of Fig. 48). Six cell-tuples lie inside the triangle. The 2-cell of these cell-tuples is always F, their 3-cell is always the solid universe S_U. The other six cell-tuples lie outside the triangle in the 2D universe. Therefore, the 2-cell of these cell-tuples is always the face universe F_U, their 3-cell is also S_U. All inner and outer cell-tuples are connected through α_2 -involutions. All cell-tuples are reflective in their α_3 -involution.

The minimal cell configuration for 3-cells results in a G-Map with 48 cell-tuples (see right depiction of Fig. 48). 24 of these 48 cell-tuples lie inside the tetrahedron and reference cell S as their 3-cell. The other 24 cells lie outside the tetrahedron in S_U. All inner and outer cell-tuples are connected through α_3 -involutions. There are no α_2 -involutions that could lead to a face universe (F_U), therefore F_U is not modelled.

3.1.6 Nets of Spatial Cells and Cell Net Builder Architecture

The Topology Module provides a framework that uses the above described notion of `CellTuple` class to create cellular complexes from existing *3D objects*[97]. The topological structure of these cellular nets is internally managed in G-Maps. After a cellular net has been created by the means of the Topology Module, the module can provide valid `CellTuple` objects that describe the topological structure of the cellular net.

97 Defined in *db3dcore*

Cell Net Builder Architecture Class Model

For the consistent creation of cell nets, the *net builder architecture*[98] is extended by the Topology Module as exemplary depicted for the 2D case triangle and face net builder[99] in Fig. 49.

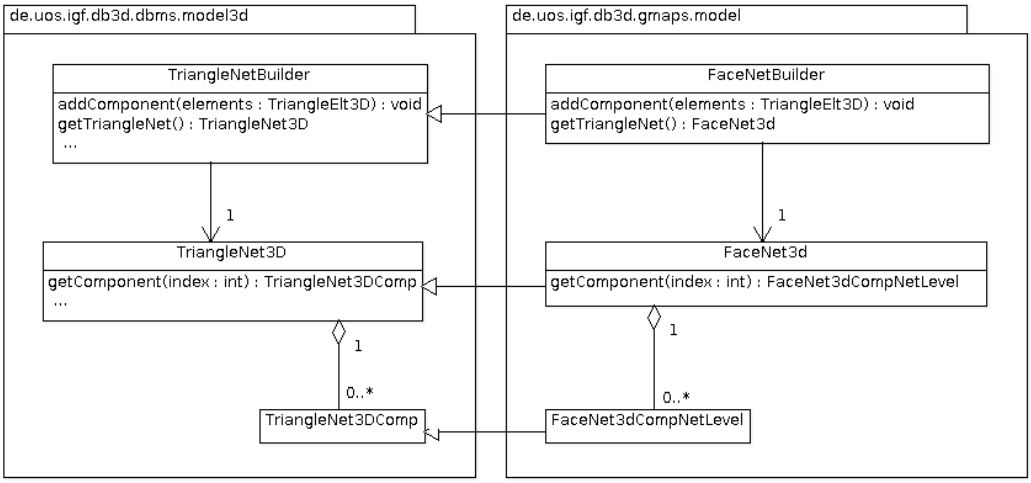


Fig. 49: Face net builder architecture as an extension of the triangle net builder architecture of DB4GeO

The diagram indicates the procedure for the creation of valid net objects. A *builder object* (e.g. an object of `TriangleNetBuilder` class, see left side of Fig. 49) is used to construct a consistent net (e.g. a `TriangleNet3D` object). The *builder design pattern* is used here for consistent ID management, arrangement of geometry elements and spatial index construction. A builder object "consumes" geometry objects, analyses their geometric configuration in space, constructs the appropriate net components and aggregates them in a net.

Through the invocation of the `getTriangleNet` method on the net builder object, an object of `TriangleNet3D` class is returned. The `TriangleNet3D` class is then used to retrieve a triangle net component (object of `TriangleNet3DComp` class) through the invocation of the `getComponent` method on the triangle net.

The G-Map topology module bases on this architecture and extends the Simplicial Complex net builders by *cell net builders* (in the example case by a *face net builder*, see right side of Fig. 49). In analogy to the triangle net builder, a face net builder returns the face net (object of `FaceNet3d` class) through an invocation of its `getTriangleNet` method and the face net in turn returns a required face net component by the invocation of its `getComponent` method.

98 Defined in *db3dcore*

99 The following examples describe only the situation for triangle nets, but the examples are analogous for the case of tetrahedron nets.

Cell Net Builder Instantiation Example

As a result of using this architecture, it is possible to instantiate a cell net (and thus a G-Map) in a way that is strongly analogous to the instantiation process of a complex geo-object in *db3dcore* (see Listing 3 for an example).

```
1. ScalarOperator sop = new ScalarOperator();
2. FaceNetBuilder build = new FaceNetBuilder(sop);
3. TriangleElt3D tri1 = new TriangleElt3D(
           new Point3D(2.0, 1.0, 1.0),
           new Point3D(2.0, 3.0, 1.0),
           new Point3D(1.0, 2.0, 1.0), sop);
4. TriangleElt3D tri2 = new TriangleElt3D(
           new Point3D(2.0, 1.0, 1.0),
           new Point3D(2.0, 3.0, 1.0),
           new Point3D(3.0, 2.0, 1.0), sop);
5. build.addComponent(new TriangleElt3D[] { tri1, tri2 });
6. FaceNet3d net = build.getTriangleNet();
7. FaceNet3dCompNetLevel comp = net.getComponent(0);
8. Face face1 = comp.getFace(1);
9. Face face2 = comp.getFace(2);
```

Listing 3: Example of an instantiation of a cell net (and G-Map)

Listing 3 shows an instantiation example, where first, two triangles are created (lines 3 and 4). The triangles are then added to a face net builder in the `addComponent` method (line 5), where the main build process takes place, and where internally, first, a triangle net is created and, second, a face net is deduced from the triangle net. Actually, in this example, two faces[100] are generated from two triangles. Inside the `addComponent` method, the G-Map is generated through an evaluation of the structure of the complex geo-object. As a result of the process, the API user has access to a face net (line 6), a face net component (line 7) and the actual faces (lines 8 and 9). The internal algorithm inside the `addComponent` method that creates the G-Map is discussed in detailed in Ch. 3.2.

Note that the methods `getTriangleNet` of `FaceNetBuilder` and `getComponent` of `FaceNet3d` class *override* the respective methods of `TriangleNetBuilder` and `TriangleNet3D` class. This approach has the advantage that a `FaceNetBuilder` object can be used exchangeably everywhere in the API where a `TriangleNetBuilder` object can be used. Similarly, an object of `FaceNet3d` can function as a substitution for an object of `TriangleNet3D` class wherever sensible and useful.

Actually, the returned face net component is a face net component at net level (class `FaceNet3dCompNetLevel`). The concept for handling levels of detail is introduced later in Ch. 3.1.8. For the sake of simplicity, for now the net level component can be considered as a cell net component that exactly reflects the object's meshing in a cell-tuple structure.

100 Two faces at net level (more information on net level in Ch. 3.5)

Fitting into Main Inheritance Hierarchy

Fig. 50 illustrates the idea of how the cell net classes fit into to the main inheritance hierarchy and dependency relations.

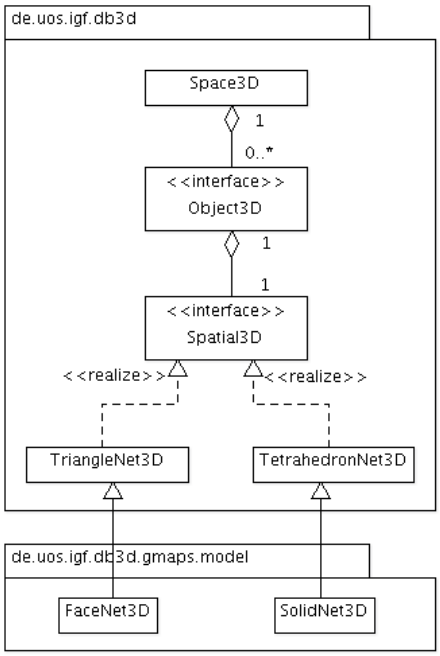


Fig. 50: Adaptation of cell nets (face net and solid net) into the central class inheritance hierarchy

Fig. 50 presents a condensed overview of a central part of the class inheritance hierarchy of the DB4GeO Kernel. It begins with a `Space3D` at the top. `Space3D` is mounted to a `Project` (not depicted in the diagram), and thus is one of the "upper", i.e. one of the entry objects of *db3dcore*. A `Space3D` can have an arbitrary number of `Object3D` (which is the class that generally represents any kind of 3D object in DB4GeO). A 3D object in turn always consists of exactly one object of `Spatial3D` type. Since `Spatial3D` is an interface, the actual spatial part of a 3D object can vary, but is always a net object. Several different realisations of `Spatial3D` can be used as the spatial part, like a `TriangleNet3D` or a `TetrahedronNet3D`. By using the Topology Module, also a `FaceNet3d` or a `SolidNet3d` can be the spatial part of a 3D object since the classes indirectly realise the `Spatial3D` interface.

The chosen approach has several advantages:

- The G-Map-enabled cell net components can also be used in place of simplex net components throughout the DB4GeO APIs by using the (*implicit*) *class cast* technique (which is common practice in Java programming).

- Since a G-Map cell net component is a net component, all methods of the simplex net component can still be used on the cell net component (reuse of existing methods).[101]
- Code duplication is reduced, since much of the already existing code of the DB4GeO Kernel does not need to be rewritten. Methods of the DB4GeO Kernel that are not overridden by the Topology Module, are easier to maintain, since bugfixes in the DB4GeO Kernel are directly available in Topology Module.

Indices of Cell Net Components

When executing the build up process as presented in Listing 3 (line 7), a face net component is returned that is defined as a composition of its *d*-cells and cell-tuples (cf. Fig. 51).

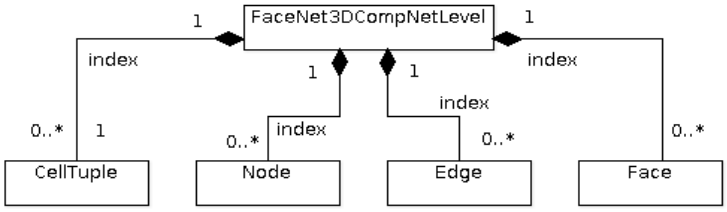


Fig. 51: FaceNet3DCompNetLevel as composition of indices of CellTuple objects and cells

Face net components consist of separate indices for faces, edges and nodes. Solid net components have an additional separate index for solids. The *d*-cell indices are needed for fast retrieval of cells by their cell ID. A face net component does not need a solids index, since it always references only exactly one solid which is the universe solid.

Additionally, an index for `CellTuple` objects is also part of the field of a cell net component. The `CellTuple` index serves for fast retrieval of `CellTuple` objects by their `CellTuple` ID. All cell-tuples of a cell net component are added to the cell-tuple index during the component construction process. The set of cell-tuples of *one* cell net component constitute a cell-tuple structure/G-Map. All *d*-cells in the cell net component indices belong to one *contiguous* component. A cell net component is part of a cell net that can have several *disconnected* cell net components.

Orientability Check of Net Component by Polarities

In the case that the Topology Module operates on top of the DB4GeO Kernel, the `polarity` property of the cell-tuples has an additional important functionality. The DB4GeO Kernel provides the possibility to *orientate* triangle net components. After a triangle net component has been imported into DB4GeO, it is possible that the triangles of

101 For a final, consistent implementation, all methods of the superclasses have to be reviewed on the need to be overridden by methods of the subclasses. However, this is not part of this work, and therefore not yet completely accomplished.

the component are *not homogeneously orientated*. The orientation of a triangle is given in DB4GeO by the *sequence of numbering of its support points*. The orientation is important, since e.g. the *direction vector* of a triangle is deduced from the sequence of numbering. If the triangles of a net component are not homogeneously orientated, the overall triangle net component is inhomogeneous in its orientation, which is not wanted in many applications.

In the case of an inconsistent orientation, the `TriangleNet3DComp` class of the DB4GeO Kernel provides the `makeOrientationConsistent` method which rearranges the triangles' orientations to realize a homogeneously orientated triangle net component. This is always done in the aftermath of an import operation. But not all contiguous surfaces are *orientable*. A well-known example of *non-orientable* surfaces is the MÖBIUS *strip*, which has been discovered by MÖBIUS and LISTING (Weisstein 2015b). The DB4GeO Kernel does not consider non-orientable surfaces and thus behaves incorrect in such cases. It tries to orientate a non-orientable triangle net component and terminates the operation in constant time. However, after the attempt, the `isOrientationConsistent` method of the `TriangleNet3DComp` class of the Kernel reports that the non-orientable surface *is orientated*, which obviously is incorrect.

In most industrial applications as well as in the standard ISO 19107, non-orientable surfaces are not applicable (cf. (Andrae 2008, 114)). Since ISO 19107 defines the foundations of CityGML (see Ch. 1.3), non-orientable surfaces are also not applicable in CityGML. As it is one goal of the Topology Module to prepare the basis for CityGML integration, it is useful if the API is able to identify and report non-orientable surfaces.

In the process of face net component creation (as described in Listing 3), all underlying cell-tuples and all polarity values are also internally built up and set (see Ch. 3.2 for details). After the process has finished, it is easy to identify a non-orientable surface with the help of the polarised cell-tuple structure. In a polarised cell-tuple structure, the polarities of all pairs of cell-tuples that are connected through an α_2-involution, must be *opposite* if the surface is orientable (cf. (B. Lévy and Mallet 1999, 8)). If the surface is non-orientable, polarities of α_2-cell-tuple-pairs *cannot* always be opposite. This verification method is used by the `isOrientable` method of `FaceNet3dComp` class, which is presented in Listing 4.

```
1. public boolean isOrientable() {
2.   for (CellTuple ct : this.cellTupleIndex.values()) {
3.     if (ct.polarity == ct.alpha2.polarity)
4.       return false;
5.   }
6.   return true;
7. }
```

Listing 4: Implementation of isOriented method of FaceNet3dComp class

In line 2 of Listing 4, all cell-tuples of the face net component are fetched and iterated. In line 3, it is checked whether there is any cell-tuple polarity that is equal to the polarity of

its α_2 -cell-tuple (*pair check*). If so, the surface is non-orientable, thus `false` is returned by the method. Otherwise if all α_2 -pairs of the component have opposite polarities, the surface is orientable and `true` is returned.[102]

3.1.7 Handling Holes in Cell Net Components

In many application cases of geodata modelling, it is important to provide the possibility to model *holes* in the geo-object's geometry, i.e. to model holes in surface and volume nets. In subsurface modelling for example, holes are useful to model *crevasses* (in surface nets) or enclosed *caverns* (in volume nets).

In the presented model, holes are treated as *universe cells*. Every universe cell has its own cell ID. Not only holes but also the *outer void* itself is modelled as a universe cell with a specific ID. Any cell net component is embedded in the outer void (cf. Ch. 3.1.5). If the cell net component is modelled with a hole inside, then the hole is created as an additional universe cell with an ID that is different from the ID of the universe cell that represents the outer void. See Fig. 52 for an example of a face net component that is placed into the outer void and which has one hole inside.

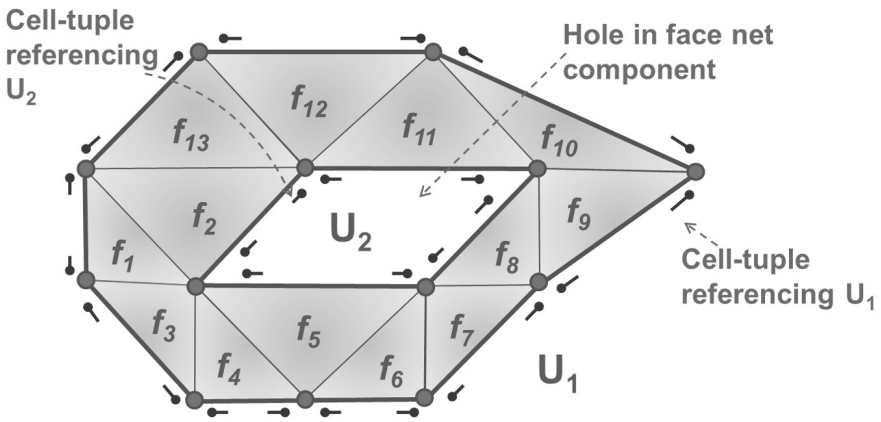


Fig. 52: Face net component in outer void (U_1) with additional inner hole (U_2)

In Fig. 52, the outer void is represented by the universe face U_1. A face net component, consisting of some triangles is embedded into U_1. Cell-tuples that reference face U_1, i.e. cell-tuples that lie in the outer void, are depicted in the figure. The face net has a hole inside of it that is represented by the universe face U_2. A set of cell-tuples (in this case eight of them) reference the inner universe face U_2 (these cell-tuples are also depicted in the figure). Every inner hole of the face net results in a new universe face with a distinct ID (though, in the example set-up, there is only one hole). Since the face net at this level is a

102 The implementation, presented in Listing 4, involves many unnecessary double checks, which degrades runtime. The runtime could be improved by an “already-visited” list, which on the other hand impairs memory usage.

direct representation of the underlying Simplicial Complex, all faces are modelled as triangles. However, this is not true for the universe faces. They can have any polygonal shape that is needed to represent the particular part of the void.

Having a clear distinction between “normal” cells and cells that represent parts of the universe, it is straightforward to implement algorithms that retrieve all cells of the cell net component's boundaries. A boundary retrieving algorithm returns – for example – all edges of a face net component that have contact to the universe. This is a collection of all edges that lie at the outer void *and* all edges at the inner holes of the face net component. However, the boundary retrieving algorithm needs to identify *all* boundaries of the cell net component, since, if a component has multiple holes, it also has multiple boundaries. But it is a priori unknown where the boundaries or where the universe cells are. It would be necessary to iterate over all cells and check every cell for whether it is a universe cell in order to sort out all universe cells.

To solve this problem, all universe cells that are created, are registered in additional separate indices already during construction of a cell net. Each component of the cell net maintains its own index of all universe cells of the respective component (see Fig. 53).

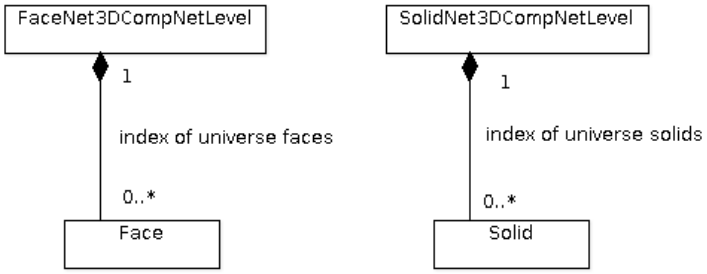


Fig. 53: Indices of universe cells are part of face net component and of solid net component

On the one hand, a face net component needs only an additional index on universe faces. There is no need for an index for universe solids in a face net component, since in the case of a face net component, there is always only exactly one solid universe, which is the solid outer void. Inner solid holes are not possible in face nets.

On the other hand, a solid net component needs only an additional index on universe solids. There is no need for an index on universe faces in a solid net component, since in the case of a solid net component, it is not possible to create a universe face that represents the 2D outer void or a 2D inner hole. This is not possible, since, when creating a solid net component, a face net component is automatically generated as the boundary of the solid net component. The face net component boundary forms the closure of the solid net component. Therefore, it has no outer void and no inner holes. All further steps of editing the faces of a solid are monitored by the editing algorithms that prohibit a manipulation

that would lead to the creation of universe faces. Thus, since there are no universe faces in a solid net component, there is no need to index face universe objects.

With the help of the indices, the boundary retrieving algorithms can be kept concise, clear, and perform in a linear runtime, depending only on boundary size. Listing 5 shows, as an example of boundary retrieving methods, the `getBoundaryEdges` method of `FaceNet3DCompNetLevel` class that efficiently returns all boundary edges of a face net component in an *ordered sequence*.

```
1.  public Collection<Edge> getBoundaryEdges() {
2.    Collection<Edge> result = new LinkedList<Edge>();
3.    for (Face uFace : this.getAllUniverseFaces()) {
4.      for (Edge edge : new EdgeIterator(uFace)) {
5.        result.add(edge);
6.      }
7.    }
8.    return result;
9.  }

10. public Collection<Face> getAllUniverseFaces() {
11.   return this.universeFaceIndex.values();
12. }
```

Listing 5: Implementation of getBoundaryEdges and getAllUniverseFaces methods of FaceNet3dCompNetLevel class

The algorithm of the `getBoundaryEdges` method starts with the retrieval of a collection of all universe faces of the face net component on which it is invoked. To retrieve all universe faces, the method internally invokes the `getAllUniverseFaces` method (line 3), which in turn accesses the `universeFaceIndex` instance property of `FaceNet3DCompNetLevel` class (this index is the *index of universe faces*, presented in Fig. 53). The method returns the values of this associative map, which are the universe faces themselves (line 11). The `getBoundaryEdges` method then iterates through all universe faces of the collection (line 3). For every universe face, the algorithm creates an *edge iterator* on the face (see instantiation of object of type `EdgeIterator` in line 4). An edge iterator is an example of a *cell iterator*. Cell iterators are introduced in Ch. 3.3.2 in detail, where the meaning of cell iterators and implementation detail is given. However, basically an edge iterator returns all edges of a face in an ordered sequence by iterating an *orbit*. It is not capable of evaluating holes by itself. The `getBoundaryEdges` method instantiates edge iterators of the face that represents the outer void, and edge iterators of all faces that represent inner holes, in order to collect all boundary edges. These edges are collected in a result collection (line 5), which is returned as the return value of the method (line 8).

Now it becomes obvious, why holes require the creation of separate instances of universe faces. The `getBoundaryEdges` method uses an edge iterator which by itself can only

collect the cells of one hole face or of the outer void universe but not of all universe faces combined. A cell iterator needs a cell-tuple to start with, which can be received from a universe face through the `getACellTupleOfCell` method (see Fig. 42). If there is only one universe face with multiple parts (the inner holes), then it would not be possible to gain e.g. all boundary edges of the universe face through an edge iterator, since not all cell-tuples of the multi-part universe face would be accessible through one orbit. Only if the multi-part universe face is modelled as multiple separate instances of universe face type, it is possible to run orbits on all parts of the universe and thus to collect all boundary cells.

Similar methods are implemented to retrieve boundary nodes (`getBoundaryNodes`) and boundary faces (`getBoundaryFaces`) (see Fig. 54).

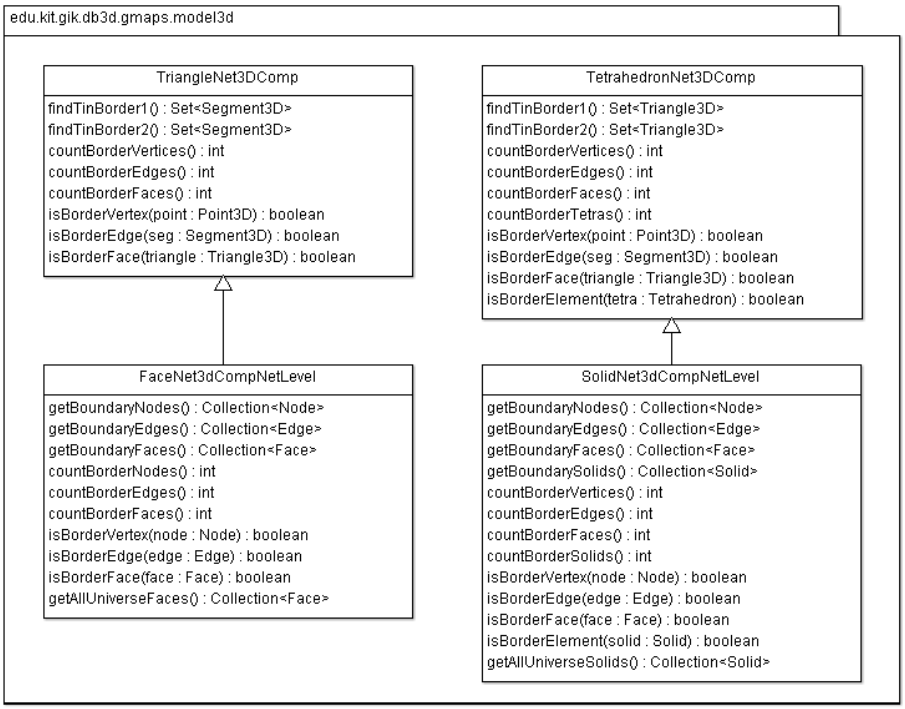


Fig. 54: Methods of cell net components that retrieve boundary cells

These methods are not only available for face net components, but also for solid net components. Additionally, solid net components also provide a `getBoundarySolids` method that retrieves all boundary solids of a solid net component. All these boundary retrieval methods operate on the same principle as presented in Listing 5.

`FaceNet3dCompNetLevel` and `SolidNet3dCompLevel` also have `countBorder<cells>` and `isBorder<cells>` methods. The `countBorder<cells>` methods are based on the `getBoundary<cells>` methods. The purpose of the `countBorder<cells>` methods is to count and to return the number of cells of the respective cell type (node, edge, face or solid) that can be found at the boundary of the cell net component. It returns the number as the method's return value of type `int`. Internally, a

`countBorder<cells>` method simply invokes the suitable `getBoundary<cells>` method and reads the size of the returned cell collection. The implementations of `countBorder<cells>` methods override the homonymous methods of its superclass `TriangleNet3DComp` (which is a class of the DB4GeO Kernel).[103]

The `isBorder<cells>` methods have the purpose to check whether a given cell is part of the boundary of the cell net component at hand. As an example of the implementation of the `isBorder<cells>` methods, Listing 6 shows the actual implementation of the `isBorderVertex(Node)` method of `FaceNet3dCompNetLevel` class.

```
1.  public boolean isBorderVertex(Node node) {
2.    if (!nodeIndex.containsKey(node.id))
3.      return false;
4.    OrbitIterator orbit1 =
           new OrbitIterator(node.anyCellTuple, 1);
5.    for (CellTuple ct : orbit1) {
6.      if (ct.isInFaceUniverse())
7.        return true;
8.    }
9.    return false;
10. }
```

Listing 6: Implementation of isBorder(Node) method of FaceNet3dCompNetLevel class

The algorithms of these methods first check whether the given cell is part of the cell net component that the method is invoked on. In the example of `isBorderVertex(Node)` method, the algorithm first checks whether the given `node` is in the `nodeIndex` of the component (line 2). If the given cell is not a cell of the component, then it cannot be part of its border and therefore `false` is returned. In a second step, the algorithm collects all cell-tuples of the given cell, which, in the case of `isBorderVertex(Node)` method, are all cell-tuples “around” the given node. To gather all cell-tuples around a node, a *1-orbit* is needed (instantiated in line 4). Orbits are given in the form of *iterators* by the API. *Orbit iterators* are described in Ch. 3.3.1 in detail. Basically, a 1-orbit iterator can be used in a `for`-loop to iterate step-by-step over all cell-tuples that reference the given node (line 5).

Each cell-tuple of the 1-orbit iterator is checked whether it also references a universe face (whether it is lying in a universe face) (line 6); if one is found, the method returns `true` (line 9). All other `isBorder<cells>` methods of `FaceNet3dCompNetLevel` class operate in a similar way, the differences only involve different orbit iterator types (line 4, depending on the given cell) and different universe cell types (line 6, depending on the type of the cell net component on which the method is invoked).

As the class diagram in Fig. 54 shows, this boundary finding functionality is already provided by the superclasses `TriangleNet3DComp` and `TetrahedronNet3DComp` of the

103 This is why the denomination of these methods break with the general rules (vertices instead of nodes etc.): these methods have to have the same name as the ones of Trianglenet3DComp that are overridden.

DB4GeO Kernel. For example, the `findTinBorder1` and `findTinBorder2` methods of the DB4GeO Kernel have the same purpose as the `getBoundary<cells>` methods of the Topology Module: they all retrieve the boundary elements of the net components.[104] Though, these methods serve the same purposes, the underlying models and algorithms are very much different. This is reflected in highly differing *asymptotic runtimes*. Since the Topology Module makes extensive use of references between the cells, it is assumed that these methods will have exceptional gains in asymptotic runtimes. Due to this fact, the structure, created by the Topology Module, can also be referred to as a "*topological index*". However, this assumption is checked with runtime tests, and the results are presented in Ch. 4.

3.1.8 Object Level and Net Level

To prepare the model for hierarchy management, a cell net component is conceptionally further subdivided into a cell net component at *network level* (or just *net level*, C_{NL}) and a cell net component at *object level* (C_{OL}) (cf. Fig. 55).

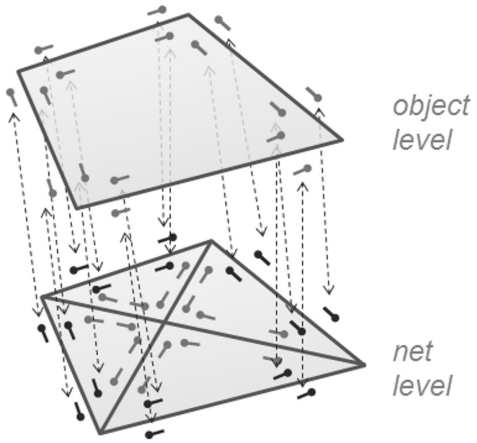


Fig. 55: Example configuration of cell net components at net level (bottom) and at object level (top)

The topology of C_{NL} is an exact reproduction of the topology of the net structure of the underlying net of simple geo-objects. The network level is mainly used for navigational purposes. It eases the algorithmic navigation on the net structure. Once C_{NL} is created through the constructor, the topology defined by the cell-tuple structure cannot be edited by the user on this level any more. This edit restrictions are backed by the employment of interfaces that define *editable* (of type `EditableCellNet3dCompLevel`) and *non-editable* (of type `CellNet3dCompLevel`) cell net levels (cf. Fig. 56).

104 The differences in functionality between the methods findTinBorder1 and findTinBorder2 are explained in Ch. 4.

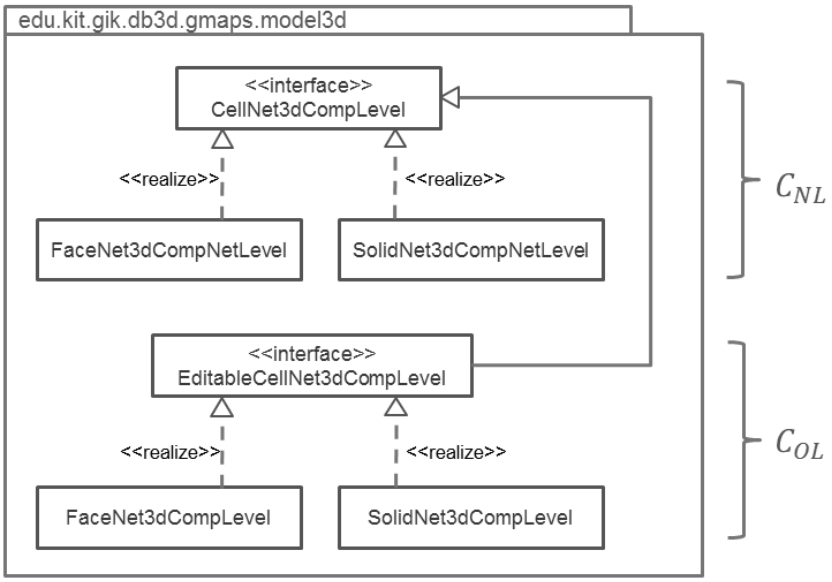


Fig. 56: Two distinctive cell net component interfaces

As soon as C_{NL} construction has finished, C_{OL} is deduced from C_{NL}. C_{OL} is the *boundary representation* of the component object (cf. top of Fig. 55).

The cells of C_{OL} are not reused cells of C_{NL} but completely newly instantiated cells. As an example, this means that one new inner face and one new face that represents the "outer void" are created *on object level* for a face net component *on net level* (without a hole). Furthermore, for every edge and node at the boundary of the component on net level, a new edge and node object is created on object level.

For every cell-tuple in C_{NL} that belongs to the "outer void" (d_U), a new cell-tuple in C_{OL} is instantiated. Additionally, for all cell-tuples $\alpha_2(d_U)$ of C_{NL}, the according new cell-tuples in C_{OL} are created. Every newly created cell-tuple of C_{OL} is linked to the respective cell-tuple at C_{NL}; this references are also modelled backwards from C_{NL} to C_{OL} (two-way link, reversible uniquely assigned). The references between the cell-tuples of the net level and of the object level are stored in `higher` and `lower` field properties of `CellTuple` class. To accommodate this functionality, the model of `CellTuple` class (Fig. 44) is extended by the two new attributes as depicted in Fig. 57.

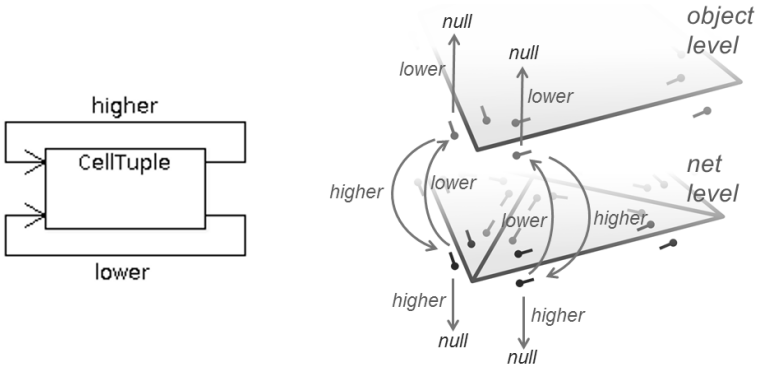


Fig. 57: Higher and lower field attributes of CellTuple class (left: class diagram; right: example set-up)

The `lower` references of all cell-tuples of C_{OL} are always set to `null` (because there can be no lower level than the object level itself). The `higher` references of all cell-tuples of C_{NL} are also always set to `null` (because there is no higher level possible than the net level). The `higher` references of all cell-tuples of C_{OL} are initially pointing to their respective cell-tuples at C_{NL}. A subset of the `lower` references of the cell-tuples of C_{NL} link to the respective cell-tuples at C_{OL}. Not all `lower` references of the cell-tuples of C_{NL} can be set, since not every cell-tuple at C_{NL} has a representative at C_{OL}. For example, directly after the construction, all the cell-tuples of C_{NL} that are not at the boundary of the component have no representative at C_{OL}. Thus, the `lower` references of such cell-tuples are also set to `null`.

The key mindset of this concept is that the net level always reflects the net structure of the underlying geometry, which is unchangeable, once build. The object level instead reflects the geo-object (as boundary representation), which can be topologically edited/changed by the user. To reflect this concept, the level indicating classes on net level and on object level differ in the interfaces they implement (`CellNet3dCompLevel` or `EditableCellNet3dCompLevel` interface) (cf. Fig. 56 and Fig. 58).

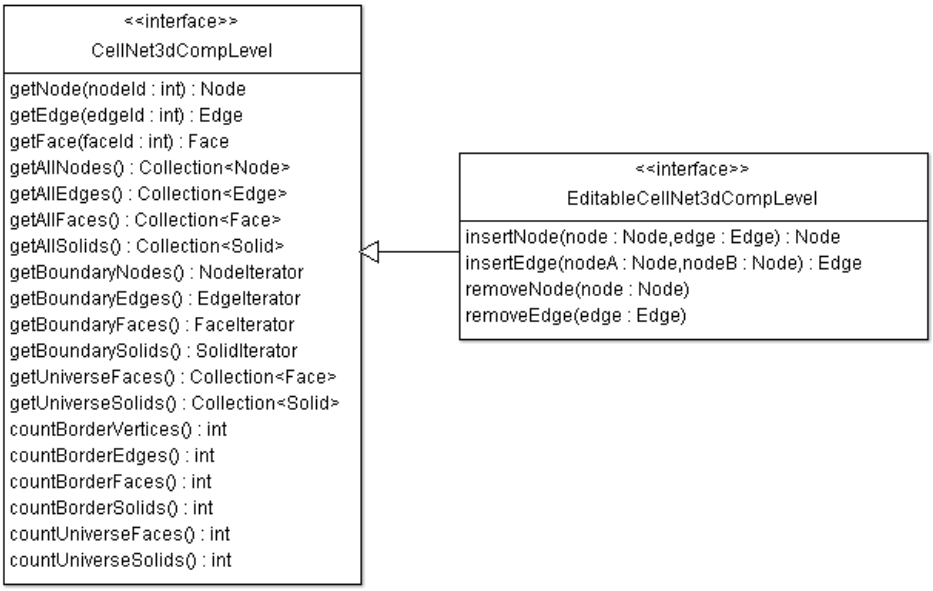


Fig. 58: Cell net comp level methods

All cell net component classes on net level (i.e. `FaceNet3dCompNetLevel` and `SolidNet3dCompNetLevel`) implement the `CellNet3dCompLevel` interface which only requires to implement methods that query the cell net structure – but it does not require/allow for methods that modify the cell net structure on that level. In contrast, all cell net component classes on object level (i.e. `FaceNet3dCompLevel` and `SolidNet3dCompLevel`) implement the `EditableCellNet3dCompLevel` interface, which extends the `CellNet3dCompLevel` interface – i.e. it requires the implementation of all the methods of `CellNet3dCompLevel` interface that require the querying of the structure, plus methods of `EditableCellNet3dCompLevel` that allow for a modification of the cell net structure on that level.

To get an overview of the allocation of the methods, it is appropriate to start with the *getter* methods. A cell net component, whether at net level or at object level, can be searched for a cell object of a certain ID through its `get<cell>` method by passing a cell ID of type `int` as a parameter. The method returns the cell with the given ID only if it actually is part of the cell net at this level; otherwise the method returns `null`. Furthermore, any cell net component can return collections that represent all cells of a certain type of the respective component by the `getAll<cells>` methods. The `getUniverseFaces` and `getUniverseSolids` methods return sets of all faces and solids that have been marked as universe cells. The `getBoundary<cells>` and `countBorder<cells>` methods have been discussed in detail in Ch. 3.1.7.

The `insertNode` method that is required by the `EditableCellNet3dCompLevel` interface shall place a node onto an edge at object level. The object of type `Node` that is required as the first parameter, must be a `Node` at *net level*. The `insertNode` method creates a new node at *object level* that “mirrors” the given node of net level at object level

(i.e. these represent the same node at net level and at object level – though both nodes are individual objects and have differing object IDs). The `insertNode` method splits the edge that is passed to the method as the method's second parameter, into two edges. Finally, the method returns the newly created node at object level as the methods return value.

The `removeNode` method that is also a method of the object level (i.e. required by the `EditableCellNet3dCompLevel` interface) deletes a node (that is located on an edge). The method only requires a `Node` object as its parameter. This has to be the node that shall be removed from the cell net at object level – i.e. it must be a node of object level. The operation removes the node and thus merges the two edges, that are incident to the node, into one edge.

The `insertEdge` method inserts an edge between two nodes. This method needs two nodes as its parameters. These nodes define where the new edge shall be inserted in-between. The method instantiates a new edge and defines the two given nodes as incident to the new edge. Also, the insertion of a new edge splits a face into two. Thus, the method also creates two new faces and defines them as incident to the newly created edge.

The `removeEdge` method deletes an edge that is located on faces. The method only requires an `Edge` object as its parameter. This has to be the edge that shall be removed from the cell net at object level – i.e. it must be an edge of object level. The operation removes the edge and thus merges the two faces that are incident to the edge, into one face.

Before the actual algorithms of the `insert<cell>` and `remove<cell>` methods are described in detail[105], it is helpful to have an elaborated inside look into the algorithms that are needed to create the cell nets out of nets of simple geo-objects and into the algorithms that simplify the traversal of the darts of a G-Map.

3.2 Constructing Cell-Tuple Structure from DB4GeO Simplicial Complexes

The Topology Module provides the means to construct cell-tuple structure (G-Map) from *triangle nets* and *tetrahedron nets* that are provided by *db3dcore*. First, for each component of a simplicial net, a distinct G-Map is created. Afterwards, all components of the cell net (`FaceNet3dComp`/`SolidNet3dComp`) are attached to the respective cell net (`FaceNet3d`/`SolidNet3d`). The cell net is finally returned by the builder object (see Ch. 3.1.6).

The construction process for a cell net component is subdivided into two main steps,

1. the construction of cells and cell-tuples at *net level* (`FaceNet3dCompNetLevel`/`SolidNet3dCompNetLevel`) and
2. the construction of cells and cell-tuples at *object level* (`FaceNet3dCompLevel`/`SolidNet3dCompLevel`).

105 Algorithms are discussed in Ch. 3.4

In fact, not a component itself but each level of a component constitutes a distinct G-Map. After the construction of G-Maps of net level and object level completes, both levels are attached to the respective cell net component.

The following section explains the algorithms for the construction of cells and cell-tuples at net and at object level. The explanation is given for triangle net components first – the differences to the algorithm for the construction out of tetrahedron net components are described thereafter (Ch. 3.2.5).

3.2.1 Framework for Cell Complex Construction

Basically, the algorithm traverses every single triangle of the triangle net once, processing the following two steps:

1. create all cells and cell-tuples for every triangle solemnly and
2. connect the cell-tuples of a triangle (face) with the cell-tuples of its adjacent triangles.

The actual algorithm implements a DIJKSTRA-based[106] approach for the traversal of all triangles of a triangle net. The algorithm utilizes an *ordered list* L_P of *face-triangle-pairs*, where $P=(f,te)$ is a pair of one *face* f (geometrically embedded by a triangle t_f) and a *triangle element* te. The algorithm has to ensure that if f and te are in P then t_f and te are geometrically adjacent to each other.

Additionally, an *indexed set* of already visited triangles S_f is prepared. Every te that is put on S_f has to be "transformed" into a face f in advance. So in fact, S_f does not contain the triangles but the faces they have been transformed into. Every f is indexed by the ID of the affiliated te, so that any f on S_f can be retrieved through the operation $f=S_f(te)$.

Listing 7 describes the global frame of the algorithm of the `cellNetBuildUp` method (of `FaceNet3dCompNetLevel` class), i.e. how a *triangle net component* (class `TriangleNet3DComp`) is traversed and the *cells* and *cell-tuples* are created:

```
Method: cellNetBuildUp
Purpose: Construct cell complex on net level from triangle net
component
Parameter: A (any) triangle element te_s of the triangle net component
for which the cell complex shall be constructed (the "start" triangle)
1. Push te_s onto L_P (this first te_s has no f companion in P
2. While L_P is not empty do
3.   Poll (take and remove) the first P of L_P
4.   Extract te from P
```

106 The Dijkstra shortest path finding algorithm is explained later when it is shown how the path finding algorithm can take advantage of the G-Maps structure of the Topology Module.

```
5.   If te has not already been visited (is not already on S_f)
6.     Create all cells (i.e. nodes, edges and the face f_current)
       and all cell-tuples for te
7.   Else
8.     f_current = S_f(te)
9.   Extract f_before from P (if possible)[107]
11.  If f_before exists (it does not exist for first P)
12.    Merge f_current with f_before[108]
13.  For all neighbour triangle elements te_i,{i∈ℕ|1≤i≤3} of te
14.    If te_i exists and is not already on S_f
15.      Create new P_N=(f_current,te_i) and push it onto the end
         of L_P
16.  Add te to S_f
```

Listing 7: Pseudocode description of the algorithm of cellBuildupOnNetLevel method

The algorithm of `cellNetBuildUp` method of Listing 7 is also depicted in a flow chart diagram in Fig. 59 for a better overview.

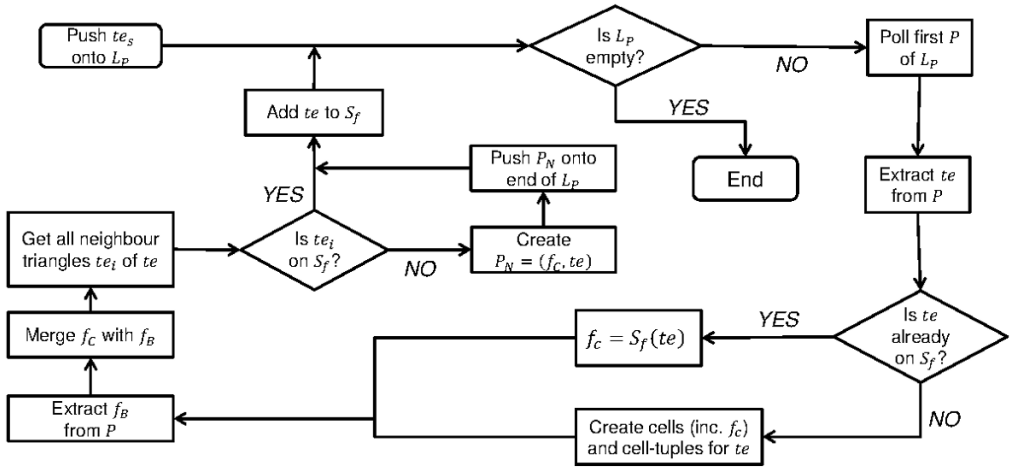


Fig. 59: Flow chart diagram of cellNetBuildUp method

In summary, the algorithm starts with the first triangle element of the triangle net component and "transforms" it into a face (i.e. all its cell-tuples are created). The face is pushed pairwise with all of its neighbouring triangles onto the list of face-triangle-pairs. Thus, the triangles of a face-triangle-pair are known to be adjacent. Then, new triangle-face-pair elements are taken from the list of face-triangle-pairs in order to be processed. They repeatedly undergo the same procedure (which is: create cell-tuples, build pair) but

107 This is not possible for the triangle element that is processed as the first one in the algorithm since there has been no triangle before that was already "transformed" into a face (i.e. that all its cell-tuples were created)

108 Merge operation is described in next sections

they also merge with their (previously generated) adjacent faces. It is guaranteed that every triangle is reached (processed) since the algorithm accesses the face-triangle-pairs in a FIFO[109] manner.

3.2.2 Creating Cells and Cell-Tuples of a Triangle

This section describes how the cells and the cell-tuples that describe the topology of a triangle element[110], are created (referred to in sixth step of Listing 7) and how two faces are merged (referred to in twelfth step of Listing 7). The creation of cell and cell-tuple objects for a single triangle is encapsulated in the `private createCellsOfTriangle` method of the `FaceNet3dCompNetLevel` class (see Listing 8).

```
Method: createCellsOfTriangle
Purpose: Create all corresponding cell-objects (i.e. nodes, edges and a
face) for the given triangle element (TriangleElt3D) and also create
all corresponding cell-tuples
Parameter: The triangle element (TriangleElt3D) for which to create all
corresponding cell objects (i.e. nodes, edges and a face)
1. Create three Node objects for the three points of the
   triangle (link the nodes to the corresponding point objects)
2. Create three Edge objects for the three line segments of the
   triangle (link the edges to the corresponding segment
   objects)
3. Create one Face object for the triangle (link face object to
   the corresponding triangle object)
4. Register all newly created cell objects in the appropriate
   index fields of the face net component instance. The cells
   are then sorted by their ID
5. Create only the six "inside-lying"[111] cell-tuple objects of
   the triangle and link each cell-tuple to its four incident
   cell objects (0-, 1-, 2- and 3-cell). Create a back-reference
   from every cell object to the respective cell-tuple
6. Put all newly created cell-tuple objects into an index field
   (face net component instance) which sorts the cell-tuples by
   their ID
```

Listing 8: A textual description of createCellsOfTriangle algorithm

The creation of cell-tuples (as mentioned in the fifth step of Listing 8) is implemented straightforward: the constructor method of a cell-tuple is given all the cell objects that constitute the path in the incidence graph that is represented by that cell-tuple (see step 1 in Listing 9 for example set-up). A "universe" solid is given as parameter to the cell-tuple constructor; a *universe* solid is a 3-cell with the ID -1 and indicates the outer space. Since

109 First in, first out

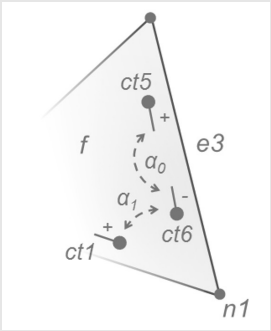
110 Implemented in *db3dcore*

111 In order to avoid the creation of unnecessary cell-tuple objects in the process, the "outside-lying" cell-tuples (that reference the universe faces) are created in the end, after the creation of all "normal" faces has finished.

the `createCellsOfTriangle` method of the *FaceNet3DComp* only regards triangle nets (i.e. always a *2-dimensional* geometry), there is always the *universe* solid on the "top" and the "bottom" of the face net component. Thus, all cell-tuples that are created in the `createCellsOfTriangle` method have *universe* as its 3-cell.

Then all the involutions ($\alpha_0, \alpha_1, \alpha_2, \alpha_3$) towards the already existing cell-tuple objects are established by assigning the appropriate values to `alpha<dim>` class attributes of the cell-tuples (see code example in Listing 9).

```
...
1. CellTuple ct6 =
     new CellTuple(node1, edge3,
     face, universeSolid, false);
2. ct5.alpha0 = ct6;
3. ct6.alpha0 = ct5;
4. ct1.alpha1 = ct6;
5. ct6.alpha1 = ct1;
6. ct6.alpha2 = ct6;
7. ct6.alpha3 = ct6;
...
```



Listing 9: Java code excerpt for the creation of a cell-tuple (ct6) in createCellsOfTriangle method

The knowledge of the cell-tuple structure, i.e. which cell-tuples are adjacent, is extracted from the net topology structure of the underlying triangle net component of the DB4GeO Kernel[112]. Instances of `TriangleElt3D` class of the Kernel reference adjacent and incident geometry objects by index numbers (see Fig. 60).

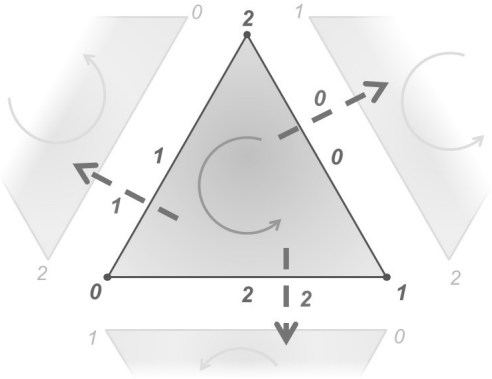


Fig. 60: Indexing of net topology in DB4GeO triangle net component

The neighbourhood relations between points, segments, and triangles are codified within the index numbers *0*, *1*, *2*. For example, an incident point and a neighbouring triangle with

112 Which is built up and managed by the `TriangleNet3DComp` class of *db3dcore*

the same index number oppose each other. A segment and two points with unequal index numbers are incident.

Since it is known inside the `createCellsOfTriangle` method, which point is transformed into which node, which segment is transformed into which edge, and which triangle is transformed into which face, all relations between edges and nodes inside a face can also be deduced.

3.2.3 Merging Cells and Cell-Tuples of Faces

After all edge, node, and cell-tuple objects of a face have been created, the face is "glued" to its neighbouring faces (see Fig. 61), with the purpose to build one contiguous face net component at net level.

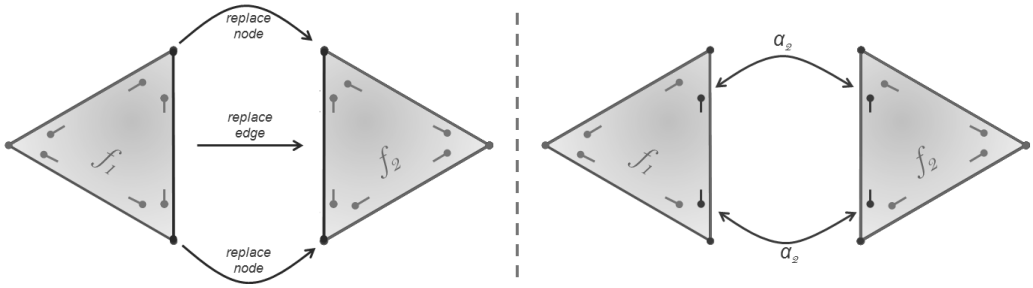


Fig. 61: Inspection of cell identity (left) and remapping of alpha-2 involutions during face merging process (in mergeFaces method)

The algorithm of Listing 7 iterates over all triangles of the triangle net component (line 2) in order to merge all faces (cf. line 12).

In the process of merging two neighbouring faces, identical nodes and edges are identified and unified, as well as the α_2-involutions between cell-tuples in the opposing faces are set. This is done by the algorithm of the `mergeFaces` method of the `FaceNet3dCompNetLevel` class (see Listing 10).

```
Method: mergeFaces
Purpose: Merge the cells and connect the cell-tuples of the two given
faces at the intersection areas of the corresponding triangles
Parameter: The two 2-cells that shall be merged are given as
parameters: face f1 and face f2 of class Face. The cells incident to
f1 will remain after the merging process whilst the cells incident to
f2 will be erased and replaced
1. For all edges (e_f1) of f1
2.   For all edges (e_f2) of f2
3.    If e_f1 equals(is geometrically equivalent to) e_f2
4.      Get all cell-tuples (d) that reference e_f1
5.      Get all d that reference e_f2
6.      Get the two nodes of e_f1
```

```
7.      Get the two nodes of e_f2
8.      Check which of the nodes of e_f1 are equal to which of
        the nodes of e_f2
9.      Based on the information of step 8: correctly update the
        α2-involution links of all d of e_f1 and e_f2
10.     The nodes and the edge of f1 replace the nodes and
        the edge of f2. Therefore: update/replace the 0-cell
        and 1-cell references of all d of e_f2. Remove e_f2 and
        the nodes of the edge from the face net component's
        nodes and edges index
```

Listing 10: Pseudocode/textual description of the algorithm of mergeFaces algorithm

In summary, the `mergeFaces` method iterates over all edges of the both faces that shall be merged (lines 1 and 2 of Listing 10). All edges are compared crosswise on geometric equality of their line segment representatives (line 3).[113] When the two equal edges are identified, then the point representatives of their nodes are checked for geometric equality in order to identify the correct alignment of the edges (Fig. 61, left).[114] After the alignment is known, the correct α_2-involution links between all cell-tuples of the both edges are set (Fig. 61, right). All duplicates of identical cells (nodes and edges) that are now merged, are removed from the structure (especially from the indices).

3.2.4 Creating Universe Faces and Object Level Structure

After merging the cells and cell-tuples of all faces of a face net component, the component is still incomplete. In the following step, the “outside structures” are created. These are all faces and cell-tuples that are located outside the boundary (in the 2D universe) of the face net component (see Listing 11).

```
Method: createUniverseFaces
Purpose: Create a universe face for the outer void and for all inner
holes and all cell-tuples that lie “outside” the face net component
Parameter: Any cell-tuple of the face net component for which to build
the “universe structure”
1. Create a set of all cell-tuples that are located at
   component's 2-cell boundary ( S_ct(B) )
2. While S_ct(B) is not empty
3.   Take a cell-tuple ( ct_B ) from S_ct(B) (memorize this
     cell-tuple also as start cell-tuple ct_S )
```

113 This yields *3x3=9* checks on geometric equality of line segments per triangle at a maximum.

114 The gathering of all the information that is needed for merging of neighbouring faces can also be achieved with improved runtime performance by an expanded use of the underlying triangle orientation structure that is presented in Fig. 61. However, this would also complicate the algorithm and make its maintenance more difficult. Thus, the presented simple approach has been favoured.

```
4.   Create a new universe face ( f_U )
5.   Put  f_U  on universe face index ( I_{f_U} )
6.   Do
7.     Create new cell-tuple ( ct_U ) as copy of  ct_B  but with
       f_U  as 2-cell (and set all involution links as far as
       possible)
8.     Set  ct_U  as  α_2  of  ct_B  (and vice versa)
9.     Set  ct_U  as  α_1  of  ct_before  (and vice versa)
10.    Remove  ct_B  from  S_{ct(B)}
11.    ct_B ← α_0(ct_B)
12.    Create new cell-tuple ( ct_{U(α_0)} ) as copy of  ct_B  with  f_U
       as 2-cell
13.    Set  α_0(α_2(ct_B))  as  α_0  of  ct_U
14.    Set  ct_U  as  α_0  of  α_0(ct_U)
15.    Set  ct_U  as  α_2  of  ct_B
17.    ct_before ← ct_U
17.    Remove  ct_B  from  S_{ct(B)}
18.    Move cell-tuple iterator "coast to coast" until it
       reaches the boundary again and set the cell-tuple as  ct_B
19.  While  ct_B ≠ ct_S
```

Listing 11: Pseudocode/textual description of the algorithm of createUniverseFaces method

In this process, for each inside cell-tuple of the component's boundary, a new "twin" cell-tuple is created as a copy of the inside cell-tuple with the difference that the new tuple gets a universe face as its 2-cell entry (see line 7 and 12 in Listing 11). When generating outer cell-tuples, it is of particular importance to consider possible holes in the component. As discussed in Ch. 3.1.7, each hole renders a new universe face instance (with a unique face ID).

The algorithm first creates a set of all cell-tuples of the component's boundary (line 1). This is achieved by iterating through all cell-tuples of the component and checking each tuple. If $\alpha_2(d)=id$ is `true` for a cell-tuple then it lies at the component's boundary and therefore is added onto the result set.

After the set is compiled, it is iterated in the next step. For each cell-tuple, a new universe cell-tuple is created. A simple 2-orbit along the "outer" component's boundary – i.e. along the cell-tuples that lie in the universe, is not possible so far since the outer cell-tuple structure is not generated at this state and thus would lead to erroneous behaviour. In order to find the inner way along the component's boundary, the iterator often needs to move "coast to coast" (see line 18). For this purpose, a private `moveToBoundary` helper method is developed that takes an inner boundary cell-tuple as parameter value, then moves "fast forward" a *0-orbit* until it reaches the boundary again (see Fig. 62) and finally returns the other (opposite) inner boundary cell-tuple.

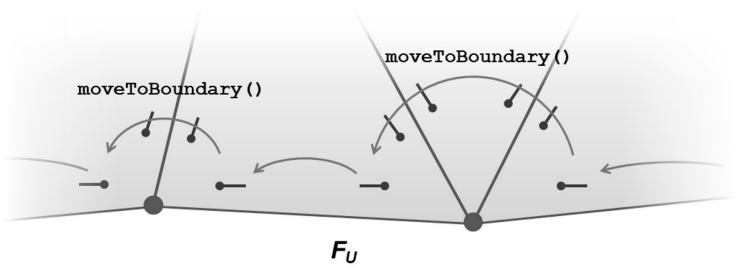


Fig. 62: Navigating along "inner side" of component boundary

The condition for reaching the boundary in *0-orbit* is to identify an inner boundary cell-tuple (d_b) for which $\alpha_2(d_b)=id$ [115] is still valid at this state.

Each time, a new boundary cell-tuple is processed, it is removed from the set of boundary cell-tuples. This way, only cell-tuples remain in the set that are part of other, dissimilar boundaries of the component. These cell-tuples are processed in the following steps of the algorithm, until all cell-tuples are removed from the set.

In a final step, the object level of a cell net component is deduced from the now completed net level in the `cellNetBuildUp` method of `FaceNet3dCompLevel`. This is simply done by performing *2-orbits* around all *universe faces* and creating object level siblings of all encountered net level cells and cell-tuples. In the course of the process, each newly created cell-tuple of object level is (bi-directionally) linked to its sibling cell-tuple of net level (cf. Ch. 3.1.8).

3.2.5 Constructing Solid Complexes From Tetrahedral Nets

So far, the process of constructing cell complexes and G-Maps from DB4GeO Simplicial Complexes has been discussed on the basis of 2-dimensional structures (triangle nets). The construction process for 3-dimensional cell complexes (on basis of tetrahedral nets in DB4GeO) is analogue but encompasses some minor differences.

Analogously to the `FaceNet3dCompNetLevel` class that represents a face net component at net level, the `SolidNet3dCompNetLevel` class represents a solid net component at net level. The cell net build up entry method `cellNetBuildUp` of `SolidNet3dCompNetLevel` class is mostly the same as the `cellNetBuildUp` method of `FaceNet3dCompNetLevel` class (cf. Ch. 3.2.1). The major difference for solid build up is that the starting point of the geometric processing is not a triangle but a tetrahedron that first has to be queried for its four incident triangles – the rest of the algorithm, concerning the lists of visited elements (now tetrahedra instead of triangles) and the list of element pairs, is similar.

---

115 This condition becomes incorrect after the universe cell-tuple are created. The condition for inner boundary cell-tuples is then $\alpha_2(d_b)=d_U$.

Accordingly to the above described `createCellsOfTriangle` method of the face net component constructor (see Ch. 3.2.2), the solid net component constructor has a `createCellsOfTetrahedron` method which creates all cells (nodes, edges, faces, and a solid) and all *48 cell-tuples* that describe one single tetrahedron. The `mergeSolids` (Listing 12) method of the `SolidNet3dCompNetLevel` class has a similar purpose as the `mergeFaces` method of `SolidNet3dCompNetLevel` class (see Ch. 3.2.3): it merges the two given solids along their equal faces.

Method: mergeSolids
Purpose: Merge the cells and connect the cell-tuples of the two given solids at the intersection areas of the corresponding tetrahedra
Parameter: The two 3-cells that shall be merged are given as parameters: solid s_1 and solid s_2 of class *Solid*. The cells incident to s_1 will remain after the merging process whilst the cells incident to s_2 will be erased and replaced

1. Find the face f_1 of s_1 and the face f_2 of s_2 whose corresponding triangles are geometrically equivalent
2. Find the both cell-tuple d_1 and d_2 of f_1 and f_2 that match – where "match" means that the point (of the node) and the segment (of the edge) of d_1 and d_2 are geometrically equivalent. In other words, d_1 and d_2 in the two solids s_1 and s_2 are located at the same point, pointing along the same segment
3. Use d_1 and d_2 as start cell-tuples to instantiate $\langle \overline{\alpha_2} \rangle (d_{1(S)})$ and $\langle \overline{\alpha_2} \rangle (d_{2(S)})$
4. **For** each involution step in $\langle \overline{\alpha_2} \rangle (d_{1(S)})$
5. Do an involution step in $\langle \overline{\alpha_2} \rangle (d_{2(S)})$
6. Link the two current cell-tuples of both sides ($d_{1(C)}$ and $d_{2(C)}$) so that $\alpha_3(d_{1(C)}) = d_{2(C)}$ and $\alpha_3(d_{2(C)}) = d_{1(C)}$
7. Replace/update the 2-cell reference of $d_{2(C)}$
8. Update 0-cell references of all cell-tuples of $\langle \overline{\alpha_0} \rangle (d_{2(C)})$
9. Update 1-cell references of all cell-tuples of $\langle \overline{\alpha_1} \rangle (d_{2(C)})$
10. Remove f_2 and the edges of f_2 and the nodes of the edges from the solid net component's nodes, edges and faces index

Listing 12: Textual description of the algorithm of mergeSolids method

The algorithm first searches for the two matching faces/triangles of the solid and then the two matching cell-tuples inside the faces. The following steps (from line 3) harness the functionality of the `OrbitIterator` framework (which is explained in detail in Ch. 3.3.1). Basically, the algorithm makes use of 2-orbits in both solids, starting with the two matching cell-tuples $d_{1(S)}$ and $d_{2(S)}$ (see Fig. 63).

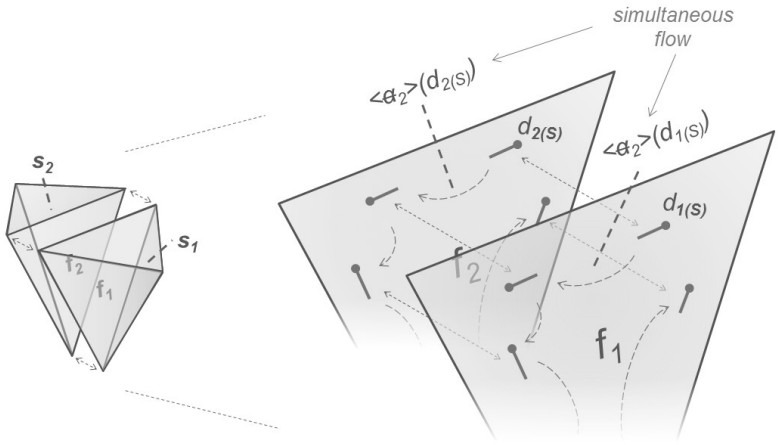


Fig. 63: Merging cells and connecting cell-tuples of two tetrahedra (left); confronting faces detail (only confronting faces are depicted): simultaneous 2-orbits on both sides of faces (right)

The both 2-orbits simultaneously lead along the equal faces of the two solids, cell-tuple by cell-tuple. On each step, the cell-tuples on both sides of the faces are linked by α_3 (line 6), and cell references of all affected cell-tuples are updated properly (lines 7 ff.).

Finally, all universe solids and all cell-tuples of universe solids are instantiated and linked. Then, the object level is deduced from net level. These processes are very similar to the processes for 2-dimensional structures described in Ch. 3.2.4. The main difference is that for finding cell-tuples of the inner boundary of the solid component, the condition for identifying such cell-tuples is $\alpha_3(d) = id$.[116]

3.3 Basic Methods of the Topology Module

The previous chapters presented the basic class model of the G-Maps topology module for DB4GeO. This was followed by an explanation of how the class model is internally used to construct valid G-Maps. Now that valid G-Maps can be instantiated by the means of the module, the next step is to illustrate the basic algorithms that provide the functionality to query the topology of cells or whole geo-objects. The algorithms are implemented on a generalized sub-level (level of iterators) from where they can be reused in concrete query methods such as in the `getNeigbour`… methods of `Cell` objects.

3.3.1 Iterating an Orbit

The notion and usefulness of orbits on cell-tuples has been discussed in Ch. 2.3.3 and an example algorithm of orbit traversal has been presented. However, in a naive approach,

116 In this case, simply all valid cell-tuples of the cell net component at net level (stored in cell-tuple index of SolidNet3dCompNetLevel class) are checked for the condition.

such orbit algorithms could be implemented inside of ordinary methods that take a start cell-tuple and an orbit dimension indicator as parameters and return all cell-tuples of the complete *i*-orbit sequence in an *array* of cell-tuples. The disadvantage of such an approach would be that the gathering of cell-tuples in arrays could end up in very big arrays that have a high memory space consumption. A better approach to this issue is to create a framework that supports the retrieval of solitary cell-tuples with each step by calculating them "on-the-fly". This can be achieved through the development of *iterators* that iterate the cell-tuples of an orbit step by step.

The orbit framework in the Topology Module currently consists of the four classes `OrbitIterator`, `NodeIterator`, `EdgeIterator`, `FaceIterator` and `SolidIterator`[117], where `OrbitIterator` is the main, polar class that is needed by the other classes.[118] An `OrbitIterator` is an appropriate approach to model orbits on a graph based implementation of the cell-tuple structure.

An Orbit to Be Iterated and to Provide Its Iterator

To realize a clean architectural approach that leads to concise source code at the sections where orbits have to be employed, the `OrbitIterator` class utilizes the *iterator framework* of the Java API and thus provides a `hasNext` and a `next` method (see `Iterator` class of `java.util` package in Fig. 64).[119]

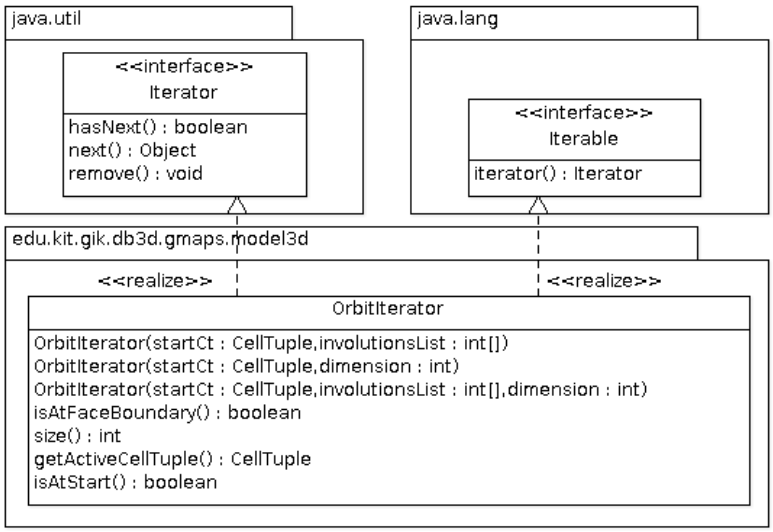


Fig. 64: Diagram of OrbitIterator class

117 All four classes are gathered in the edu.kit.gik.db3d.gmaps.model3d package. NodeIterator, EdgeItrator, FaceIterator and SolidIterator are discussed in next section.

118 Since the `OrbitIterator` class is only package visible, an `OrbitIterator` object can only be instantiated from inside its package (cf. the description of the principle of encapsulating the cell-tuple structure in the sense of information hiding in Ch. 3.1.2).

119 Regardless of the fact that the OrbitIterator implements the Iterator interface, substantially it is a circulator, as the start and end point of an orbit always coincide by definition.

There are at leased three straight-forward methods to create an orbit. The most basic method is to provide a start cell-tuple and an *array of integer values* that define the *involution sequence of the orbit*. This is realized in a constructor of `OrbitIterator` class. An example of a constructor invocation can be: `new OrbitIterator(startCt, new int[]{1, 3}` for the instantiation of an $\langle \alpha_{1,} \alpha_3 \rangle (startCt)$ -orbit (see line 1 of Listing 13 for complete example).

Another method is to provide a start cell-tuple and an integer value $n, \{n \in \mathbb{N} | 0 \leq n \leq 3\}$ that defines the dimension of the orbit. This method of orbit creation can always be translated into the basic method. The method determines an involution sequence according to the given orbit dimension. For example, if the given *n* value is *1*, then the orbit is $\langle \alpha_{0,} \alpha_2 \rangle (d_s)$ which translates into the basic method with `int[]{0, 2}`.

The third method to create an orbit is similar to the first but additionally requires another array of cell-tuples in order to be performed. This additional array of cell-tuples is used by the orbit as an artificial border where the orbit cannot go beyond. Application cases for this method are introduced in a following section.

After the creation of an orbit, the dimension of the orbit cannot be queried subsequently, since not every involution sequence (that can be provided to the creation method) yields a defined orbit dimension.[120]

Since an orbit is *iterable* by definition, it provides several methods that facilitate, and some that simplify the traversal of a cell-tuple orbit. An iterable orbit has to provide a method to query whether there is a next cell-tuple in orbit (`hasNext`) and a method that provides the next cell-tuple in orbit (`next`).[121]

Due to the architectural approach (an orbit is an iterator), the instantiation and usage of an `OrbitIterator` is quite simple and straight-forward, as presented in Listing 13 by the example of two different orbits:

```
1. OrbitIterator orbit13 = new OrbitIterator(startCt, new int[]{1, 3});
2. for(CellTuple ct : orbit13){
3.   System.out.println(ct);
4. }
5. OrbitIterator orbit0 = new OrbitIterator(startCt, 0);
6. for(CellTuple ct : orbit0){
7.   System.out.println(ct);
8. }
```

Listing 13: Java code example, demonstrating the usage of OrbitIterator in enhanced for-loops

120 e.g. $\langle \alpha_{1,} \alpha_3 \rangle (d_s)$ is not traversing the tuples of any cell of a certain dimension

121 These methods are required by the Iterator interface. The `Iterable` interface requires the `interator` method that returns an object of the `Orbit` class itself.

In Listing 13, it is shown how orbits can elegantly be used in *enhanced* `for`-*loops*[122]. Line 1 to line 4 of Listing 13 demonstrate an orbit *invocation by involution list* and print all cell-tuples of the orbit $\langle \alpha_1, \alpha_3 \rangle (d_s)$ to standard out. Line 5 to line 8 demonstrate an *invocation by orbit dimension* and print all cell-tuples of a *0*-orbit ($\langle \alpha_0 \rangle (d_s)$), i.e. an orbit around a node, starting and ending at the start cell-tuple `startCt`, to standard out.

Designing an Iterator On an Orbit

An iterator over an orbit consists of at least an *involution list*, a *start cell-tuple*, a set of *marked* cell-tuples and a *stack of collected darts*. The stack of collected darts is implemented in `OrbitIterator` as a field of the class (member variable `ctStack` is a `CellTuple` typed Java `Stack`, cf. first member variable in Fig. 65).

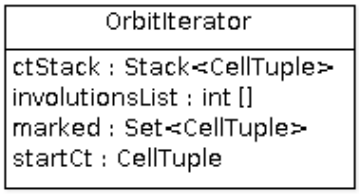


Fig. 65: Member variables of OrbitIterator class

In order to provide its functionality, an iterator over an orbit memorizes the *involution sequence of the orbit*. Thus, the class has an `involutionsList` class field, modelled as an integer array (`int[]`). Additionally, an iterator memorizes the start cell-tuple (class member `startCt` of type `CellTuple`) in order to be able to reach the termination condition. Finally, the iterator has a set of *marked* cell-tuples (member `marked` of type `Set`) in order to mark all cell-tuples that have already been visited by the orbit.

The `marked` set contains all cell-tuples that have been "touched" by the orbit on its way. This detail differs from the algorithm of Ch. 2.3. In Ch. 2.3, `marked` has been modelled as a variable of the dart class itself. Such an approach induces the disadvantage that only one orbit can be performed upon a cell net component at the same time. The approach of maintaining the markings of the cell-tuples in sets, having every `OrbitIterator` object posses its own set, causes the iteration of orbits to be *suitable for multi-threading*. This is of particular importance especially for the implementation of orbits in a database, since here, multiple users could wish to iterate over the same data at the same time with different orbits.

When an orbit shall be created by using a given orbit dimension, then the dimension integer must be translated into the matching involution sequence. This is realised by an

122 An object of the `OrbitIterator` can be used in an enhanced for-loop, since it is iterable (realising the `Iterable` interface).

`OrbitIterator` constructor that translates the `dimension` integer parameter value into the appropriate `involutionsList` array, the processing instructions in Listing 14:

```
Constructor: OrbitIterator
Parameter: The start cell-tuple for the orbit (startCt) and the
dimension (integer value) of the orbit to be computed (dimension).
1.  switch (dimension) {
2.  case 0:
3.    this.involutionsList = new int[] { 1, 2 };
4.    break;
5.  case 1:
6.    this.involutionsList = new int[] { 0, 2 };
7.    break;
8.  case 2:
9.    this.involutionsList = new int[] { 0, 1 };
10.   break;
11. case 3:
12.   this.involutionsList = new int[] { 0, 1, 2 };
13.   break;
14. default:
15.   throw new IllegalArgumentException("Unsupported
      dimension!");
16. }
```

Listing 14: Translating dimension integer parameter value into an involutionsList in OrbitIterator(startCt:CellTuple, dimension:int) constructor (Java code)

This algorithm simply conducts the translations: *0*-orbit = $\langle \alpha_1, \alpha_2 \rangle (d_s)$, *1*-orbit = $\langle \alpha_0, \alpha_2 \rangle (d_s)$, *2*-orbit = $\langle \alpha_0, \alpha_1 \rangle (d_s)$ and *3*-orbit = $\langle \alpha_0, \alpha_1, \alpha_2 \rangle (d_s)$. Other dimensions are not allowed and provoke an `IllegalArgumentException`.

Realisation of hasNext and next method of OrbitIterator

The realisation of the `hasNext` and `next` methods of `OrbitIterator` are straight-forward and analogous to the orbit traversal algorithm in Ch. 2.3. The `hasNext` method returns `true` if there is still *at least one* cell-tuple in the stack, i.e. the cell-tuple stack (`ctStack`) is *not empty* (cf. Listing 15):

```
return !ctStack.empty();
```

Listing 15: Idempotent implementation of OrbitIterator.hasNext method (Java code)

The `hasNext` method implementation of `OrbitIterator` is *idempotent*[123].

123 The invocation of the `hasNext` method does not manipulate/change the state (the structure of the class members) of `OrbitIterator`. This behaviour satisfies the requirement for `hasNext` method that is stated by the Java `Iterator` interface.

In the first step of an orbit, the orbit always provides the start cell-tuple itself. Thus, there is always a cell-tuple in the iterator in the first step (which is `startCt` itself), i.e. `hasNext()==true` is always the case in the first invocation of `hasNext`.

The main functionality of an iterator over an orbit is to provide the *next* cell-tuple of the orbit. This functionality is realized in the `next` method of the `OrbitIterator` class (cf. Listing 16).

```
1.  if (this.ctStack.empty()) {
2.    throw new NoSuchElementException();
3.  }
4.  CellTuple result = this.ctStack.pop();
5.  CellTuple currCt;
6.  for (int i = 0; i < this.involutionsList.length; i++) {
7.    currCt = result.getInvolution(this.involutionsList[i])[124];
8.    if (!this.marked.contains(currCt.getID())) {
9.      this.ctStack.push(currCt);
10.     this.marked.add(currCt.getID());
11.   }
12. }
13. return result;
```

Listing 16: Implementation of OrbitIterator.next method (Java code)

The first step in iterating an orbit is to check whether the *stack of collected cell-tuples* is not empty (cf. line 1). If `ctStack` is empty, a `NoSuchElementException` is thrown by the `next` method. However, this will never happen if the API user always invokes `hasNext` in advance (which is a mandatory convention of iterator semantics). The top cell-tuple of `ctStack` is *popped* of the stack and assigned to be the `result` of this invocation of `next` method (in first step, this is the `startCt` itself).

Afterwards, all the involution steps that are registered in the `involutionsList` array of this `OrbitIterator`, are iterated one-by-one, conducting the appropriate involution step (depending on the integer value in the `involutionsList`) on the `result` cell-tuple and assigning the result of that operation to `currCt` (see lines 6 and 7 of Listing 16).

Finally, it is checked whether `currCt` is in the `marked` set. If `currCt` is not already in the `marked` set (i.e. has not already been marked), then `currCt` is pushed on `ctStack` and added to the `marked` set (i.e., `currCt` is marked). The `result` cell-tuple is given back as the return value of the method.

124 The `getInvolution` method of `CellTuple` class is a helper method that simply performs the involution which is defined by the integer number of the parameter value and returns the involution cell-tuple as the method's return value.

Orbits With Virtual Cell Barriers

As indicated earlier, the Topology Module provides the means to create an orbit with virtual cell barriers.[125] In order to create such orbit, an additional integer value (dim_{OL}) has to be provided. dim_{OL} defines the dimension of a *cell* at *object level* that cannot be exceeded by the orbit at *net level*. This is achieved by prohibiting the orbit to do an involution step that would "touch" one of the cell-tuples that lie outside the net level representation of the cell at object level (cf. Fig. 66 for an example based on $\langle \alpha_0, \alpha_1, \alpha_2 \rangle (d_{S(NL)})$).

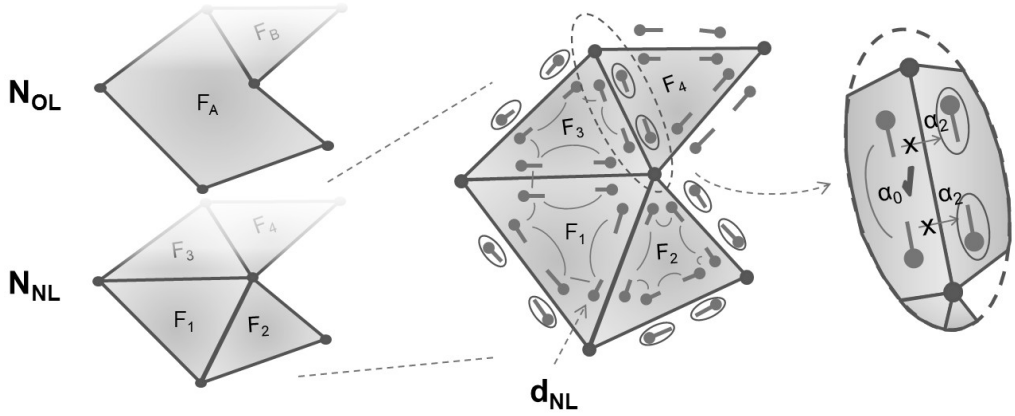


Fig. 66: Example case for OrbitIterator barrier cell-tuple list

The left hand side of Fig. 66 shows a 2-G-Map at net level with four faces and an $\alpha_0 - \alpha_1 - \alpha_2$ -orbit, starting at $d_{S(NL)}$. At object level, the whole complex is subdivided into two faces $F_{A(OL)}$ and $F_{B(OL)}$, with $F_{A(OL)}\{F_{1(NL)}, F_{2(NL)}, F_{3(NL)}\}$ and $F_{B(OL)}\{F_{4(NL)}\}$. If the constructor is now invoked with an arbitrary cell-tuple of $F_{A(OL)}$ ($d_{S(OL)}$), then the orbit will start with its "sibling" at net level $d_{S(NL)}$. Presuming $dim(OL)=2$ in this example set-up, the cell of object level that is not allowed to be left by the orbit is the *2*-cell (i.e. the face; in this example $F_{OL(A)}$). For the algorithm this means that at the α_2 -transitions between $F_{NL(3)}$ and $F_{NL(4)}$, the orbit simply omits the α_2 -involution and thus never can "break-through" from $F_{NL(3)}$ into $F_{NL(4)}$ (illustrated in detail view on right hand side of Fig. 66). So in this example, only the cell-tuples of the complex that belong to $F_{A(OL)}\{F_{1(NL)}, F_{2(NL)}, F_{3(NL)}\}$ are collected.

To implement this functionality in `OrbitIterator`, only some small changes in the `next` method are necessary. In the `for`-loop of Listing 16 (line 8), it has to be checked whether there exists a d_{OL} for d_{NL} . If d_{OL} exists, then the insertion of d_{NL} into `ctStack` is only

125 Realized in the third constructor of the `OrbitIterator` class.

allowed (`pushCurrCtOnStackAllowed=true`) if $\alpha_2(d_{OL})$ is not an outer boundary cell-tuple of $F_{A(OL)}$ (see Listing 17):

```
1.  if (this.ctStack.empty()) {
2.    throw new NoSuchElementException();
3.  }
4.  CellTuple result = this.ctStack.pop();
5.  CellTuple currCt;
6.  for (int i = 0; i < this.involutionsList.length; i++) {
7.    currCt = result.getInvolution(this.involutionsList[i]);
8.    boolean pushCurrCtOnStackAllowed = false;
9.    if (!this.marked.contains(currCt.getID())) {
10.     if (this.dimForObjLevel == -1) {
11.       pushCurrCtOnStackAllowed = true;
12.     } else {
13.       if (!ctsOnBoundary.contains(currCt.
14.            getInvolution(dimForObjLevel).getID())) {
15.         pushCurrCtOnStackAllowed = true;
16.       }
17.     }
18.   }
19.   if (pushCurrCtOnStackAllowed) {
20.     this.ctStack.push(currCt);
21.     this.marked.add(currCt.getID());
22.   }
23. }
24. return result;
```

Listing 17: Extension of OrbitIterator.next method to handle cell barriers (Java code)

This mode of `OrbitIterator` becomes especially useful when it comes to the constraints checking process of the editing algorithms (cf. Ch. 3.4).

3.3.2 Traversing Cells with the Help of Cell Iterators

To simplify the usage of the iterators over orbits that traverse all cell-tuples of a *certain, given* cell, it is useful to introduce an additional abstraction layer that eases the creation of such topological queries.

Designing Cell Iterators

A cell iterator concept can be defined on the basis of the orbit iterator notion. In analogy to the orbit iterator, an abstract cell iterator is also *iterable* and provides an *iterator* (Fig. 67).[126]

126 And thus also can conveniently be used in enhanced for-loops and lead to a stepwise processing when it comes to querying for the neighbourhood properties of cells (like demonstrated in previous section for orbit iterator).

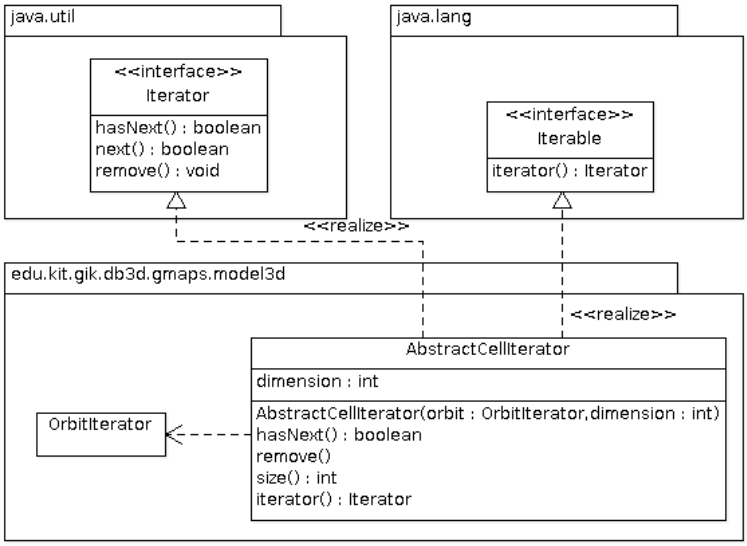


Fig. 67: AbstractCellIterator abstract class

The `AbstractCellIterator` class has been designed to partially realise the `Iterator` and `Iterable` interfaces. All cell iterators base on the same principle that they consume an `OrbitIterator` instantiation as well as a `dimension` integer value and then process an orbit of the given dimension. This means that cell iterators only consume a subset of all orbit iterators, indeed only those that can be identified to be of a certain dimension.[127]

All concrete cell iterators can be grouped into the *boundary cell iterators* (that are `NodeIterator`, `EdgeIterator`, `FaceIterator` and `SolidIterator`; cf. Fig. 68) and the *closure cell iterators* (cf. Fig. 71).

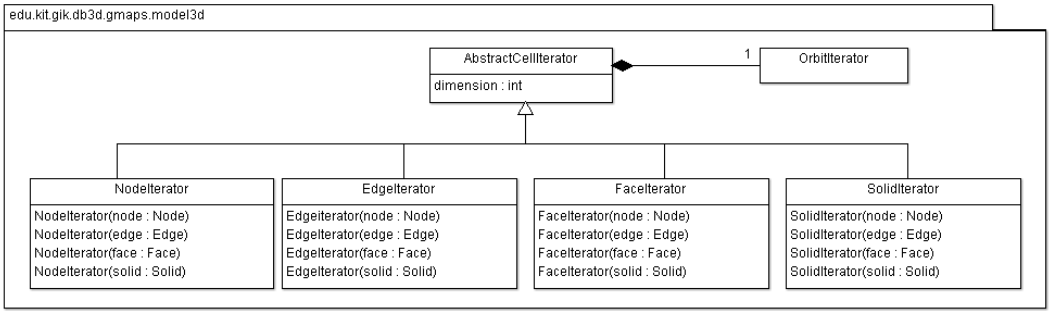


Fig. 68: Boundary cell iterator classes

All cell iterators internally forward topological queries to an orbit iterator. Cell iterators translate topological queries on cells into suitable orbits. Thus, every *cell iterator* has a reference to one instance of orbit iterator (inherited from `AbstractCellIterator`).

127 e.g. the orbit $\langle \alpha_0, \alpha_3 \rangle(d)$ cannot be an orbit of a cell iterator since it does not traverse the cell-tuples of a cell (of a certain dimension).

Internally, all the cell iterators always operate on an `OrbitIterator`. In order to create a *boundary cell iterator*, a cell (i.e. a node, edge, face, or solid) has to be provided. The cell iterators return all cells

- that are of the type that is indicated by the respective cell iterator's class designation and
- that are adjacent or incident (depends on the cell's dimension) to the cell that is given as parameter to the constructor of the cell iterator.[128]

Concept Details and Implementation Examples

A graphical representation of the cell iterators' functionalities is given in Fig. 69.[129]

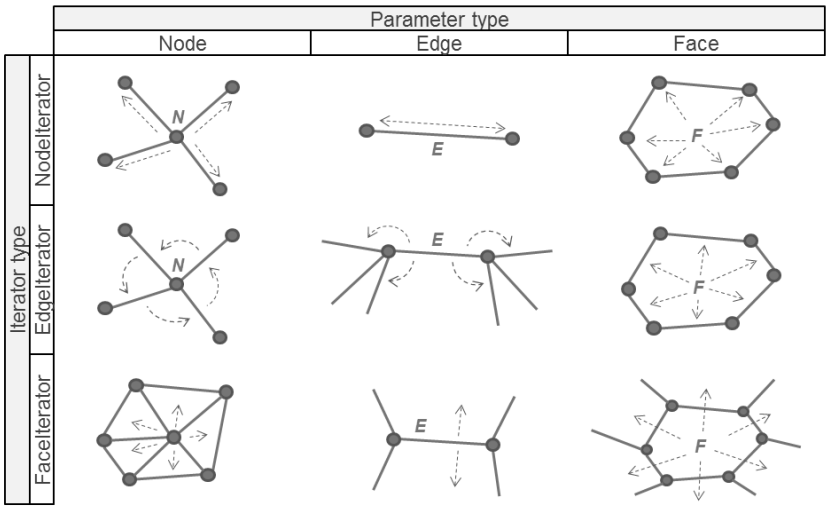


Fig. 69: Graphical representation of the respective iterating functionalities of cell iterators

Fig. 69 shows (for some example set-ups), which cells are provided by the different cell iterators, depending on the type/dimension of the cell iterator and depending on the type/dimension of the cell that is given to the cell iterator's constructor. The returned cells are indicated by the arrows in the depiction. For example, the illustration in the last row of the last column shows the iterating functionality of a `FaceIterator` that has been instantiated with an object of type `Face` as constructor parameter: the resulting cell iterator returns all faces that are adjacent to the given face. To formulate such a topological query that iterates all faces, adjacent to a given `face` object, the cell iterator API allows for a concise code (cf. Listing 18).

128 e.g. with the invocation of NodeIterator(Face) constructor, the instantiated iterator returns all nodes that are incident to the given face in an ordered sequence, whereas the invocation of FaceIterator(Face) constructor returns all faces that are adjacent to the given face

129 The SolidIterator has been omitted since the depiction would lead to ambiguities...

```
FaceIterator faceIterator = new FaceIterator(face);
for(Face adjacentFace : faceIterator){
  System.out.println(adjacentFace);
}
```

Listing 18: Usage of cell iterator API for the iteration of adjacent faces (Java code)

Internally, the several constructors of the cell iterators instantiate appropriate `OrbitIterators` (depending on which cell iterator constructor has been invoked) and pass it to the `AbstractCellIterator` constructor. This shall be exemplified by the constructors of `NodeIterator` class in Listing 19[130] (this is similar in all other constructors).

```
public NodeIterator(Node node) {
  super(new OrbitIterator(node.anyCellTuple, 0), 0);
}
public NodeIterator(Edge edge) {
  super(new OrbitIterator(edge.anyCellTuple, 1), 1);
}
public NodeIterator(Face face) {
  super(new OrbitIterator(face.anyCellTuple, 2), 2);
}
public NodeIterator(Solid solid) {
  super(new OrbitIterator(solid.anyCellTuple, 3), 3);
}
```

Listing 19: Implementation of translation between a cell iterator and orbit iterators (Java code)

Any concrete extension of `AbstractCellIterator` has still at least to implement the `next` method (of the `Iterator` interface).[131] On net level, on each step of the cell iterator, it needs to do two steps on the orbit and then query the cell of the demanded dimension.

For example, the iteration of faces that are adjacent to a given face f_g should be executed by instantiating a `FaceIterator` with f_g as constructor parameter. The inner implementation of the appropriate `FaceIterator` constructor instantiates the orbit $\langle \alpha_0, \alpha_1 \rangle (d_{f_g})$. Then, at every invocation of the `next` method of `FaceIterator`, the algorithm of the `next` method (only) needs to do an α_0-involution, ignore the resulting

130 In order to convey the idea, the concept presented in this chapter explains a simple model that has been used in an earlier version of *GMapsDb3dModule*. The latest procedure employs a more complex model in order to handle holes. Basically, the new cell iterators do a complete scan of a “big cell” (with 3-orbit) and search for all cell boundaries. This procedure is similar to the one of closure iterators, which are addressed below in this chapter.

131 The `hasNext` method cannot be realised in `AbstractCellIterator`, because the implementation of `hasNext` highly depends on the type of cell iterator.

cell-tuple and then, in the same step of `next`, do an α_1 -involution and return the α_2 -face of the resulting cell-tuple as the result of the `next` method's invocation (Listing 20[132]):

```
public Face next() {
  CellTuple ct = this.orbit.next();
  Face result = ct.alpha2.face;
  this.orbit.next();
  return result;
}
```

Listing 20: Algorithm (simplified) of next method of FaceIterator (Java code)

This principle also works for the other cell iterator instantiations in similar ways.

Cells of Object Level in Cell Iterators

For object level cells, the situation is more complicated. On object level it has to be considered that some incident/adjacent cells may appear more than once in an orbit. This is explained on three examples in Fig. 70.

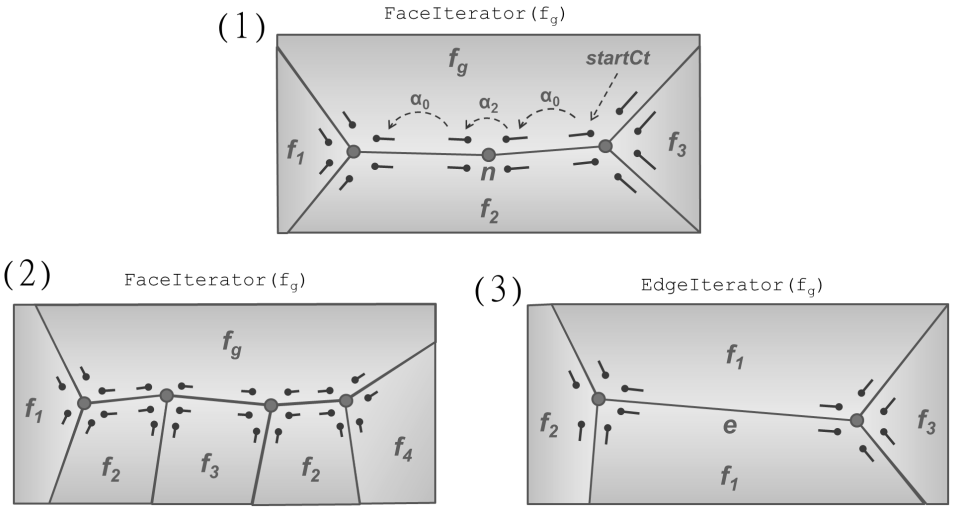


Fig. 70: Examples of cell iterators where adjacent/incident cells occur more than once

Illustration (1) of Fig. 70 shows a detail of a 2-G-Map, where a `FaceIterator(` f_g `)` shall iterate over all adjacent faces ($f_1, f_2, f_3, \cdots$) of f_g. Now the simple approach of iterating an $\langle \alpha_0, \alpha_1 \rangle (startCt)$ orbit and returning the face of an α_2 -involution at every second step of the orbit would not produce the desired result, since f_2 is incident to multiple edges along the orbit. A similar issue occurs if an adjacent face f_3 lies *in between* another adjacent face f_2 (as in the second illustration). The third illustration

132 The algorithm in the Listing starts with α_1 instead of α_0, according to the sequence, implemented in the algorithm of OrbitIterator, as explained in Ch. 3.3.1.

shows that a similar issue occurs not only with `FaceIterator` but also with `EdgeIterator(` f_g `)`, where $\langle \alpha_{0,} \alpha_2 \rangle (d_e)$ may return the same face f_1 on both sides of the edge.

As a consequence, all cell iterators additionally need to maintain a set of visited cells (field attribute `visited`) that has to be checked at every step of `next` (Listing 21 shows an extended version of the method of Listing 20):

```
public Face next() {
  Face result = this.orbit.next().alpha2.face;
  while (this.visited.contains(result)) {
    this.orbit.next();
    result = this.orbit.next().alpha2.face;
  }
  this.orbit.next();
  this.visited.add(result);
  return result;
}
```

Listing 21: Algorithm (simplified) of next method of FaceIterator (Java code)

The various cell iterators are used by the `getNeighbour<cells>` methods of the cell classes (Ch. 3.1.3). In fact, the implementation of the `getNeighbour<cells>` methods consists of nothing more than an invocation of the appropriate cell iterator – as exemplified by the implementation of the `getNeighbourEdges` method of `Face` class in Listing 22:

```
public EdgeIterator getNeighbourEdges() {
  return new EdgeIterator(this);
}
```

Listing 22: Implementation of getNeighbourEdges method of Face class (Java code)

Tabular Overview of Correlation Between Cell Iterators, Cells and Orbits

Every `getNeighbour<cells>` method uses a unique combination of cell iterator and constructor parameter. An overview is compiled in Table Table 2.

	NodeIterator	*EdgeIterator*	*FaceIterator*	*SolidIterator*
Parameter type	**Node (n)**			
OrbitIterator invocation	*OrbitIterator*(d_n , 0);			
Orbit definition	$\langle \alpha_{1,} \alpha_2 \rangle (d_n)(\langle \overline{\alpha_0} \rangle (d_n))$			
getNeighbour<cells> method of n	*Node.getNeighbourNodes()*	*Node.getNeighbourEdges()*	*Node.getNeighbourFaces()*	*Node.getNeighbourSolids()*
Parameter type	**Edge (e)**			
OrbitIterator invocation	*OrbitIterator*(d_e , 1);			
Orbit definition	$\langle \alpha_{0,} \alpha_2 \rangle (d_e)(\langle \overline{\alpha_1} \rangle (d_e))$			
getNeighbour<cells> method of e	*Edge.getNeighbourNodes()*	*Edge.getNeighbourEdges()*	*Edge.getNeighbourFaces()*	*Edge.getNeighbourSolids()*
Parameter type	**Face (f)**			
OrbitIterator invocation	*OrbitIterator*(d_f , 2);			
Orbit definition	$\langle \alpha_{0,} \alpha_1 \rangle (d_f)(\langle \overline{\alpha_2} \rangle (d_f))$			
getNeighbour<cells> method of f	*Face.getNeighbourNodes()*	*Face.getNeighbourEdges()*	*Face.getNeighbourFaces()*	*Face.getNeighbourSolids()*
	Solid (s)			
OrbitIterator invocation	*OrbitIterator*(d_f , 3);			
Orbit definition	$\langle \alpha_{0,} \alpha_{1,} \alpha_2 \rangle (d_f)(\langle \overline{\alpha_3} \rangle (d_f))$			
getNeighbour<cells> method of s	*Solid.getNeighbourNodes()*	*Solid.getNeighbourEdges()*	*Solid.getNeighbourFaces()*	*Solid.getNeighbourSolids()*

Table 2: Overview constructor invocation and functionality of cell iterator objects

Table Table 2 shows an overview of which cell iterator is used in combination with which cell type as its parameter in which `getNeighbour<cells>` method. Therefore, all cell iterators are listed in columns, whereas the cells are listed in rows. A combination of a cell iterator and a cell leads to a certain query for neighbouring cells (a certain `getNeighbour<cells>` query). For example, the `EdgeIterator` with a constructor parameter cell of type `Face` is used in the method `Face.getNeighbourEdges` (query for all incident edges of a face). Additionally, the table shows for each cell iterator/constructor parameter combination, which `OrbitIterator` constructor invocation is used and how the respective orbit is defined. In the given example (query for all incident edges of a face), a 2-orbit ($\langle \overline{\alpha_2} \rangle (d_f)$) is instantiated by invoking the constructor `OrbitIterator(` d_f `, 2)`.

Closure Cell Iterator Classes

As noted earlier, some cell iterators can be classified as *closure cell iterators* (see Fig. 71).

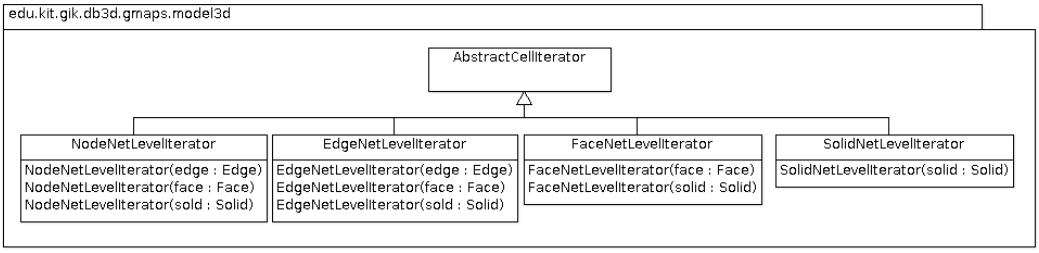


Fig. 71: Closure cell iterator classes

The closure cell iterators are similar to the boundary cell iterators despite that they do not iterate only the *boundary* cells of a given cell but also the "*inner*" cells of the given cell at net level[133]. For example, a `NodeNetLevelIterator` that is instantiated with a parameter object of type `Face` iterates all nodes at net level that are inside and at the boundary of the given object-level face (cf Fig. 72).

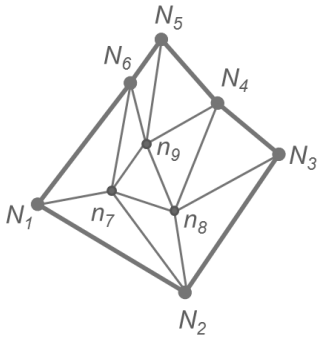


Fig. 72: Sample configuration of a face net at object and net level

Fig. 72 shows a sample net configuration with nodes $N_{OL}=\{N_1, N_2, N_3, N_4, N_5, N_6\}$ at object level that constitute a face $F_{OL}(N_{OL})$ at object level. The thicker lines in the illustration represent the object level while the thinner lines represent the net level. Thus, the nodes of the net level are comprised of $n_{NL}=\{n_7, n_8, n_9\}$ as well as of the siblings of the cells of N_{OL} on N_{NL}. While the boundary cell iterator `NodeIterator(` F_{OL} `)` would return N_{OL}, the closure cell iterator `NodeNetLevelIterator(` F_{OL} `)` returns n_{NL} and the siblings of N_{NL} on N_{NL}.

The closure cell iterators internally use `OrbitIterator` objects that are instantiated with the third constructor, i.e. orbit iterators that operate on net level and additionally never do

133 This is the reason why the naming of these iterators always contains a "NetLevel" part.

an involution step that would "breach" the boundary of the object level cell of the given dimension (Ch. 3.3.1).

Generally, the closure or net level cell iterators provide several constructors, defined as $r_j(c_i), [i, j \in \mathbb{N} | i \geq j]$, with c being the constructor parameter cell, r being the cell iterator type, i being the dimension of c and j being the dimension of r. However, some constructors are omitted since they would make no sense, such as `NodeNetLevelIterator(Node)`, since it makes no sense to query for all closure nodes of a node.

The presented classes and methods operate on a very basic architectural level. They can be used as a construction kit to create more complex, composite methods that manipulate/edit the cellular structure of the cell complexes. However, before such complex methods can be created, an additional level of abstraction should be conceived that allows for the formulation of constraints that can be used in such complex editing methods.

3.3.3 Finding the Shortest Path on a G-Map

One of the primary implementation objectives of the Topology Module is the improvement of the navigability on top of a network component. The best way to demonstrate the improved navigation capabilities is to implement an algorithm for finding shortest paths on top of the cell-tuple structure. Such an algorithm naturally places high demands on the navigation capabilities on top of a network. Moreover, the discovery of a shortest path is a relevant operation for a variety of applications – e.g. for the insertion of an edge into a face: in this case, the framework user specifies only the two nodes (start and target) between which an edge should be created, the framework has to determine the shortest path on the net-level and create the edge accordingly (Fig. 73).

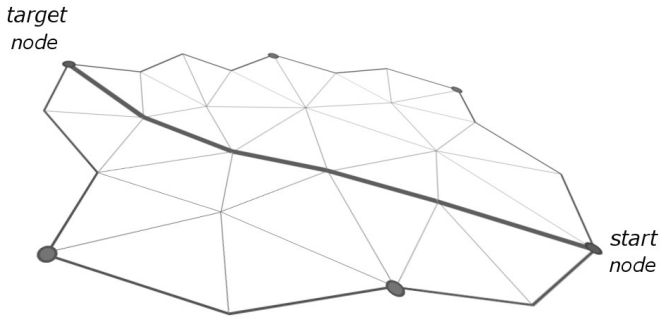


Fig. 73: Finding a path on net level (light lines) between two nodes of an object level face (thick lines)

A fundamental algorithm to determine a cost minimizing path between a start node and a target node is DIJKSTRA's algorithm (Dijkstra 1959) which is a search algorithm for *edge-weighted graphs*. The basic idea of the algorithm is to follow at each node only those edges

that comprise the lowest costs so far.[134] Arbitrary types of costs can be minimized by the algorithm, e.g. the gasoline cost of a journey or the strength of curvature of a path or the length (distance) of a path.

In the Topology Module, the determination of cost minimizing paths focuses on calculating the *minimal distance*.[135] The DIJKSTRA algorithm works only on edge-weighted graphs, thus the path finding operates in two steps:

1. the minimal costs to reach any node are calculated,
2. the path with the lowest overall cost is determined.

In fact, the first step is computed "backwards", i.e. starting at the target node and leading to the start node, and then the second step calculates the shortest path in the "forward direction".

Initially, the algorithm needs an empty list[136] of nodes that still have to be visited (`nodesToVisit`) and an empty list of nodes that already have been visited (`visitedNodes`). Furthermore, the algorithm needs an extended definition of a node. Therefore, a class `DNode` ("distance-node") is defined that extends the notion of node (class `Node`) by the specification of a distance value (cf. `distance` attribute of type `double` in Fig. 74).

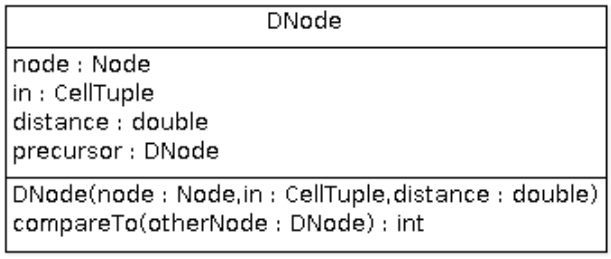


Fig. 74: Definition of DNode

Thereby, every `DNode` is a node that knows its *path distance* from the target node. In addition, every `DNode` memorizes its `precursor`; a `precursor` is defined as the `DNode` that the path finding algorithm identifies as the preceding node on a path. A `DNode` further

134 The implementation of the algorithm in the *GMapsDb3dModule* is loosely following (Waldura 2007).

135 The method is implemented as the (single) method of the *function object class* `Dijkstra`. A function object class is a class that provides only one method – thus its only reason for existence is to provide the functionality of that method. The method of the class is called `getShortestPath`. It takes the parameters `start` and `target` of type `Node` and returns an array of cell-tuple objects (`CellTuple`) that are "touched" by a sequence of involutions that represent the shortest path between the given `start` and `target` parameters; or it returns an *empty array* if no shortest path could be found.

136 Also FIFO

notes the "*in*"-cell-tuple of the node: this is the one cell-tuple of every node that shows into the direction of the `precursor`.

The full algorithm of the `getShortestPath` method is described in Listing 23.

```
Method: getShortestPath
Purpose: Finds the shortest path between the given start and target
nodes
Parameter: The two nodes (of class Node) start and target between which
the shortest path is to find
1. Add the target node itself to the nodesToVisit list[137]
2. While the list of the nodes that still are to visit
   (nodesToVisit) is not empty
3.   Poll a node from nodesToVisit and define the node as
     currentNode
4.   Put currentNode into the set of visited nodes
     (visitedNodes)
5.   Get all nodes that are adjacent to currentNode
6.   For every adjacent node
7.     If the adjacent node has not already been visited (is not
       part of the visitedNodes set)
8.       Calculate the Euclidean distance between currentNode
         and the current adjacent node as incrementalDistance
9.       Add incrementalDistance to the distance of currentNode
         as distanceToTargetNode
10.      If the adjacent node is not part of the nodesToVisit
         set
11.        Add the node to the nodes of the nodesToVisit list
12.        Update the distance value of the adjacent node with
           current distanceToTargetNode value and update the
           path of the adjacent node
13.      Else
14.        Compare the distance value of the adjacent node with
           distanceToTargetNode value
15.        If distanceToTargetNode is less than the distance
           value of the adjacent node
16.          Update the distance value of the adjacent node with
             current distanceToTargetNode value and update the
             path of the adjacent node
```

Listing 23: Pseudocode description of getShortestPath method of the Dijkstra class

The determination of adjacent nodes in step number 5 of Listing 23 is a costly operation in *db3dcore* of *quadratic runtime* but an inexpensive operation of *linear runtime* with the help of the cell-tuple structure (cf. Ch. 2.2).

Fig. 75 shows a simple example of a path finding scenario. Each of the four single drawings shows the state of the algorithm (and the state of the lists `nodesToVisit` and `visitedNodes` and the nodes that are adjacent to the current node) at the second execution step of Listing 23.

137 This is the natural precondition for the algorithm to work since it needs at least one node that has to be visited.

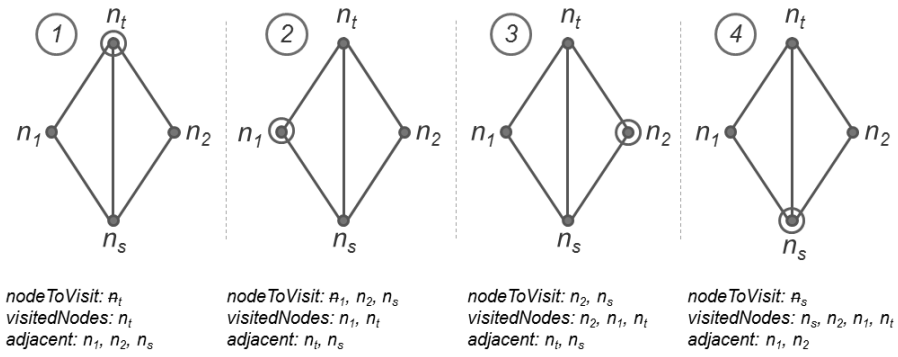


Fig. 75: Simple example of a path finding scenario

In the first step (depiction 1), the algorithm starts with the *target node* n_t (n_t is active). The node is removed from the `nodesToVisit` list and added to the `visitedNodes` list. The adjacent nodes of n_t are identified, which are n_1, n_2, and n_s and put onto the `nodesToVisit` list. Then the EUCLIDEAN distances between n_t and all adjacent nodes are calculated and memorized. The `precursor` of nodes n_1, n_2, and n_s (*start node*) are set to n_t. Next (depiction 2), node n_1 is set as the active node, it is removed from `nodesToVisit` list and added to `visitedNodes` list. The nodes that are adjacent to n_1 are identified, which are n_t and n_s. Of the adjacent nodes, only the nodes that have not already been visited (that are not already on the `nodesVisited` list) are further processed: in this case only n_s is further processed. The distance of n_s to n_t along n_1 node is calculated and compared to the direct distance between n_s and n_t. Since the straight distance between n_s and n_t is smaller than the distance along node n_1, the `precursor` of n_s is not changed: it still directly points from n_s to n_t. The same steps are undergone for node n_2 (depiction 3). Finally (depiction 4), n_s is evaluated. Since all adjacent nodes of n_s have already been visited, there is no processing necessary anymore. The algorithm terminates, since there are no more elements in the `nodesToVisit` list.

At this point, an edge-weighted graph has been created on top of the net component. In the next step, the shortest path can be calculated, simply by following the `precursor` of every node, starting at n_s and collecting the path cell-tuples at every node in a result list:

```
17. Get the start node and call it currentNode
18. While currentNode is defined (not null)
19.   If currentNode has a precursor node
20.     Add the cell-tuple of currentNode (that is pointing to
        the precursor) to the result list
21.     Add the α0 cell-tuple of the cell-tuple of step 18
        (i.e. the cell-tuple of the adjacent node of
        currentNode) to the result list
22.   Assign the precursor of currentNode to currentNode
```

Listing 24: Collecting cell-tuple of shortest path in a result list

Once the `while`-loop of the 18. step of Listing 24 quits, all cell-tuples of the shortest path from n_s to n_t are collected in a `result` array and given back as the method's return value. To be precise, these are the cell-tuples that are “leading in and leading out” of the nodes of the shortest path.

Due to the complexity of the geo-objects that may be modelled in *DB4GeO*, the writing of an algorithm for navigation on that geo-objects is easier with the cell-tuple structure. By introducing this structure, even the complexity of incidence/adjacency graphs[138] that have nodes with high *degrees* could be reduced so much that an intelligible modelling became feasible.

3.4 Methods that Manipulate the Cellular Structure

This chapter will describe the algorithms that are used to edit the cellular structure of the cell nets. They have already briefly been introduced in Ch. 3.1.8 as methods that are required by the `EditableCellNet3dCompLevel` interface. The editing algorithms always operate two-staged:

1. The algorithms check whether the intended editing of the cell net is permissible – in the sense that no inconsistent state of the cell net arises from the editing operation. This is achieved through a process that checks several constraints that are defined individually for each editing method.
2. The actual alteration of the cell net is performed.

In the following sections these two steps are exemplary explained for the editing methods of `FaceNet3dCompLevel`). A systematic evaluation of editing operations and constraints is not provided since this is out of scope of this thesis. However, such evaluation would be of particular value and should be performed in future.

3.4.1 Method to Insert a Node on a Face Net Component

A basic editing operation for a face net component is the operation to insert a node into an already existing edge. In order to insert a node, two elements have to be provided which are a node at net level (n_{NL}) and an edge at object level (e_{OL}). n_{NL} is used as a “template” to create a new node at object level (n_{OL}) as its “sibling”. “Sibling” means here that the geometric part of n_{OL} shall be equal to the geometric part of n_{NL} (cf. right illustration of Fig. 76).

138 The graph that is meant here is the graph that is mentioned in Ch. 2.2, where the nodes are the d-cells and the edges are the connections betweens the cells.

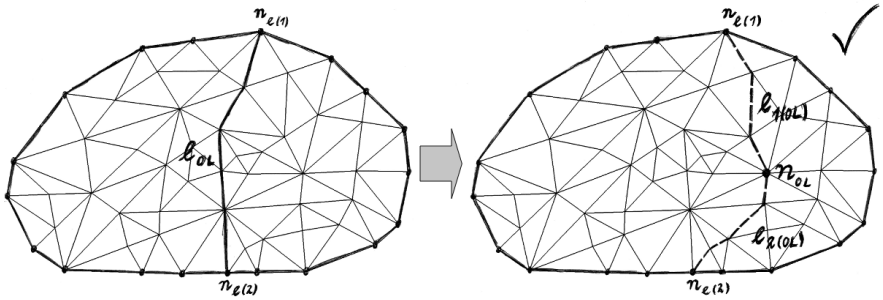


Fig. 76: Example of insertNode, represented on object and on net level (left: before operation, right: after operation)

Fig. 76 shows an example of an `insertNode` operation where the left illustration shows a face net component before editing and the right illustration shows the same face net component after the editing process.[139] In the figure, the thick lines and dots represent edges and nodes at object level while the thin lines represent edges at net level.[140]

In the left illustration, e_{OL} represents the edge that is passed to the `insertNode` method as the second parameter in this example set-up. The geometric representation of e_{OL} runs along the shortest path (on top of the triangle net of the net level) between the nodes $n_{e(1)}$ and $n_{e(2)}$ (i.e. $e_{OL}(n_{e(1)}, n_{e(2)})$).

The `insertNode` method splits e_{OL} into $e_{1(OL)}(n_{OL}, n_{e(1)})$ and $e_{2(OL)}(n_{OL}, n_{e(2)})$ (cf. right illustration of Fig. 76). This leads to a re-computation of the path of the geometric representation of the edge. Two new shortest paths are computed (with the help of the `shortestPath` method)[141] – one between $n_{e(1)}$ and n_{OL} and one between $n_{e(2)}$ and n_{OL}. These shortest paths are stored nowhere permanently but are only computed to perform consistency checks before conducting the operation.

There are several consistency conditions that have to be verified before a node can be inserted into an edge. Beginning with obvious and simple verifications such as

- the check whether the given n_{NL} and e_{OL} actually exist (i.e. are not `null`), proceeding with checks
- whether n_{OL} is already part of another edge, or
- whether n_{NL} and e_{OL} are part of the cell net component on which the `insertNode` method has been invoked.

Such consistency requirement can be checked efficiently due to underlying model of the Topology Module. For example, the last requirement can be checked easily and fast, since

139 Though, this could also depict two arbitrary faces somewhere inside a more complex face net component.

140 i.e. both levels are shown here superimposed

141 c.f. Ch. 3.3.3

every object of a realising class of `CellNet3dCompLevel` interface has ordered sets of cells – these sets can efficiently be queried for whether they contain a certain cell.

The next consistency condition to be checked is whether n_{OL} would lie in one of the both faces (f_1 and f_2) that are incident to e_{OL} after the operation (cf. illustration *a)* in Fig. 77[142]).

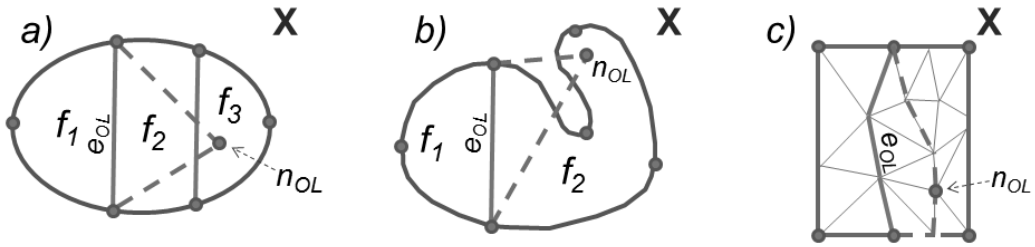


Fig. 77: Examples of illegal states for insertNode method

If the node lies neither in f_1 nor in f_2 but in another face (like in f_3 in the illustration), the insertion operation will not perform (the `insertNode` method aborts and returns an `Exception`), since the intended operation in such set-up is actually not `insertNode` but a more complex operation that is a *composite operation* of a sequence of editing operations.

The presented consistency condition check is a suitable example of a complex topological query, but yet is simple to formulate due to the well defined model of the Topology Module. In Listing 25, for example, the `boolean` variable $n_{NL}^{ispart?}$ stores the result whether n_{NL} is a *part of the neighbour faces* of e_{OL} .

```
1. boolean n_NL^ispart? = false;
2. for(Face f_eOL : e_OL.getNeighbourFaces()){
     NodeNetLevelIterator nodeNlIt =
           new NodeNetLevelIterator( f_eOL );
3.   for(Node nodeTemp : nodeNlIt){
4.     if(nodeTemp == n_NL ){
5.       n_NL^ispart? = true;
6.       break;
7.     }
8.   }
9. }
```

Listing 25: Topological node-in-face query in blended pseudo code

142 In the illustrations of Fig. 77 the underlying triangle net structure is hidden for simpler reception. However, it should always be assumed to be given

To determine whether n_{NL} is a part of the neighbour faces, first, the neighbour faces of e_{OL} are queried (line *2*). For each neighbour face of e_{OL} (line *3*), all nodes of $f_{e_{OL}}$ at net level (use of closure cell iterator) are checked for equality with n_{NL} (line *4*). If an equality is detected, the algorithm terminates successfully (line *5* and line *6*), i.e. with the result that n_{NL} is in a neighbour face of e_{OL}.

Another consistency condition is that the paths of the geometric representation of $e_{1(OL)}$ and $e_{2(OL)}$ are not allowed to intersect the geometric representation of any other edge of the neighbouring faces (cf. illustration *b)* in Fig. 77), since this would also yield a composite operation or even be impossible in the case that the new edges run through the "outer void" (due to the underlying functionality of DB4GeO).

Furthermore, it is not permissible that two or more edges at object level share the same geometric representation (depicted in illustration *c)* in Fig. 77). This is because it would break the *refinement* postulate (cf. Ch. 2.4.3) and lead to a situation that cannot be modelled as correlation between object and net level without ambiguities.

After the constraints for `insertNode` have been checked, the alteration of the cell net itself is performed (Fig. 78).

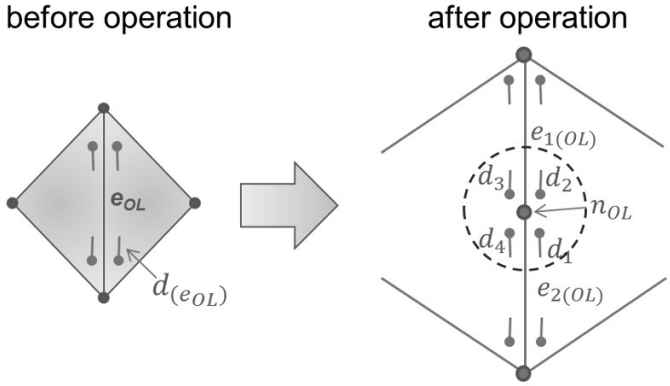


Fig. 78: Adding new cell-tuples when performing insertNode method

During the operation of inserting a new node into an existing edge, the edge is split in two parts by the node. For the node at net level (n_{NL}), a new "mirror" node at object level is created. The given node at net level is be linked to the newly created node at object level as its *higher level* of detail "sibling".

In the first step of the process, a new node at object level n_{OL} is instantiated that is the "sibling" of n_{NL}. Then, two new edges $e_{1(OL)}$ and $e_{2(OL)}$ are created at object level. An arbitrary cell-tuple (d_e) of e_{OL} is assigned to be a cell-tuple of $e_{1(OL)}$ from now on (i.e. before: $d_e = d(e_{OL})$, after: $d_e = d(e_{1(OL)})$; cf. Fig. 78). The cell-tuple (d_α) of an α_0-involution of $d(e_{1(OL)})$ is assigned to be a cell-tuple of $d(e_{2(OL)})$ (i.e.: before: $d_\alpha = \alpha_0(d(e_{OL}))$, after: $d_\alpha = d(e_{2(OL)})$).

Since the preceding edge will be deleted and instead two new edges are inserted, all *1-cell* (edge) *references* of affected cell-tuples need to be exchanged. Effected cell-tuples are $d(e_{OL}), \alpha_0(d(e_{OL})), \alpha_2(d(e_{OL}))$ and $\alpha_0(\alpha_2(d(e_{OL})))$. For all these cell-tuples, the edges are reset ($e_{OL} \rightarrow e_{1(OL)}$ and $e_{OL} \rightarrow e_{2(OL)}$).

The splitting of an edge has the consequence that four new cell-tuples of n_{OL}, which are d_1, d_2, d_3, d_4, are introduced into the structure. The cell-tuples are instantiated and consistently linked with the appropriate cell, such that every cell-tuple represents a valid incidence graph in the end of the process (according to the concepts introduced in Ch. 2.2 and Ch. 2.3). The newly created cell-tuples are embedded into the existing cell-tuple structure by setting all missing involution links (such as $\alpha_0(d_1) = d(e_{OL})$) and all back references (such as $\alpha_0(d(e_{OL})) = d_1$).

As the situation between net level and object level also changes when a node is inserted, the links between net and object level cell-tuples are updated. The cell-tuples of n_{NL} are re-linked to the cell-tuples of n_{OL} (for example $higher(d_1) = d(n_{NL})$ and its back-reference $lower(d(n_{NL})) = d_1$).

Finally, all newly created cells and cell-tuples are added to the appropriate component's indices (`cellTupleIndex`, `edgeIndex` etc.), e_{OL} is removed from the edge index and n_{OL} is returned as the methods return value.

3.4.2 Method to Insert an Edge on a Face Net Component

Another basic editing operation for a face net component is the operation to insert a new edge e into an already existing face of a face net component. e is inserted between two given nodes n_1, n_2 (cf. Fig. 79).

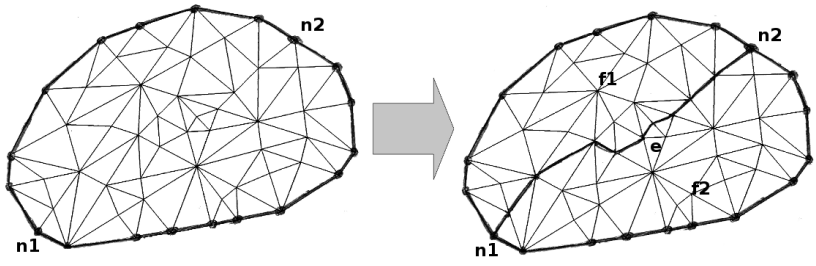


Fig. 79: Example of insertEdge operation, represented on object and on net level

Fig. 79 shows an example of an `insertEdge` operation where the left illustration shows face net component before editing and the right illustration shows the same face net component after the editing process. In the figure, the thick lines and dots represent edges and nodes at object level while the thin features represent objects at net level.

To perform the `insertEdge` operation, three parameter values are needed, one object of type `Face` f and two objects of type `Node` n_1, n_2. n_1, n_2 have to be the nodes that will constitute the boundary nodes of the new edge. f has to be the face that is split by the edge insertion operation into two new faces f_1, f_2. Obviously, n_1, n_2 have to be (boundary) nodes of the same face (f), otherwise the new edge would intersect with existing edges. A shortest path is computed between n_1 and n_2 (with the help of the `shortestPath` method). The shortest path is not stored permanently but is only computed to perform consistency checks before conducting the operation.

There are several consistency conditions that have to be verified before an edge can be inserted into a face. Beginning with obvious and simple verifications like the check whether given n_1, n_2 and e actually exist (i.e. are not `null`), proceeding to checks whether n_1, n_2 and e are part of the cell net component on which the `insertEdge` method has been invoked.

Another consistency condition to be checked is whether e would intersect or partly overlay any other edge (cf. Fig. 80).

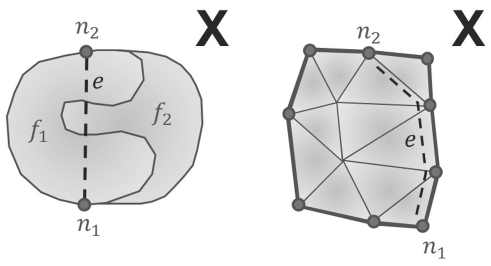


Fig. 80: Examples of illegal states in insertEdge method

The left illustration of Fig. 80 shows an example, where if e (dotted line) would be inserted, it would (twice) intersect a boundary edge of f (which would be an illegal operation). The right illustration shows the object level (thick lines) and the net level (thin lines) of an example set-up. Due to the spatial configuration of the net level, e would partly overlay an already existing boundary edge of f on object level (which would be an illegal operation as well).

After the discussed constraints (and some other simpler tests) have been checked, the alteration of the cell net itself is performed (Fig. 81).

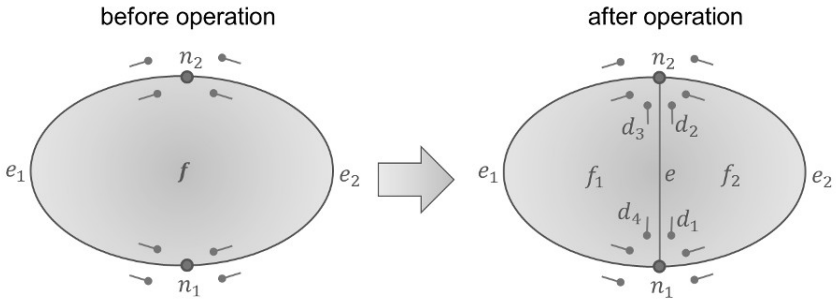


Fig. 81: Adding new cell-tuples when performing insertEdge method

In the insertion operation, first a new object *e* of type `Edge` and two new objects f_1, f_2 of type `Face` are created on object level and put onto the *edge* and *face index* of face net component of object level. Next, the shortest path on net level between start node n_1 and target node n_2 is computed. The shortest path is needed in order to find the cell-tuples at net level that are *hierarchically linked* to the newly created cell-tuples of *e* (d_1, d_2, d_3, d_4) at object level. All cells (i.e. $n_1, n_2, e, f_1, f_2, s_U$) are set on the newly created cell-tuples as well as all involutions (i.e. $\alpha_0, \alpha_1, \alpha_2, \alpha_3$). For example, α_1 of (n_1, e_2, f, s_U) now points to d_1 instead to (n_1, e_1, f, S_U) (as it did before operation). Finally, all face references of all cell-tuples of f are reset to f_1 and f_2 respectively. Therefore, a 2-orbit around f is performed in order to distinguish cell-tuple that belong f_1 from cell-tuples that belong to f_2.

3.4.3 Method to Remove Node and Edge From Face Net Component

Removing Nodes From Edges

The inverse operation to adding a node into an edge is to remove an existing node from an edge. In order to remove a node, one element has to be provided which is the node *n* that has to be deleted. *n* lies in between (connects) exactly two edges e_1, e_2. If *n* is removed, then e_1, e_2 are merged into a new edge *e*. The `removeNode` method is the inverse operation to the `insertNode` method. Thus the figures used in describing the `insertNode` method also apply here. For an illustration of an exemplary operation process, depicted on object and net level, you may confer to Fig. 76 and read it backwards.

As with the insertion methods, also in the `removeNode` method, constraints have to be checked before the actual delete operation can be performed. First, simple constraints like that the given `Node` object cannot be `null` or that the given node has to be part of the net component that the `removeNode` method is invoked on, are checked. Then more complex constraints are evaluated, such as that e is not allowed to intersect or partly overlay any other edge (cf. Fig. 80), or that a node may be removed only if exactly two edges are incident to the given node (cf. left illustration of Fig. 82).

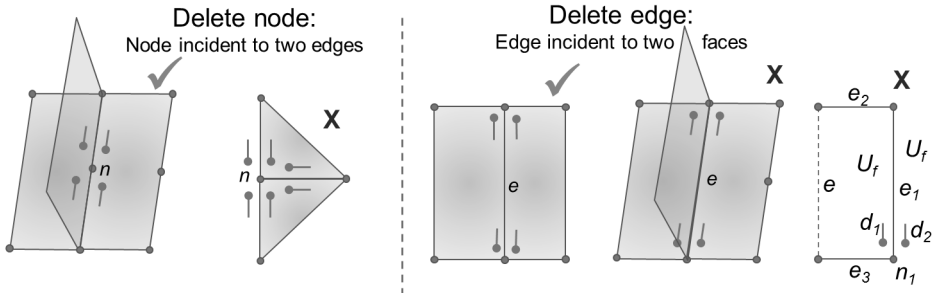


Fig. 82: Examples of valid and invalid spatial configurations for node and edge delete operations

In Fig. 82, a valid spatial configuration is depicted on the left side in "delete node" section, since here, exactly two edges are incident to *n*. An invalid spatial configuration is depicted on the right side in "delete node" section, since more than two edges (here: three edges) are incident to *n*. This is an important validity check for node removal, since an intersection between two edges always causes an intersection node. Thus, a deletion of a node in an intersection of more than two edges leads to an invalid edge intersection without an intersection node. As an advantage of the Topology Module framework, the validity check, whether a node is incident two exactly two edges is quite simple, following the code of Listing 26:

```
boolean isExactlyTwoEdges = false;
isExactlyTwoEdges = node.getNeighbourEdges().size() == 2;
```

Listing 26: Checking for edge neighbourhood properties of a node (Java code)

In Listing 26, `isExactlyTwoNodes` variable contains BOOLEAN value that encodes whether the node to delete is incident to exactly two edges (`true`) or not (`false`).

After the discussed constraints (and some other simpler tests) have been checked, the alteration of the cell net itself is performed. The remove node operation performs similar steps as the insert node operation, but in a backwards manner. Explained on the basis of Fig. 78, first, a newly created e_{OL} is added to the edge index. The edge references of cell-tuples $\alpha_0(d_1),\ldots,\alpha_0(d_4)$ are reset to e_{OL}. The α_0-involution of cell-tuple $\alpha_0(d_1)$ is re-linked such that $\alpha_0(\alpha_0(d_1))=\alpha_0(d_2)$. This process is similarly repeated for the other cell-tuples $d_2,\ldots,d_4$. Finally, all cell-tuples that contain n_{OL} (d_1,d_2,d_3,d_4) are removed from cell-tuple index. Then the cells n_{OL}, $e_{1(OL)}$, and $e_{2(OL)}$ are also removed from the indices.

Removing Edges From Faces

The inverse operation to adding an edge into a face is to remove an existing edge *e* from a face. *e* lies in between (connects/separates) exactly two faces f_1, f_2. If *e* is removed, then

f_1, f_2 are merged into a new face f. The `removeEdge` method is the inverse operation to the `insertEdge` method. Thus, the figures used in describing the `insertEdge` method also apply here. For an illustration of an exemplary operation process, depicted on object and net level, you may confer to Fig. 79 and read it backwards.

As in the insertion method, also in the `removeEdge` method, constraints have to be checked before the actual delete operation can be performed. Beside trivial constraints like that the given `Edge` object cannot be `null` or that the given edge has to be part of the net component that the `removeEdge` method is invoked on, another important constraint is that an edge may be removed only if exactly two faces are incident to the given edge (cf. left side in "delete edge" section of Fig. 82). In Fig. 82, a valid spatial configuration for edge deletion is depicted on the left side in "delete edge" section. Here, exactly two faces are incident to *e*. Invalid spatial configurations are depicted in the centre and on the right side in "delete edge" section. The centre depiction shows a spatial configuration where more than two faces (in this case: three faces) are incident to the edge that shall be deleted. This is an important validity check for edge removal, since an intersection between two faces always causes an intersection edge. Thus, a deletion of an edge in an intersection of more than two faces leads to an invalid face intersection without an intersection edge.

The right depiction of Fig. 82 shows a spatial configuration where the edge *e* to be deleted is part of a face that is surrounded by U_f. The deletion of *e* converts the inner face into U_f which leads to an inconsistent cell-tuple structure (non-manifold). The cell-tuples d_1 and d_2 are not unique any more after the operation but represent the same cell-tuple (n_1, e_1, U_f, U_S).

After the discussed constraints (and some other tests) have been checked, the alteration of the cell net itself is performed. The remove edge operation performs similar steps as the insert edge operation, but in a backwards manner. Explained on the basis of Fig. 81, first, a newly created face f_{OL} is added to the face index. The face references of cell-tuples $\langle \alpha_2 \rangle(d_1)$ and $\langle \alpha_2 \rangle(d_4)$ are reset to f_{OL}. The α_2-involution of cell-tuple $\alpha_2(d_1)$ is re-linked such that $\alpha_2(\alpha_2(d_1)) = \alpha_2(d_4)$. This process is similarly repeated for the other cell-tuples $d_2, \ldots, d_4$. Finally, all cell-tuples that contain *e* (d_1, d_2, d_3, d_4) are removed from cell-tuple index. Then the cells *e*, f_1, and f_2 are also removed from the indices.

So far, the concepts, inner design principles, construction and editing operations of the Topology Module have been described that add an abstraction layer to the DB4GeO database architecture to manage the topology of arbitrary 2D and 3D cells. The module utilizes the concept of interconnected net and object levels of cell nets. However, the concept of subdividing cell net components into net and object level is also a precursor to a concept that allows for the modelling of arbitrary numbers of hierarchy levels – not only two levels.

3.5 Management of Levels of Detail of Cell Net Components

The modelling of several levels of detail of data is a pivotal task, particularly in applications dealing with geoinformation.[143] The main scheme in this approach is to extend the already implemented concepts and to reuse much of the development efforts. The classes that are designed for the management of the LoD are based on the net and object level architecture described in Ch. 3.1.8. The basic idea of the LoD approach of the Topology Module is to allow for the insertion of any number of additional detail levels in between the object and the net level. These additional detail levels behave more "object-level-like" than "net-level-like", since they also are *editable* – like the object level (cf. Fig. 83).

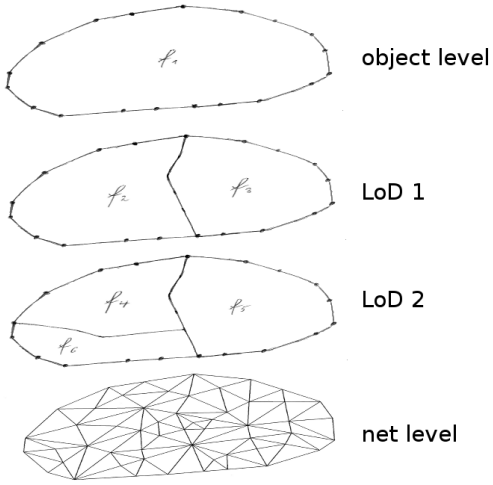


Fig. 83: Simple face net example of several LoD

Fig. 83 shows a simple sample configuration of a face net component with several LoD (the illustration can be interpreted as four layers of LoD, which are depicted simultaneously one over the other). It is also the result of an editing session that was performed on the different LoDs. The session is described in the following section.

At object level the face net component consists of one face f_1 (see top of illustration). The net level shows the triangle net structure of the face net component (bottom of illustration). The two face net components *LoD 1* and *LoD 2* are additional levels of detail that are forming *intermediate steps* between the net and the object level.[144] These intermediate levels are derived from copies of existing levels. After the copies of existing levels are created, they are edited. At *LoD 1*, a new edge is inserted that splits f_1 into the

143 For some thoughts on the application relevance of systems that are capable of managing multiple LoDs, see Ch. 2.4 and (Butwilowski 2007, 13 et seq.)

144 Counting starts at 1 with the lowest level of detail after the object level (the object level itself is actually defined as LoD 0)

two new faces f_2 and f_3. So f_2 and f_3 are cells that only exist at *LoD 1*. At *LoD 2*, again a new edge is inserted and f_2 is split into the two faces f_4 and f_6. Thus, f_4 and f_6 only exist at *LoD 2*. Although face f_5 of *LoD 2* is identical to f_3 of *LoD 1*, this two cells are separate instances. This behaviour is intentional by design, since it is an objective to be able to distinguish similar cells at different LoD in order to assign different property values to the equivalent cell on different levels.

Before the algorithmic characteristics of the model are described, an overview of the class model[145] of the chosen approach is given in the following section (cf. Fig. 84).

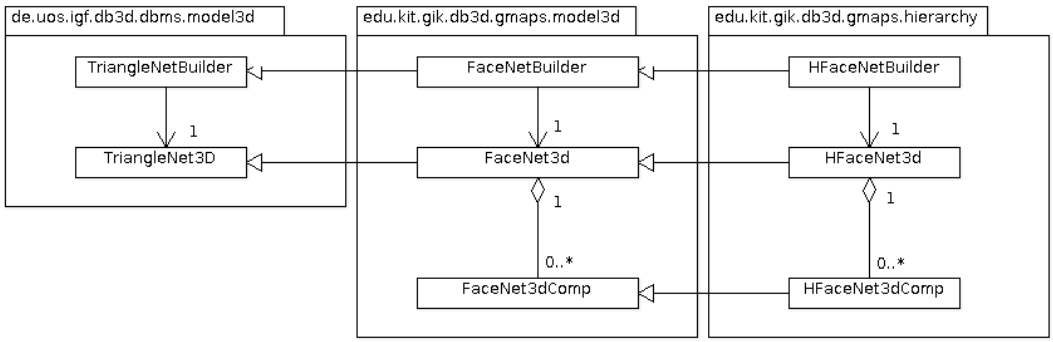


Fig. 84: Architecture of hierarchical net builder as an extension of the cell net builder architecture

Fig. 84 is based on Fig. 49 of Ch. 3.1.6 and shows, how the classes of the Geometric Model, the `gmaps.model3d` package and the `gmaps.hierarchy` package relate to each other. While Fig. 49 only shows how the classes of the `gmaps.model3d` package extend classes of the Geometric Model, Fig. 84 also shows how the classes of the `gmaps.hierarchy` package further extend classes of the `gmaps.model3d` package and thus reuse some of the already available functionality and extend them with LoD functionality.[146]

The most important design decisions are that the `HFaceNetBuilder` class extends the `FaceNetBuilder` class, the `HFaceNet3d` class extends the `FaceNet3d` class and the `HFaceNet3dComp` class extends the `FaceNet3dComp` class. The same applies to curves and solids. From this, the instantiation process of an “LoD enabled” cell net component is analogue to the instantiation process of a “simple” cell net component; the required code has been presented in Listing 3: at first, an `HFaceNetBuilder` is constructed by providing it the triangles of a net as parameter in an array through its `addComponent` method.[147] The `HFaceNetBuilder` returns an *h*-face net (an instance of `HFaceNet3d`)

145 All classes that provide the functionality for managing several hierarchy levels of cell components are gathered in the separate package `edu.kit.db3d.gmaps.hierarchy`. The set of classes of the hierarchy package is analogue1 to the classes of the above discussed `...gmaps.model3d` package

146 The class schema is similar for solid nets/tetrahedral nets

147 The addComponent method of HFaceNetBuilder overrides the homonymous methods of

through its `getTriangleNet` method. An *h*-face net consists of an arbitrary (*0* to unlimited) number of *h*-face net components (objects of type `HFaceNet3dComp`).

Fig. 85 shows a more detailed view, concerning the `HFaceNet3dComp` class.

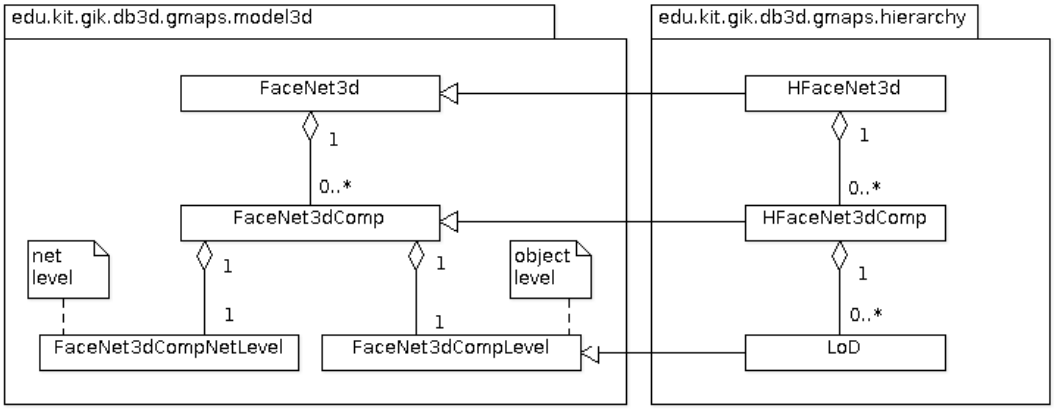


Fig. 85: Relationship between the classes that build the net level, object level and LoD

Fig. 85 shows that an *h*-face net component consists of an arbitrary number of levels of detail (i.e. objects of type `LOD`). A level of detail is a face net component level (extends the `FaceNet3dCompLevel` class). Since a `FaceNet3dCompLevel` is an editable cell net component level (i.e. it is a realisation of the `EditableCellNet3dCompLevel` interface), also a level of detail is an editable cell net component level (cf. Fig. 86).

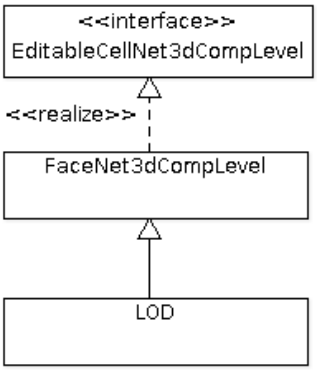


Fig. 86: Class LOD is a realisation of EditableCellNet3dCompLevel interface

Since `HFaceNet3dComp` is a `FaceNet3dComp`, every instance of `HFaceNet3dComp` consists of exactly one `FaceNet3dCompNetLevel` (the net level) and exactly one

FaceNetBuilder and of TriangleNetBuilder

`FaceNet3dCompLevel` (the object level) as well as an arbitrary amount of additional LoD.

When the API user has instantiated an "LoD enabled" face net by the means of the `HFaceNetBuilder`, that face net at first only consists of one net level, one object level and no levels of detail. Only then the user may begin to create additional LoD. To do so, the user has to utilize methods of the `HFaceNet3dComp` class (cf. Fig. 87).

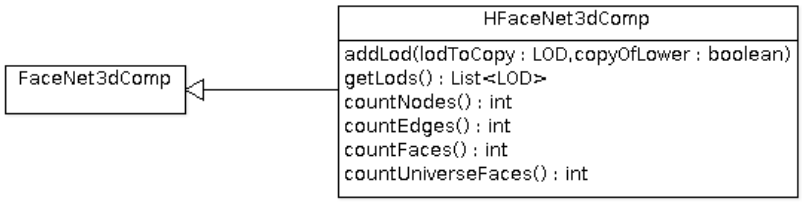


Fig. 87: Additional methods of HFaceNet3dComp class

Fig. 87 shows the methods of `HFaceNet3dComp` class. `HFaceNet3dComp` class inherits the behaviour (i.e. all methods) of `FaceNet3dComp` class and adds methods for appending and retrieving additional LoD. Furthermore, the `HFaceNet3dComp` class reimplements and overrides all the `count<cells>`-methods of `FaceNet3dComp` class since these methods return the cell numbers of the whole component – i.e. of all detail levels together. `FaceNet3dComp` only had to consider net and object level when calculating cell numbers but `HFaceNet3dComp` also needs to take into account the cells of all intermediate LoD. Thus, the `count<cell>`-methods need to be overridden.

To insert a new LoD into an *h*-face net component, the `HFaceNet3dComp` class provides an `addLod` method that takes two method parameters, an `lodToCopy` of `LOD` type (LOD_{master}) and the `boolean` value `copyOfLower`. LOD_{master} is the LoD of which the topological structure is copied. It is an exact copy where for every existing cell, a new cell (a new instance) is created on the new LoD (LOD_{copy}). The `boolean` parameter `copyOfLower` indicates whether LOD_{copy} is created as a copy of a *lower* LOD_{master} or as a copy of a *higher* LOD_{master} (cf. Fig. 88).

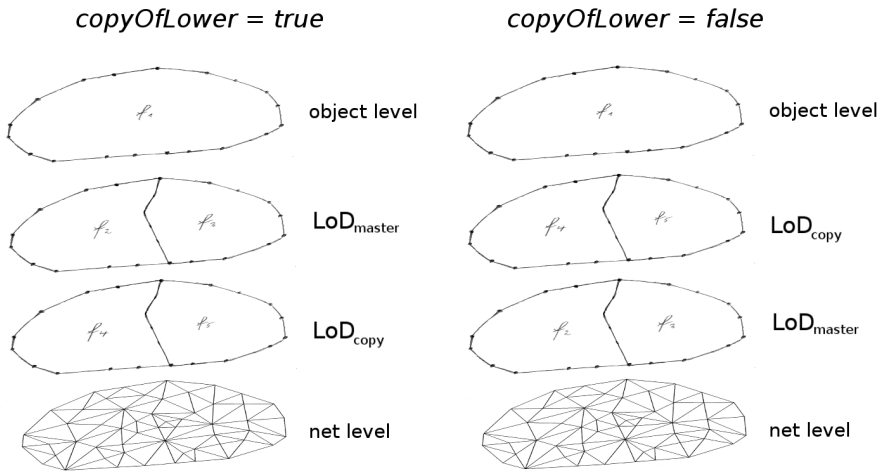


Fig. 88: Effect/purpose of copyOfLower parameter (on the LoD copy process)

If `copyOfLower` is `true`, then LOD_{copy} is created as a copy of a *lower* LOD_{master} (i.e. LOD_{copy} is defined as a *higher* LoD than LOD_{master}) if otherwise `copyOfLower` is `false` then LOD_{copy} is created as a copy of a *higher* LOD_{master} (i.e. LOD_{copy} is defined as a *lower* LoD than LOD_{master}).

The `LOD` class itself is a subclass of `FaceNet3dCompLevel` class and thus inherits its behaviour but also adds some `LOD` class specific methods (cf. Fig. 89).

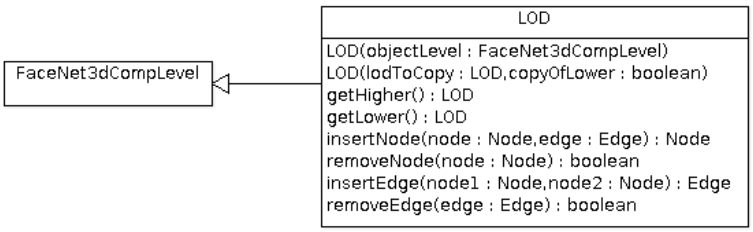


Fig. 89: Additional methods and constructors of LOD class

To freely navigate between levels of detail, the `LOD` class adds two methods `getHigher` and `getLower` that each return an object of `LOD` class that represents the LoD one step higher (i.e. with more details) or one step lower (i.e. with less details). Each `LOD` instance also provides a `getFaceNetComponent` method that allows to navigate "back" to the superordinate *h*-face net component, which in turn facilitates direct steps to the net level (`getNetLevelComp` method of `FaceNet3dComp` class) and to the object level (`getObjectLevelComp` method of `FaceNet3dComp` class).

The `LOD` class provides two constructors for instantiation (cf. Fig. 89). The first constructor consumes a `FaceNet3dCompLevel` object as its `objectLevel` parameter. This constructor just transforms an existing `FaceNet3dCompLevel` object into an object of `LOD` type without making any further computations (especially there is no copy process involved at this stage). Such an LoD object is yet not integrated into the hierarchy structure

of any *h*-face net component. Thus, to complete the building of a fully functional `LOD` object, the object has to be integrated into an *h*-face net component by passing the new `LOD` object as a parameter to the `addLoD` method of an object of `FaceNet3dCompLevel` class. This process is outlined in Listing 27, where `objectLevel` is the object level of a face net component and `hFaceNetComp` is an object of type `HFaceNet3dComp`.

```
FaceNet3dCompLevel objectLevel;
...
LOD lodToCopy = new LOD(objectLevel);
hFaceNetComp.addLod(lodToCopy, true);
```

Listing 27: Integration of a new LOD into an h-face net component by passing the object as parameter value (Java code)

In Listing 27, the object level of an existing face net component is used as the master to create a copy LoD which then is inserted into an *h*-face net component as the *LoD 1* – i.e. the LoD is defined more detailed than the object level but less detailed than the net level (analogue to the left depiction in Fig. 88).

A remarkable advantage of the outlined LoD management approach is that it is not invasive to the non-LoD core of the Topology Module. For example, the `CellTuple` class of core Topology Module had not to be manipulated in order to operate in LoD management. The `lower` and `higher` links of `CellTuples` that are used to switch between net level and object level, are also used to switch between two LoDs in LoD management. All `higher` and `lower` links are set between `CellTuples` of different LoD by the same principles as described in Ch. 3.1.8.

3.6 Implementation of a Geo-DBA For Time-Varying Topologies

The historical development of the currently utilized spatio-temporal models of DB4GeO for the management of temporal change in geometry and the theoretical foundations of these and related models have been outlined earlier in Ch. 2.5. The required capabilities of a temporal topology module implementation for DB4GeO are discussed on the basis of case studies in the following. Then a proposal for an architectural frame for a Temporal Topology Model is developed. The model is based on the *Temporal Joint Model* in DB4GeO that has been implemented by KUPER (cf. Ch. 2.5.5). Finally, it is clarified, which adjustments have to be performed on the *Piesberg data* (Ch. 1.6) in order to be able to process the data in further experiments of the Temporal Topology Module.

3.6.1 Required Capabilities of a Temporal Topology Module

The DB4GeO Temporal Joint Model is capable of managing temporal change of geometry and also supports the functionality to switch the underlying meshing of the geo-object at

certain time steps. Fig. 90 demonstrates the functionality of the Temporal Joint Model through an example set-up.

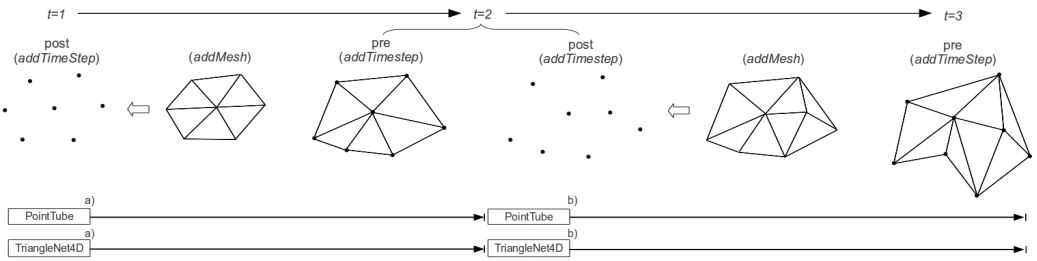


Fig. 90: Example of creation of temporal objects with DB4GeO Temporal Joint Model

In the example, first, at time step *t=1*, a set of points SP_a (`Point3D` objects) is added to the temporal object (left side of Fig. 90) through the `addTimeStep` method of `Object4D` class. This operation also prepares point tube PT_a and adds the set of points as the first entry of PT_a. Additionally, a meshing for the existing set of points is appended through the `addMesh` method of `Object4D` class. At this step, `TriangleNet4D` TN_a also starts to exist retroactive at time step *t=1*. At time step *t=2*, a new set of points SP_b is added through `addTimeStep` method of `Object4D` class. The points of SP_b geometrically differ from SP_a, nonetheless the topological configuration of TN_a stays constant. Subsequent, another set of points (SP_c) is also added at time step *t=2*. SP_c partially differs from SP_b. SP_c has more points than SP_b. It is not possible any more to map all points of SP_b to the points of SP_c. Thus, the already utilized point tube cannot be used for the new points. Subsequent, a new point tube (PT_b) is created.

Also, a new mesh is added for the new set of points (`addMesh` method). This mesh constitutes the net topology of TN_b. The meshing of TN_b also partially differs from the meshing of TN_a. The new set of points, together with the new mesh, constitutes the post object of *t=2*. At time step *t=3*, again a new set of points is added through `addTimeStep` method. This set of points constitutes the pre-object of *t=3* and again only differs in its geometry from the set of points of the post-object of *t=2*. The pre-object of *t=3* has the same number of points as the post-object of *t=2*. The new points are added to PT_b and also adopt the net topology of TN_b.

The example demonstrates, how it is possible to model a spatio-temporal object that changes in geometry and in its topological net configuration through time with the built-in spatio-temporal module of DB4GeO. As mentioned before, this spatio-temporal component of DB4GeO uses the Simplicial Complex approach to model the geo-objects geometry. It is not capable of modelling complex cells. Thus, the objective is to combine the existing functionality of the built-in spatio-temporal module (Temporal Joint Model) with the already implemented Topology Model. The intended functionality is to support the

modelling of complex cells over time (spatio-temporal complex cells) as depicted in Fig. 91.

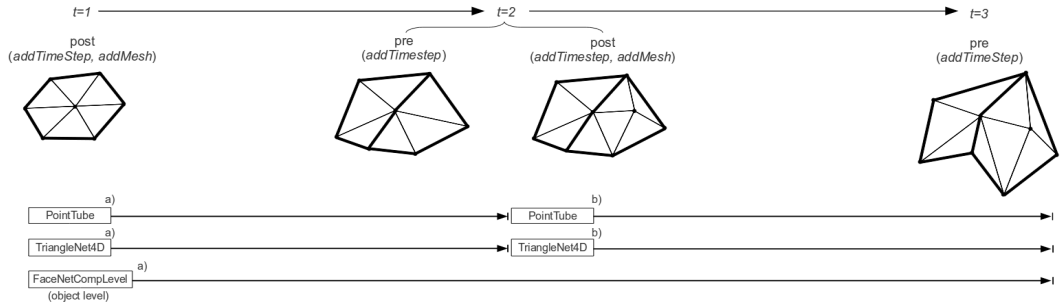


Fig. 91: Example of creation of temporal "big cells" on object level with Topology Module

Fig. 91 shows a similar temporal object creation case as in the previous example. But this example is extended with complex cells (i.e. non-simplicial cells) that are outlined by thicker lines (in the depiction of the net meshes). At *t=1*, there is not only a `TriangleNet4D` created but also a `FaceNet3D` and a `FaceNet3DCompLevel` at object level (C_{OL} , cf. lower left side of Fig. 91). At *t=1*, C_{OL} consists of only one face that extends over the whole net component (cf. left side of Fig. 91). At *t=2* not only the geometry has changed as in the previous example but also the topology of C_{OL} . The one face of C_{OL} has been split into two new faces. The topological change occurred somewhere in between *t=1* and *t=2* and is discussed below. In the transition from pre-object of *t=2* to post-object of *t=2*, the topology of the geo-object at net level changes, but the topology of the object level stays constant. The change of the net level is also apparent in the diagram of DB4GeO classes (bottom of Fig. 91). It shows that the temporal triangle net (`TriangleNet4D` class) and `PointTube` (both represent the net level) end to exist at time step *t=2*. They are both replaced by new objects of same type. However, the `FaceNetCompLevel` (object level of faces) does not end live at *t=2* but persists over the entire period. This is only possible if there is a direct mapping between all cells of C_{OL} of the pre-object and all cells of C_{OL} of the post-object.

At *t=3*, the geometry of the geo-object changes, as in the previous example, but the topology of C_{OL} stays still constant. This is possible, since there is a direct mapping of all nodes of the geo-object's mesh between the post-object of *t=2* and the pre-object of *t=3* (the net topology does not change between these two objects).

Fig. 92 provides a closer look at the time interval between *t=1* and *t=2*, where a change of the topology at object level occurs. At *t=1.5*, an edge is inserted into the (object level) face of C_{OL} by the use of the `insertEdge` method, thus splitting the face *a)* into two new faces *b)* and *c)*.

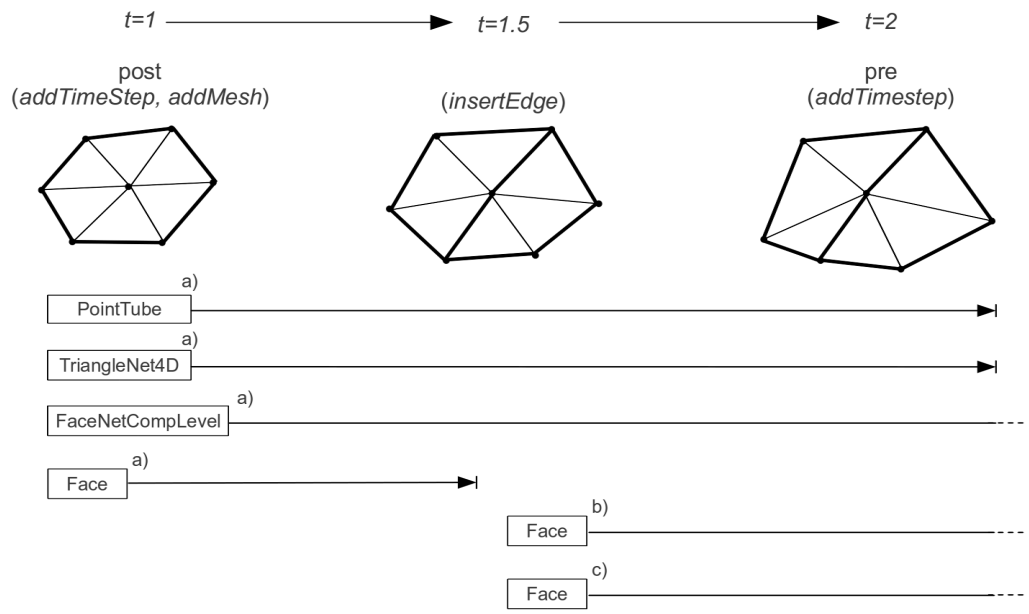


Fig. 92: Detail of Fig. 91: Interval between t=1 and t=2. An edge is inserted at t=1.5

Editing operations of object levels (such as the performed `insertEdge` method) can be conducted at any point in time (i.e. in time steps and in time intervals). Internally, the face net component at a certain (object) level is modified. In this case, face *a)* is split into two new faces *b)* and *c)*. All constraint checks and changes to the cells and cell-tuple structure of an insert edge operation that have been discussed in Ch. 3.4.2 are performed. Since all links between net level and object level cell-tuples are known, they can be adapted to the new structure accordingly. When a new net level topology (meshing) is introduced through post object of time step *t=2*, the links have to be recomputed according to the new spatial relation between object level and net level.

3.6.2 Architecture and Model of Temporal Topology Module

The development of the spatio-temporal model for the Topology Module is conducted in strict compliance with object oriented principles, especially inheritance semantics. They are used to maximize source code re-usage and modularity.

The main connector between the Temporal Joint Model and the Temporal Topology Model is the inheritance relationship between the `ServicesFor4DObjects` class of Temporal Joint Model and the `CellNetServicesFor4DObjects` class of the Temporal Topology Model (see Fig. 93).

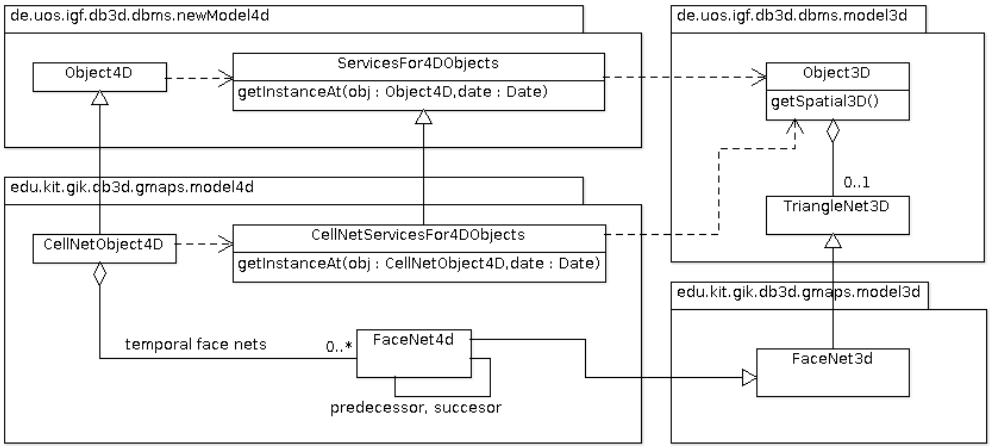


Fig. 93: The extraction of a 3D face net from a temporal cell net object as class diagram

`ServicesFor4DObjects` is a *service class*[148] that is responsible for the processing of temporal geo-objects. It currently basically provides one method that is called `getInstanceAt`. The `getInstanceAt` method is a "processing unit" that consumes a temporal object (`Object4D`) and a `Date` object and returns an object of type `Object3D` as a result of the processing. This means that the service class creates a snapshot of a temporal geo-object, i.e. "translates" that temporal geo-object into a 3D geo-object at a certain time step (`Date`).

The `CellNetServicesFor4DObjects` enters at this point and specializes the `ServicesFor4DObjects` class by overriding the `getInstanceAt` method. The specialized `getInstanceAt` method consumes a *temporal cell net object* (object of type `CellNetObject4D`) instead of an `Object4D` but also provides an `Object3D` as the process' result. A temporal cell net object is a specialisation of a temporal object. Whereas a temporal object only provides functionality for the management of geo-objects whose spatial part consists of a temporal Simplicial Complex, a temporal cell net object additionally models a temporal cell net as the spatial part of the geo-object.

A temporal cell net object in first place consists of a set of discrete *temporal cell nets*. In the current implementation, temporal *face* nets (`FaceNet4d`, see Fig. 93) are the only realisation of temporal cell nets.[149] Internally, the set of temporal face nets is managed in a (sequential) list (`LinkedList`) that is part of the field of the temporal cell net object (see Fig. 94).

148 The distinctive feature between a utility/helper class and a service class is that while the former is intended to solve typical internal issues, the latter is mainly intended to provide an access interface for (external) clients/API users.

149 Thus the model is discussed only on the basis of temporal face nets. Temporal solid nets are still to be developed and to be implemented.

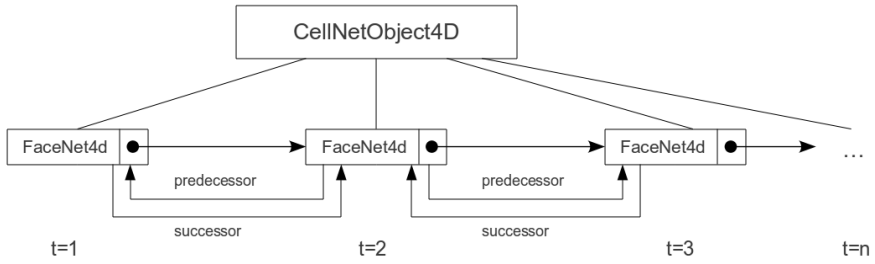


Fig. 94: A temporal cell net object includes a sequential list of temporal face nets

The fact that a temporal cell net object consists of temporal face nets is an analogy to the Geometry Model of DB4GeO. But whereas a geo-object (`Object3D`) has one triangle net as its spatial part, a temporal cell net object has several face nets as its spatial part. All temporal face nets of a temporal cell net object in fact represent one (the same) face net that changes in time and has a representation at several time steps (its manifestations). Every entry in the field list is a temporal manifestation (cf. Fig. 94).

The `getInstanceAt` method of `CellNetServicesFor4DObjects` takes a *temporal cell net object*. Then the method first creates a new 3D geo-object, determines the temporal face net that is valid at the given `Date`, attaches this face net at the newly created geo-object as its spatial part and returns the geo-object. Fig. 95 shows this flow of application states exemplary on a `CellNetObject4D` that already has three face nets at three different time steps.

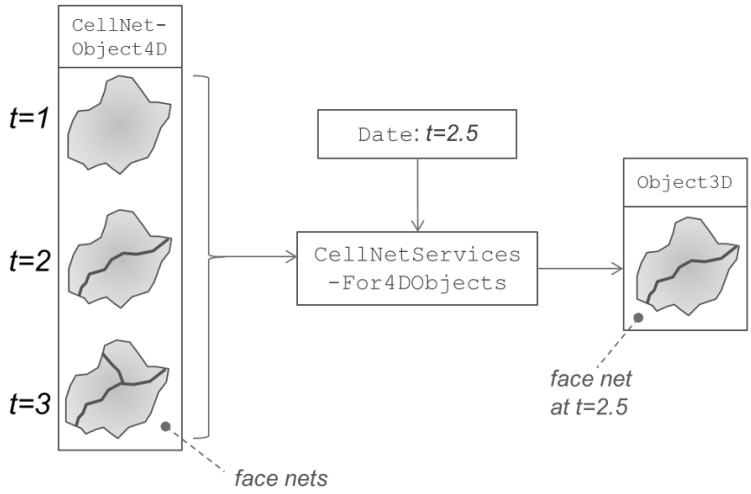


Fig. 95: Service class use case, employing temporal cell net object

In this example, the service class consumes the temporal cell net object with the three temporal face nets and a `Date` object with *t=2.5* and provides an `Object3D` as result. The `Object3D` holds exactly one temporal face net (`FaceNet4d`) that is valid at *t=2.5* as its spatial part, which is the temporal face net at *t=2*. This is possible, since a `FaceNet4D` is a

`FaceNet3d` which in turn is a `TriangleNet3D` (cf. Fig. 93); thus a `FaceNet4D` can "automatically" be the spatial part of a geo-object (by the rules of polymorphism). `Object3D` provides the temporal face net through its `getSpatial3D` method (see Listing 28).

```
CellNetServicesFor4DObjects services =
     new CellNetServicesFor4DObjects();
Object3D geoObj =
     services.getInstanceAt(obj4d, date2013);
FaceNet4d tempFaceNet = (FaceNet4d) obj3d.getSpatial3D();
```

Listing 28: Retrieving a temporal face net as the spatial part of a geo-object

A temporal face net is simply a 3D face net (`FaceNet3d` class), but additionally has an extended class field, with the state references `predecessor` and `successor` to the respective *temporally preceding* and *succeeding* 4D face nets. These references may also be `null` if the respective temporal face net has no predecessor or no successor. So temporal face nets may always be accessed from their parental temporal cell net object or from another temporal face net through the `state` references.

Since a temporal face net (`FaceNet4d`) is a specialisation of face net (`FaceNet3d`), it not only can be used in any context where a `FaceNet3d` can be used[150], but it also inherits its set of an arbitrary amount of face net components (see Fig. 96).

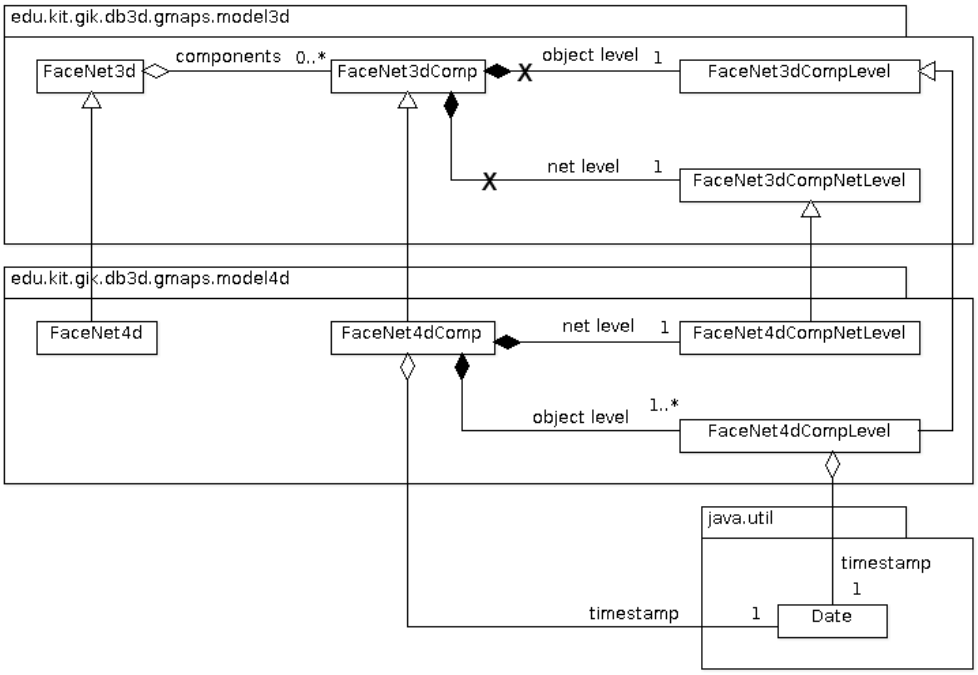


Fig. 96: The temporal net components (with their NL and OL) of temporal face nets in a class diagram

150 For example as the special part of a basic 3d geo-object

Originally, a 3D face net aggregates 3D face net components. Since a *temporal face net component* (`FaceNet4dComp`) *is* a face net component (`FaceNet3dComp`), a face net is also capable of aggregating an arbitrary amount of temporal face net components. As a result, it can be stated that every temporal face net (`FaceNet4d`) is an aggregation of temporal face net components (`FaceNet4dComp`).

Originally, a face net component is a composition of exactly one face net component at object level (one `FaceNet3dCompLevel`) and exactly one face net component at net level (one `FaceNet3dCompNetLevel`) which are the net level and object level representations of the same face net component (cf. Fig. 49 and top of Fig. 96). These references to object level and net level components cannot be used here any more as a consequence of the differing definitions of the static net component in comparison with the temporal net component: unlike the static net component, the temporal net component may consist of multiple net level and multiple object level components (cf. Fig. 96) that describe the temporal progression of the net component at net and at object level. This principle is also depicted in Fig. 97.

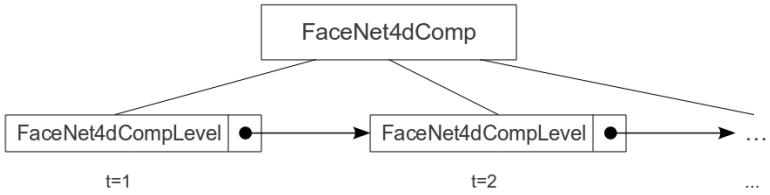


Fig. 97: A temporal face net component includes sequential lists of face net components at net level and at object level

A temporal face net (`FaceNet4dComp`) provides a (sequential) list (`LinkedList`) of net level components (`FaceNet4dCompNetLevel`) and object level components (`FaceNet4dCompLevel`) of which everyone exists at a certain point in time (these are the manifestations of the object level component and the net level component). Thus, the original references of `FaceNet4dComp` to exactly one C_{OL} and exactly one C_{NL} are not sufficient for the modelling of a temporal net component and must be replaced by the above-mentioned list. As a result, a `FaceNet4dComp` is a composition of any number of (but at least of one) temporal face net components at net level (`FaceNet4dCompNetLevel`) and of any number of (but at least of one) face net components at object level (`FaceNet4dCompLevel`). The face net components at object level are the multiple temporal object levels of a face net component and the face net components at net level are the multiple temporal net levels of the same face net component.

Both, the face net components (`FaceNet4dComp`) as well as all temporal object levels (`FaceNet4dCompLevel`) of every face net component have each of them one `Date` object that provides the *timestep* at which the object began to exist.

As depicted in Fig. 91, an objective of the temporal module is to establish a cell based temporal link between the cells of the pre- and the post-cells of a time step. In the current implementation of this doctoral thesis, this is realised for node cells. For this purpose, the already existing basic `Node` class (of `model3d` package) has been extended with the two reflexive references `successor` and `predecessor` that point to the nodes that are temporally succeeding and preceding the respective node (see Fig. 98).

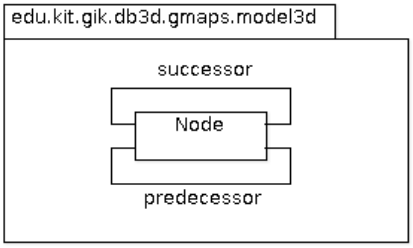


Fig. 98: Basic Node class (of model3d package) extended with temporal field

These references between preceding and succeeding nodes are compiled every time when a new net meshing is added to the temporal geo-model (through the `addGeometry` method of `CellNetObject4D`). In this case, the `addComponent` method of `FaceNet4dBuilder` is invoked internally by the `addGeometry` method, and all components are added to the face net. If actually a face net has a preceding face net, then the object level components of both face nets are extracted at the respective `Date`[151] and equal nodes are identified on both nets (by their equal ID) and are assigned to each other.

3.6.3 Preparation of Piesberg Dataset

The whole process can be exemplified on the Piesberg dataset (that is introduced in Ch. 1.6). The following examples show actual outputs of the Topology Module modelling that is visualized with *ParaviewGeo* and manually labelled. To be used with temporal G-Maps, the Piesberg dataset needs to be prepared. The dataset has exactly one version of the meshing for every of the 12 time steps, but no pre- and post-objects. Thus, the pre- and post-objects have to be created manually.

For a test run, two time steps are chosen from the dataset. To arrange a more catchy example, the two chosen time steps are not two sequential time steps of the dataset but are temporally more separated. These are the time steps of the years 1976 and 1983 (though the sequentially following time step in the dataset to 1976 is 1978). To produce the required pre- and post-objects, the following individual operations are carried out.

151 Pre component and post component have the same date since they are modelled at the same time step.

First, the geo-object of 1976 is chosen to be the pre-object of 1976. Second, the coordinates of the points of the geo-object of 1983 are parallel projected[152] onto the surface of the pre-object of 1976. This results in the post-object of 1976. Thus, the post-object of 1976 has the meshing of the pre-object of 1983 but the surface geometry of the pre-object of 1976. Next, the geo-object of 1983 is determined to be the pre-object of 1983. Then the coordinates of the points of the geo-object of 1993 are parallel projected onto the surface of the pre-object of 1983 to produce the post-object of 1983. This four prepared datasets are imported and managed by topological 4D module as presented in Fig. 99.

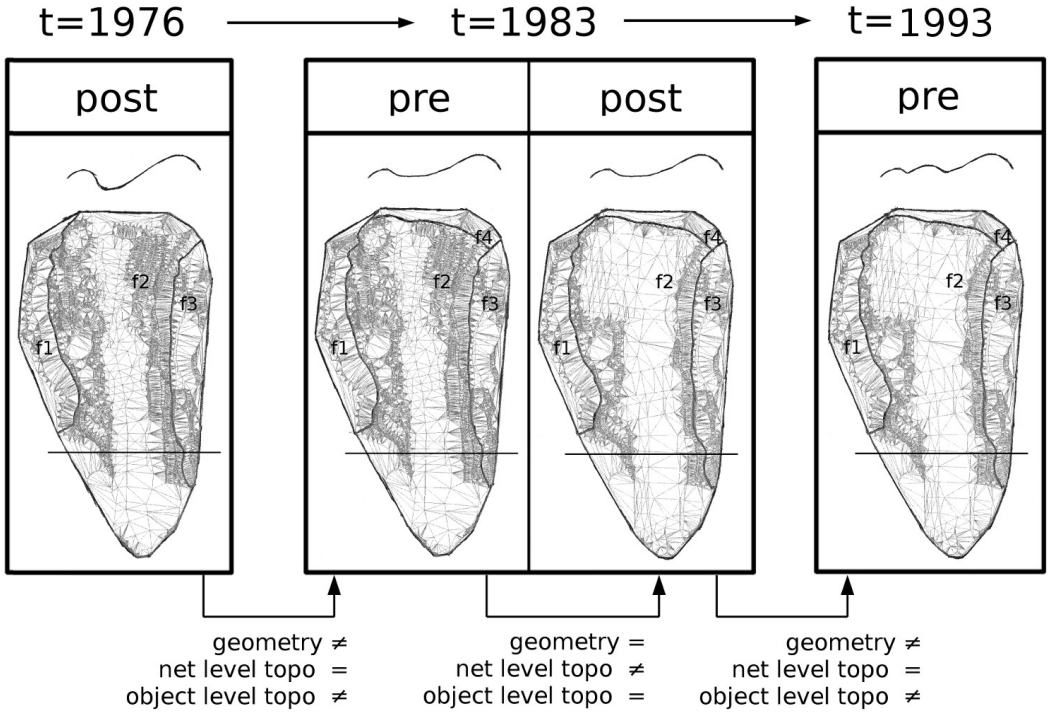


Fig. 99: Top views of the Piesberg dataset pre- and post-objects of the years 1976, 1983 and 1993

Fig. 99 shows the top views of the Piesberg dataset's pre- and post-objects of the time steps 1976 and 1983. Time only elapses between the post-object of 1976 and the pre-object of 1983. This is the time frame where the geometry changes, while the topological configuration of the triangle mesh stays constant. Thereby, it is possible to generate “artificial” geo-objects of time steps where the geo-objects were not modelled explicitly, such as in the Piesberg dataset e.g. in 1979 (cf. Fig. 15). The change of the geometry of the geo-object surface is indicated in the illustration by the curly lines directly above the top views of the geo-objects. These curly lines show the cross-section of the surface at the continuous horizontal cross lines that are drawn onto the top views. It is evident from the cross-sections that the surface geometry does not change between the pre- and the post-

152 By the means of the geometric operations of DB4GeO (composite operation PolyLineDrillingOperation)

object of one time step, but it changes between the post-object of a preceding time step and the pre-object of a succeeding time step.

The changes between the depicted geo-object states of Fig. 99 are categorized into three types of changes. These are changes in the surface geometry ("geometry" in Fig. 99), changes in the topology of the net level (or the meshing, "net level topo") and changes in the topology of the object level ("object level topo"). In Fig. 99, there are links between the depicted geo-object states that are labelled with the types of change that occur in the respective transition. In the transition from pre-object of 1976 to post-object of 1976, the net level topology changes while the geometry stays constant. "Constant geometry" means in this context the constance of the "overall geometry" of the geo-object, i.e. in this case the geometry of the whole surface. While the surface geometry stays constant, the point objects of the surface are replaced. Though, the points are exchanged, one important constraint of this replacement is that all "post-points" remain on the surface of the pre-object. The net level topology (i.e. the mesh configuration of the triangle net) also changes. The procedures on object level are discussed later. In the transition from post-object of 1976 to pre-object of 1983, the meshing of the triangle net stays constant (also the amounts of points, edges and triangles stay constant) while the geometry of the surface geometry, i.e. the coordinates of the points, changes. The transition from pre-object of 1983 to post-object of 1983 is analogous in geometry and on net level.

At object level, there are at first three faces (*f1*, *f2* and *f3*) in pre- and post-object of 1976 and later four faces (*+f4*) in pre- and post-objects of 1983. Not depicted in Fig. 99 is that a new face has been added at object level in 1980 (see Fig. 100).

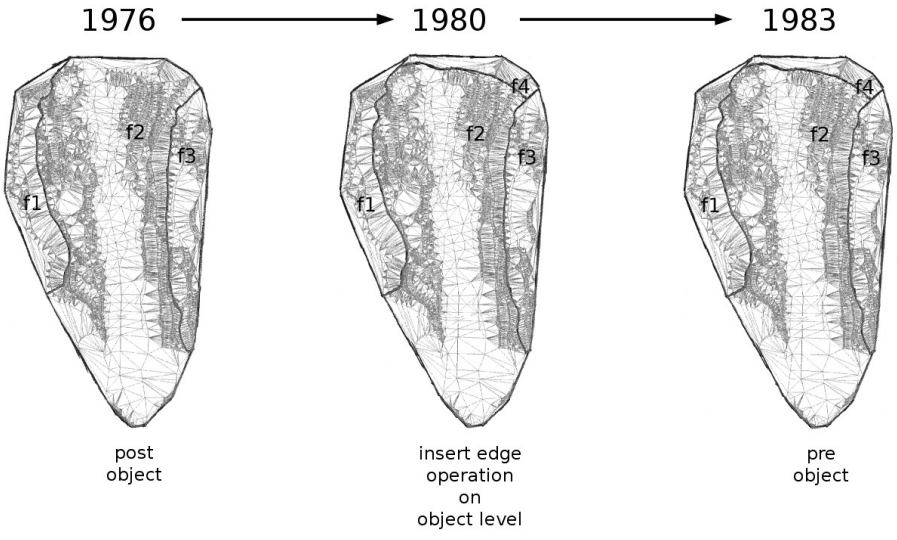


Fig. 100: Adding a new face at object level (in 1980) through insertEdge method

Particular attention should be paid to the fact that the faces (at object level) are preserved through time, though in the meantime the geometry and the meshing of the geo-object changes.

This example demonstrates the broadness of the chosen modelling approach. The model not only allows for the temporal change of the geo-object's geometry but also permits the temporal change of the geo-object's meshing as well as supports the management and editing of temporal “big cells” (on object level) that even preserve when the meshing changes.

4 Performance Measurements and Comparisons

This chapter presents metrics on *CPU runtime* and *memory consumption* of the Topology Module.[153] Where appropriate, the measured data is compared to performance data of the DB4GeO Kernel. The purpose is to gain a better judgement of the advantages and disadvantages in execution performance that result from the usage of the module, compared to the usage of the Kernel solely.

All measurements are compiled on a common desktop machine with ~ 280 MFLOP/s[154]. The test data are synthetically generated by an algorithm (see Ch. 4.1). Therefore, it is possible to easily adjust the data size for different measurement iterations. The measuring of CPU runtime and memory consumption can be subject to random influences, due to the characteristics of modern operating systems (OS). Since modern OS are capable of multitasking, it is not certain, when processes are switched. Obviously, a process switch has a negative impact on runtime. In order to reduce the effects of such unpredictable external factors, all runtime values are calculated as an average of at least *ten* runtime measurement cycles. Similar considerations apply to the measuring of memory consumption.

153 The source code of all performance tests and the result data are documented in a separate Eclipse development project named “Db3dProfiling”.

154 The calculation of “floating point operations per second” is considered more accurate than the provision of the more common “instructions per second”, since FLOP/s incorporates multiple properties of a computer's architecture like main memory, bus system etc. However, the presented value is determined with the Java Linpack Benchmark of the Oak Ridge National Laboratory (for further information visit http://www.netlib.org/benchmark/linpackjava/)

4.1 Construction of Net Components

In the first test set-up, the creation of geometry and topology structures is measured. For this purpose, an array of triangles is prepared as synthetic geometry data. All triangles together form a flat, contiguous net of quadratic shape. The array of triangles is passed to the constructors of Simplicial Complex (triangle nets with DB4GeO's `TriangleNetBuilder`) and G-Maps face nets (`FaceNetBuilder` of Topology Module). The duration of the construction process is measured in milliseconds (ms). The averaged results are presented in the left diagram of Fig. 101.

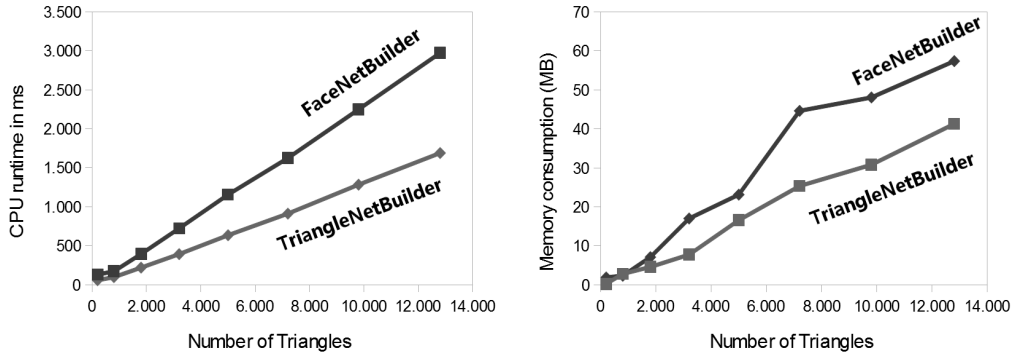


Fig. 101: Runtimes of net component construction (left); memory consumption of net component construction (right)

As can be seen in the left diagram of Fig. 101, with a rising number of triangles, the CPU runtime of net component construction also rises proportionally. With a number of triangles slightly below *2.000*, the runtime amounts to about a *quarter of a second* for `TriangleNetBuilder` and to about *half a second* for triangle *and* face net construction (`FaceNetBuilder`). At the other end of the scale, it takes *1.5 s* for `TriangleNetBuilder` and *3 s* for `FaceNetBuilder` to process about *13.000* triangles.

The right diagram of Fig. 101 presents the memory consumption of the process of triangle net and face net creation in megabyte (MB). With a number of triangles slightly below *2.000*, the memory consumption adds up to *5 MB* for triangle net construction and *7 MB* for face net construction. Whereas a model with about *13.000* triangles needs *40 MB* to construct only a triangle net and *60 MB* to construct a face net.

It can be concluded that the construction of a triangle net is generally faster and consumes less memory than the construction of a face net. This result is obvious since the construction of a face net includes the construction a triangle net. Furthermore, the face net builder needs time and memory to generate all the additional topological links and objects of a face net. In practice, this means that the user has to wait *1.5 s* longer and provide *20 MB* more main memory or permanent storage in average if he has a dataset of *13.000* triangles and he wants to construct a G-Maps face net instead of a DB4GeO triangle net.

In the process of net component construction, each net element (i.e. triangle element or tetrahedron element) is traversed mostly only once. Thus, the asymptotic runtime is a linear function of n ($O(n)$ in BACHMANN-LANDAU notation), with n being the number of elements in the net component. The graph of Fig. 101 confirms these assumptions, since it shows a clear linear growth in runtime behaviour as well as in memory consumption for DB4GeO Kernel and Topology Module.

4.2 Basic Spatial/Topological Queries

In the second test set-up, the retrieval of topological information is measured. In this measurement, larger arrays of triangles are synthetically prepared in order to get significant test results. Apart from that, these are again flat, contiguous nets of quadratic shape. After triangle nets and face nets are constructed, the duration of selected retrieval operations are measured in *ms*. The diagrams of Fig. 102 show averaged results of *boundary retrieving* operation runtimes (left diagram) and *get-2D-for-0D* operation runtimes (right diagram).

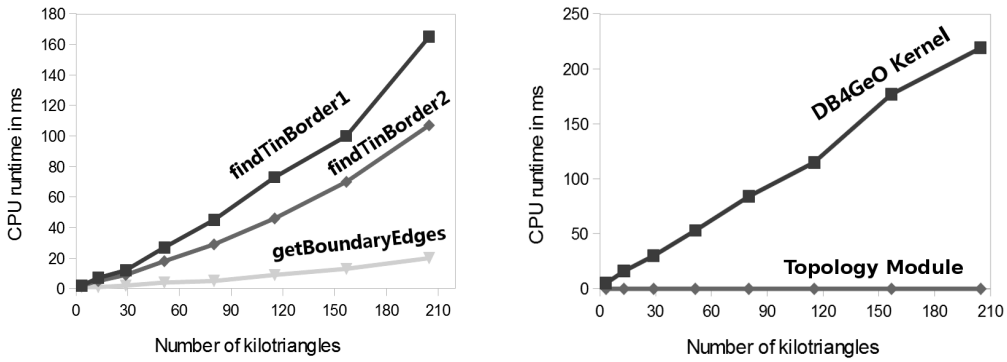


Fig. 102: Runtimes of boundary retrieving operations (left); runtimes of get-2D-for-0D operation (right)

The boundary retrieving operations have been implemented in order to receive the boundary geometry of a complex geo-object, for example the linestring that bounds a surface. These operations were programmed by the means of DB4GeO Kernel (`findTinBorder` methods) and by the means of Topology Module (`getBoundaryEdges` method) independently.

In DB4GeO Kernel, the boundary retrieving operations have been realized in two ways. The first algorithm (`findTinBorder1` method of `TriangleNet3DComp` class) simply iterates over all triangles of a surface and checks each triangle whether it is at the surface's boundary. If so, the boundary segments are added to the result set. This algorithm returns an *unordered* set of boundary segments.

The second algorithm (`findTinBorder2` method of `TriangleNet3DComp` class) first finds an arbitrary boundary triangle, then it follows the triangles along the boundary and

collects the boundary segments. This algorithm returns an *ordered* set of boundary segments.

In Topology Module, the `getBoundaryEdges` method of `FaceNet3dCompNetLevel` class returns all boundary edges of a face. The algorithm takes advantage of *2-orbits*, therefore it simply follows explicitly modelled links along the boundary edges. This algorithm returns an *ordered* set of boundary segments.

The different boundary retrieving operations have different asymptotic runtime behaviour. In worst case, the `findTinBorder1` method has to evaluate *all* triangles (*n*) of the net component, each once, in order to find all boundary segments. Additionally, it has to add all boundary segments (*m*) to the result set. Thus, it has a linear asymptotic runtime $O(n+m)$. This assumption is confirmed by the graph of `findTinBorder1` method in left diagram of Fig. 102 that shows an at least linear growth rate.

The `findTinBorder2` method first needs to find a boundary segment of an arbitrary triangle at the component's boundary in *k* steps. Then it iterates only along the boundary segments. The iteration along the boundary still needs some computation, since there are no precomputed links along boundary triangles or segments. If the number of boundary segments is *m*, then the asymptotic runtime is $O(m+k)$. The runtime of `findTinBorder2` method is assumed to be usually less than the runtime of `findTinBorder1` method, since the number of boundary segments *m* is usually less then the total number of triangles *n* of a component. The supposed asymptotic runtime is backed by the graph of `findTinBorder2` method in left diagram of Fig. 102 that also shows a generally linear growth rate. However, as also assumed, the runtime is lower than the runtime of `findTinBorder1`.

The `getBoundaryEdges` method simply navigates along the explicitly pre-computed 2-orbit links along the boundary segments. By this, the asymptotic runtime only depends on the number of boundary edges *m* that have to be collected. Thus, its asymptotic runtime is $O(m)$, which is the optimum runtime, since each boundary element needs to be visited at least once in order to become part of the result set. Furthermore the retrieval of neighbouring edges is fast, since boundary edge neighbourhood is precomputed and boundary edges are linked. The graph of `getBoundaryEdges` method in left diagram of Fig. 102 reassures this assumption, since the average runtime of Topology Module method is much lower than the average runtimes of DB4GeO Kernel methods. Such evidence gives reason to call the data structure of Topology Module a “topological index”.

Working with a geo-object with the size of about 13.000 triangles, the average runtime gain of Topology Module against DB4GeO Kernel is *4 ms* at each invocation of a boundary retrieving method. The average runtime loss of Topology Module against DB4GeO Kernel during the construction process is *1.3 s* (cf. Fig. 101). This means that, with a geo-object of 13.000 triangles, the usage of Topology Module starts to pay off in

terms of runtime consumption after approximately 300 invocations of `getBoundaryEdges` method.[155]

The right diagram of Fig. 102 shows averaged runtimes of *get-2D-for-0D* operations. These operations return the 2D geometries (surfaces) that are incident to a 0D geometry (point). *Get-2D-for-0D* operations are available in both, DB4GeO Kernel and Topology Module. The DB4GeO Kernel returns all triangles that are incident to a given point. In order to compute the result set, the algorithm iterates over all triangles of the triangle net component and compares all points of each triangle until an equal point is found. This has the effect that all triangles (*n*) have to be visited in order to get the result. Accordingly, the asymptotic runtime is linear ($O(n)$), which corresponds to the graph of *get-2D-for-0D* operation of DB4GeO Kernel in right diagram of Fig. 102.

The *get-2D-for-0D* operation of Topology Module returns all faces that are incident to a node. In contrast to the *get-2D-for-0D* operation of DB4GeO Kernel, the operation in Topology Module simply follows the link to a cell-tuple of the node and then runs a 0-orbit "around" the node in order to collect all incident faces. This operation has to perform only one step for each incident face. The runtime only depends on the number of faces in the result set (*m*), which is usually very small compared to *n*. Thus, the asymptotic runtime is $O(m)$, which is the optimum runtime, since each incident face needs to be visited at least once in order to become part of the result set. With a runtime of $O(m)$, the Topology Module decouples the calculation of the result set from the total amount of triangles. Having a geo-object of about 200.000 triangles, the Topology Module saves *200 ms* with each *2D-for-0D* operation compared to DB4GeO Kernel.

4.3 Additional Performance Tests

The presented ratios in runtime behaviour can be found in a multitude operations of DB4GeO Kernel and Topology Module, since many other methods base on the presented fundamental methods, and since the presented algorithmic approaches are used in a similar way in related operations. As an example, the average runtimes of *get-1D-for-0D* and *countBorderEdges* operations are depicted in Fig. 103.

155 *300 x 4 ms = 1.2 s*, which is nearly 1.3 s that is needed for construction

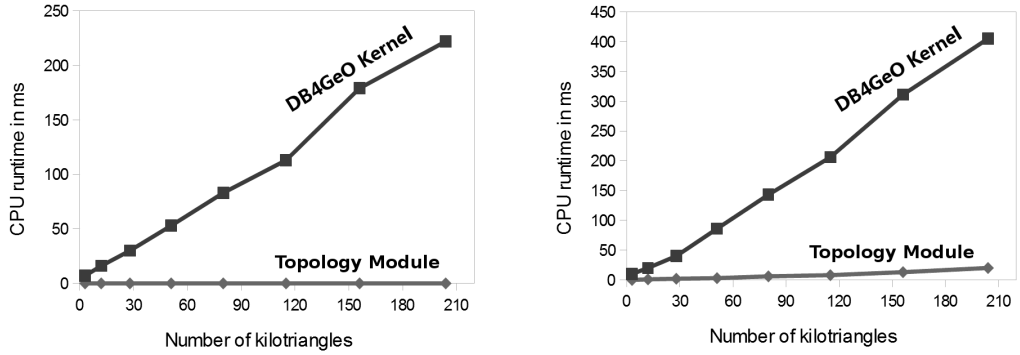


Fig. 103: Runtimes of get-1D-for-0D operations (left); runtimes of countBorderEdges operations (right)

The left diagram of Fig. 103 shows the average runtimes for operations that retrieve the *segments* (*edges*) that are *incident* to a given point (node). These operations are similar to the *get-2D-for-0D* operations presented in right diagram of Fig. 102. Actually, the DB4GeO Kernel internally uses the same code base as *get-2D-for-0D* operation with the difference that it additionally needs to check, to which segments of a triangle a given point belongs. The *get-1D-for-0D* operation of the Topology Module is also similar to the *get-2D-for-0D* operation of Topology Module. Internally, they also both use the same code base that executes a *0-orbit* around the given node. Since the *get-1D-for-0D* operations are based on the same principles as the *get-2D-for-0D* operations, the resulting runtime behaviour is also similar, which can be recognized by comparing left diagram of Fig. 103 to right diagram of Fig. 102.

The right diagram of Fig. 103 shows the average runtimes for operations that count the *number of boundary segments/edges*. These operations are strictly based on the *boundary retrieval* operations that have been introduced earlier in this section, since internally they do nothing more then retrieving the boundary segments and return the size of the result set. Thus, the resulting runtime behaviours are similar, which can be recognized by comparing right diagram of Fig. 103 to left diagram of Fig. 102.

This chapter shows the advantages and disadvantages in terms of runtime behaviour and memory usage of Topology Module compared to DB4GeO Kernel. While the memory usage of Topology Module is approx. *40 %* higher (due to the additional extensive topological structure) and the construction process takes approx. *80 %* more time than the DB4GeO Kernel, the Topology Module then saves runtime, e.g. as shown in figures 102 and 103, each time the user states a spatial/topological query. As the module is used in a Geo-DBA, this runtime behaviour becomes especially useful, since a database for geo-applications is imported and stored in a DBMS once, but queried, retrieved, and exported several times (eventually by multiple users).

5 Discussion

5.1 Summary and Conclusion

The doctoral thesis at hand deals with the examination of methods for topological data handling, as well as designing and implementing a toolkit for the geosciences for the management of topological and temporal 3D data and models in Geo-DBA. The conceptual work and realisation process takes into account the management of multi LoD since it is considered to be an integral part of geo-modelling.

The first main chapter (introductory chapter) unfolds the motivation for handling geoscientific data and employing database management systems for geodata. The current state of processes, modelling tools, and DBMS is presented in application-oriented use cases. Furthermore, the introduction gives a first insight into the topics of model integration, abstraction of geodata and spatio-temporal modelling. These first insights already include references to early pioneering and basic literature. Finally, the test dataset of this work is presented.

In the chapter it is shown that data and model management for the geosciences is a non-trivial, multifaceted task. Geoscientific data is characterized by a heterogeneity of models and applications. The integration of multiple models and applications into common systems is a striving goal of the international scientific community. It allows for new types of analyses that have not been possible before. Geo-DBMS are a useful platform for data management and model integration, but market (and close-to-market) solutions are still in an early stage and leave many questions of geoscientific data management open.

As Rhind (1973), Buttenfield (1993) and others have shown, multi-representation of geodata is broad in topic so that the perspective has to be narrowed. However, a key object of study is the adequate maintenance of linking between the geo-objects of multiple resolutions. The question of how to generate and organize the hierarchy links greatly influences the navigation and editing capabilities of the multi-resolution database. Contemporary research efforts by Haunert and Sester (2005), and Anders and Bobrich (2004) in the management of multi LoD of DLM are introduced.

Another key topic of research in the geoinformation community is the management of spatio-temporal data. The introductory chapter lays out the importance and application-relevance of temporal geodata by presenting an elaborate use case in city planning. The research range is widened by an overview of key research topics that have been compiled by Lester (1990) which are the understanding of time, temporal logic, architecture of temporal GIS, and how to deal with alternative representations. The focus of his thesis is placed on the architecture of temporal GIS. In this context, Shoham and Goyal (1988) identify four different reasoning tasks that can be supported by temporal GIS, which are prediction, explanation, learning new rules, and planning.

According to Wachowicz (1999), there are two fundamental approaches on temporal GIS: layer-based or object-oriented. Regardless of the approach, Dadam et al. (1984) identify two types of strategies for incremental update, which are forward oriented and backward oriented versioning. According to Snodgras and Ahn (1985), time can be recorded in the two types valid time and transaction time. Depending on the level of assistance of types of time, Snodgras and Ahn distinguish four kinds of (chronological) databases, which are snapshot, rollback, historical and temporal databases. Hägerstrand (1975) established the notion of space-time trajectories of geo-objects that can be taken as a basis to conceptualise longitudinal and branching configuration of valid time. Additionally, space-time paths introduced states and events in the lifetime of geo-objects.

The second chapter gives a deeper insight into relevant literature, explains theoretical considerations in more detail, and presents the current state of research. The chapter starts with an introduction into the architecture of DB4GeO's geometry kernel and exposes its capabilities and deficiencies in navigation on the geometry. The concepts of cell-tuple structure by Brisson (1989) and G-Maps by Lienhardt (1989) are introduced as an alternative approach to model the topology of complex geo-objects. Inter alia due to their advanced navigational capabilities, these concepts are chosen as a basis for a new topological kernel architecture for DB4GeO that promises to achieve data access in constant time in many cases (“topological index”) and to simplify the handling of topological information, especially in complex environments such as multi-representation and spatio-temporal modelling. For testing purposes, the new prototype kernel is developed as a plug-in/module for DB4GeO and not directly incorporated into the existing DB4GeO Kernel.

The topic of multi-representation is further deepened by going into more detail of the foundational research work on multi-representation in 2D (hierarchical DLM) by Anders/Bobrich and Haunert/Sester. Another foundational abstraction technique that applies to the geometry of 3D geo-objects, the PM by Hoppe (1996), is explained in detail. The explanation of PM demonstrates what kind of geometric models are relevant in multi-representation applications. After having presented concrete, application-related approaches, the more theoretical, generalized approach MTR by Bruegger and Kuhn (1991) is introduced. Bruegger and Kuhn identify two types of relations between cells of different LoD and set the basic framework for further reflections on multi topological representations. The H-G-Maps model of Fradin et al. (2005) can be seen as a concrete

implementation of multiple topological representations. In H-G-Maps, each level of topological representation is modelled as a complete separate G-Map. Hierarchy links between the multiple G-Maps are established through links between the G-Maps' darts.

The topic of spatio-temporal modelling is continued with a contrasting juxtaposition of concepts of continuous and discrete temporality. It is identified that the geometry of a geo-object can change continuously in time while the topology changes in discrete steps. In order to model both, the geometry and topology of a geo-object in one joint system, it is necessary to develop an integrative model. Polthier and Rumpf (1995) present such an integrative model by introducing TimeStep, an adaptive time-dependent discretization. By incorporating time-steps with pre- and post-objects into the model, the approach allowes to handle continuous change of geometry and discrete change of topology in one model.

The DB4GeO Kernel has been extended by the temporal point tube model of Rolfs (2005) that is based on the ideas of TimeStep. The temporal point tube model combines two representatives of a simplex element at two different time steps to a space time element. A series of space time elements is combined to a space time sequence, and a set of spatially non-overlapping space time sequences forms a space time component. The temporal point tube model can be categorized as an accumulative forward oriented versioning system that manages valid time in a longitudinal configuration. Later, the temporal model of Rolfs has been revised by Kuper (2010). The new spatio-temporal model for DB4GeO supports non-accumulative forward oriented versioning. It merges the concepts of PointTube, delta storage, and Polthier/Rumpf in one new model in order to manage continuous change of geometry.

Though, the presented models do a great deal in modelling dynamic geometry, they do not cover the modelling of temporal topology. 2-dimensional temporal topology has been covered by Raza and Kainz (1999) in their concept of STAO. STAO introduces a concept of temporal cells and the notion of temporal cell-tuple structure. It provides the means to model cell complexes that change their topological configuration in discrete time instances.

The third chapter explicates the design and implementation of the Topology Module for the modelling of spatio-temporal topological objects. Since the module is attached to the DB4GeO Kernel, the chapter begins with an explicit description of the Simplicial Complex geometry model of the Kernel. Generalized cells are introduced as wrappers for the simple geo-object types of DB4GeO: nodes for points, edges for segments, faces for triangles, and solids for tetrahedra. All cells (i.e. one cell of each dimension) are then integrated into `CellTuple` class. Sets of `CellTuple` objects create a G-Map that represents the topology of a cell net component. A cell net component is a collection of conjunct, non-overlapping cells realized as an extension of simplex net components of DB4GeO. Multiple cell net components can loosely be coupled to a cell net realized as an extension of simplex nets of DB4GeO.

The employed cell-tuple model encompasses universe cells, cell-tuple polarity, and holes in cell net components. The differentiation into net level and object level allows to create a “big cell” that comprises an arbitrary number of cells. The net level is defined as the G-

Map that exactly represents the topology of the Simplicial Complex. The permissible cells at net level are identical to the simplices. At object level, instead, cells can have any geometric representation. Usually, one big cell of the object level is detailed by multiple cells of the net level. References between both detail levels are realised through `higher` and `lower` links between darts of the respective levels. The topology of net level is non-editable, since it is completely dependent upon the underlying Simplicial Complex of DB4GeO Kernel. However, the topology of object level can be altered within some defined restrictions.

The construction of cell complexes (and G-Maps) is based on the construction of Simplicial Complexes of DB4GeO Kernel. So the process of cell net creation has three stages. After the API user hands over a set of simplices, first, the DB4GeO Kernel analyses the geometric configuration of the simplices and aggregates them based on this information to a Simplicial Complex. Secondly, the Topology Module receives the pre-engineered Simplicial Complex from DB4GeO Kernel and evaluates the neighbourship configuration of the structure. From the evaluation, all information can be gathered that is needed to construct a cell complex and the corresponding G-Map at net level. In the third step, the cell complex at net level is taken as a basis to derive the cells and G-Map of object level.

Once a cell net and G-Map are created, the Topology Module provides the API user with a sophisticated methodology to freely traverse the topological structure in any direction. This is achieved in an accurate architectural approach by the means of `OrbitIterator` class, which realizes the `Iterator` interface of standard Java. The `OrbitIterator` helps the API user to create different kinds of orbits. The user only needs to provide the involution sequence or orbit dimension of interest. The algorithms of `OrbitIterator` already demonstrate the benefits of the G-Maps approach. They are comparably short, crisp, and elegant. `OrbitIterator` serves as a basis for the definition of `CellIterator` class. Cell iterators help the API user to iterate over the neighbours of cells. For example, with an appropriate cell iterator the user can easily query for all faces that are adjacent to a given face.

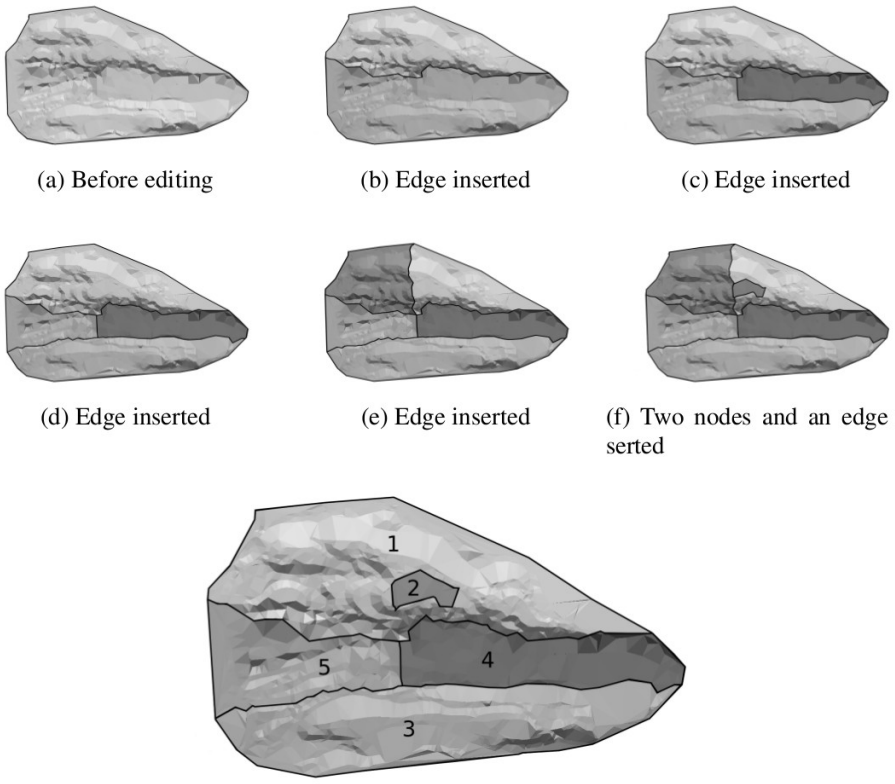
The advanced navigation capabilities of the Topology Module allow to easily implement an algorithm that finds the shortest path on top of the meshing of a cell net component. Since such an algorithm is helpful for editing methods, a DIJKSTRA-based path finding algorithm (Dijkstra 1959) has been designed and implemented that operates on top of G-Maps structure.

Given net level and object level, and having advanced navigation capabilities with orbit and cell iterators, it becomes feasible to develop methods with which the topology of cellular complexes can be edited. Editing methods have to obey sets of constraints in order to function properly. Thus, a `Constraint` class is designed that allows to define custom constraints. A constraint is represented by a stored query. The stored query is evaluated and the query result is checked against a target value. If the query result equals the target value then the constraint is active. An example of a constraint is that the amount of faces being adjacent to another certain face has to equal a specific number. A number of such

constraints is checked before an editing operation can be conducted. Editing operations are implemented prototypically in order to demonstrate feasibility: insert node, insert edge, delete node, and delete edge. The insert node method adds a new node onto an existing edge, which can modify the course of the edge. The method splits the existing edge on which it is inserted into two new edges. Delete node is the inverse function to insert node and removes an existing node form an edge, which also can modify the course of the edge and which merges the two former edges into one new common edge. The insert edge method adds a new edge onto a face between two given nodes. The course of the new edge has to be determined by a suitable method. The Topology Module uses the DIJKSTRA shortest path algorithm to determine the course of the edge. The new edge splits the existing face on which it is inserted into two new faces. Delete edge is the inverse function to insert edge and removes an existing edge from a face, which merges the two former faces into one new common face. Due to the concept of net level and object level, the topological changes of all these methods are performed only on object level.

The concept of net level and object level is not only necessary to facilitate editing capabilities, but it is also a sound preparation for the management of LoD of cell net components. Since net level and object level already exist, there is a basis which can be extended by additional detail levels. Additional detail levels are inserted in-between net level and object level. They are numbered ascending from lowest to highest LoD, i.e. the level with the lowest LoD, except for object level, has the index *1*. The additional detail levels follow the same principles as the object level, i.e. they are also editable. This is reflected in the class model where the `LOD` class extends object level class. This means that every detail level is an object level, except the net level. The connections between levels are still established finely granulated through `higher` and `lower` links of cell-tuples. The API user employs levels of detail in the editing process in order to add or remove detail. In the workflow, the API user first chooses a detail level to alter. Depending on whether intendedly to add or remove detail, the module creates a copy of the detail level above or below the original level. Then the API user can carry out the editing operations on the newly created level. Meanwhile, the module observes the consistency of all `higher` and `lower` links of the cell-tuples. Following BRUEGGER and KUHN, only refinement relations between levels of detail are permissible.

In order to demonstrate the application-side performance of LoD management, an example test case has been set up by GOLOVKO on the basis of the Piesberg landfill dataset (Breunig, Butwilowski, Kuper, et al. 2013, 9 et seq.). The test application has completely been set up by the means of Topology Module. The aim was to show an application example where the area of an initially homogeneous landfill site is subdivided into several subregions that could represent regions with different material properties of the soil or with different land use classes (see Fig. 104).



(a) Before editing (b) Edge inserted (c) Edge inserted

(d) Edge inserted (e) Edge inserted (f) Two nodes and an edge inserted

(g) Two edges removed. Land use classes: 1 - mining, 2 - wind energy, 3 - old landfill, 4 - active landfill, 5 - compost

Fig. 104: Example of editing session on Piesberg dataset (visualized with ParaviewGeo) Source: (Breunig, Butwilowski, Kuper, et al. 2013, 10)

Golovko extracted the geo-object from the Piesberg dataset (at an arbitrary year) as the starting point for the demonstration. In the first step, the Topology Module models the geo-object as a cell net component with one net level that represents the underlying meshing and one object level that represents the boundary (polyline) (see state *(a)* in Fig. 104). In a second step, a new edge is inserted between two points of the existing boundary in order to delimit the northern mining field of the site from the southern landfill area *(b)* (example fictitious). In the next step, she added another additional edge in order to subdivide the newly created southern field into two new fields *(c)*. Through multiple additional edge insertion operations, the Piesberg geo-object is stepwise subdivided into six subregions (see *(d)*, *(e)*, *(f)*). Afterwards, two edges that subdivided the mining field, were removed again so that the mining area is represented by one region finally *(g)*. Please note that as a result of the last edge removal operation, an island/hole remains insight the mining face. Due to model specifications of the Topology Module, this hole does not lead to an inconsistent state of the model.

The spatio-temporal model of the Topology Module builds on the foundation of the *Temporal Joint Model* of Kuper that has been implemented in DB4GeO Kernel. The Temporal Joint Model operates on Simplicial Complexes and models geometric change, but it does not track topological change of “big cells”. *Continuous geometric change* is

carried out in the form of altering of the support point coordinates of the Simplicial Complex through the Temporal Joint Model, whereas *discrete topological change* of "big cells" is modelled by the spatio-temporal model of the Topology Module. Since both models are in a loose coupling, it is possible to change the underlying meshing, while preserving the general topological configuration of object level cells over periods of time. Conversely, the editing of cells of object level triggers the creation of a new, different cell complex, while the underlying meshing stays unchanged. These capabilities of the spatio-temporal model are audited and demonstrated on the basis of a realistic temporal geo-object that is derived from Piesberg landfill site dataset.

Finally, different aspects of runtime and memory performance of the Topology Module are evaluated and compared to the performance of the DB4GeO Kernel. As an overall outcome can be stated that the Topology Module consumes more memory and needs more time for cell net construction (for details cf. Ch. 4). This is not surprising, since a cellular complex of the Topology Module needs a fully constructed Simplicial Complex of DB4GeO Kernel in first place. This means that runtime and memory of DB4GeO Kernel are mandatory for construction process. But in operation mode, the Topology Module provides significant benefits in runtime behaviour, which shows that the structure can be designated as a "topological index".

5.2 Outlook

In future, the Topology Module can be used as the basis for advanced developments, e.g. to implement new high level functionalities in DB4GeO, such as a *CityGML exporter*.

5.2.1 The Topology Module as Basis for a DB4GeO CityGML Im-/Exporter

As noted in the introduction (especially in Ch. 1.3), the integration of multiple geometric and topological models (especially B-Rep and Simplicial Complex) into one architecture, can be of great value in particular application scenarios. In order to provide an example of the added value of functionality that uses an integrated model, Daria GOLOVKO implemented *OpenGisDb3dModule*. *OpenGisDb3dModule* mainly is a prototypical CityGML exporter/importer for DB4GeO on the basis of the Topology Module.

GOLOVKO's module internally uses *citygml4j* in order to read and write the XML exchange format of CityGML. *citygml4j* is a Java library that provides classes of GML and CityGML model (e.g. `MultiSurface` or `Building`) and an XML compiler that transforms XML data into the Java classes. In order to combine *citygml4j* with the Topology Module, GOLOVKO developed a translation between the classes of both models and implemented a polygon triangulation mechanism. For example, the topology of a `CompositeSurface` in *citygml4j* is internally modelled as a `FaceNet3dComp` of the Topology Module. With this concept, the geo-objects of CityGML can internally be represented by B-Rep of object level and by Simplicial Complex of net level simultaneously (see Fig. 105).

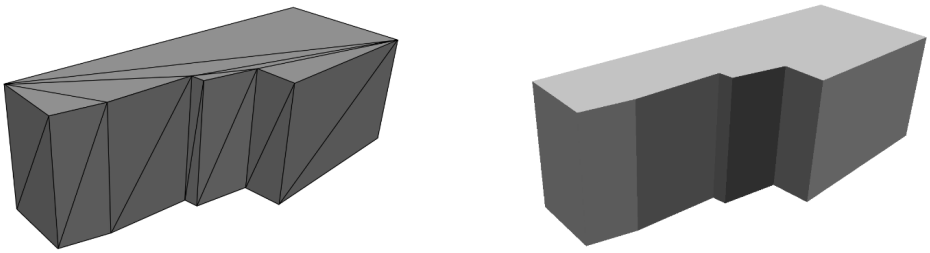


Fig. 105: Net level and object level representation of building model
Source: (Breunig, Butwilowski, Golovko, et al. 2013, 102), visualized with ParaViewGeo

The depiction shows a building model which is a detail of a CityGML dataset from British Ordnance Survey (OS) that has been made available as test data on *citygml.org*. The CityGML XML file of OS (displayable by CityGML software such as Autodesk® LandXplorer) has been imported into DB4GeO with the Topology Module installed and then exported from DB4GeO as GOCAD format (*.ts*) (displayable by subsurface visualizers such as ParaViewGeo). Though, the original geo-object stems from the field of city models (and thus uses B-Rep), it can also be loaded into subsurface modelling software (in the form of Simplicial Complex). The illustration in Fig. 105 shows the OS geo-object in the subsurface visualizer ParaViewGeo. The right side shows the polygons (B-Rep) of the geo-object that were modelled on object level by the Topology Module. The left side illustrates the underlying triangulation (Simplicial Complex) of the same geo-object that was modelled on net level.

5.2.2 Direct Integration of the Topology Module into DB4GeO Kernel

Currently, the Topology Module is implemented as a plug-in in order to study architectural modifications gradually step-by-step. Now that the Topology Module matured very well, it becomes feasible to integrate it directly into the main development repository of DB4GeO and replace the previous kernel. The replacement process could also be used as an opportunity to refine some architectural arrangements that were necessary as a consequence of the plug-in approach. The plug-in approach forced a class architecture where simplexes are part of cells. Though, the approach is feasible and leads to useful results, it is not perfectly clean from the viewpoint of OOM. From an OOM perspective, cell could be seen as the base interface not only for concrete cells but also for simplices (see Fig. 106).

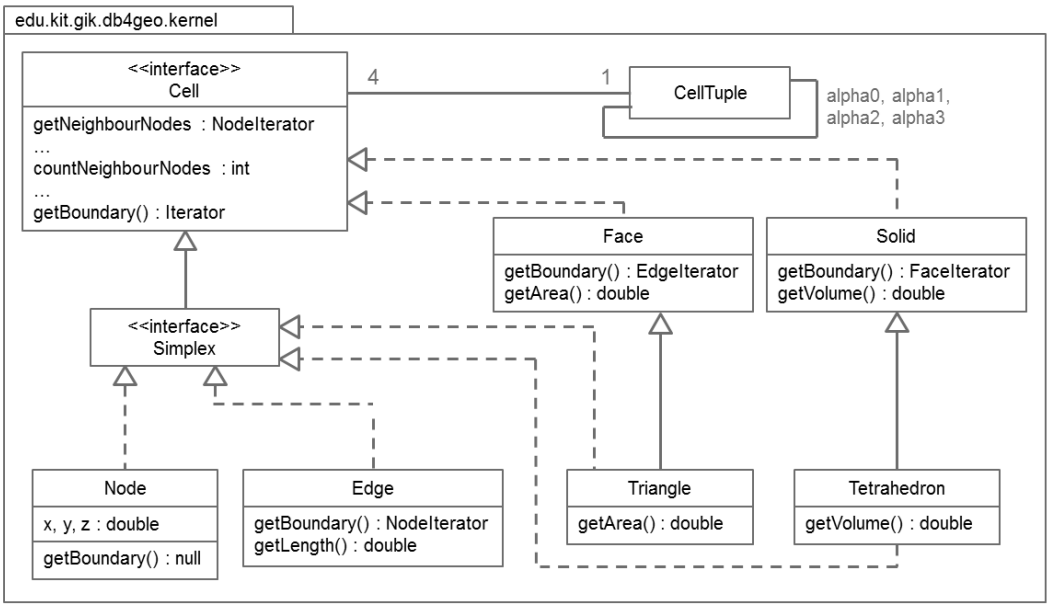


Fig. 106: Cell complex model for direct integration into DB4GeO Kernel

The diagram shows a model with the `Cell` interface as the fundamental basis for all geometric and topological elements. Like in the current model of the Topology Module, the topology information of cells is managed and stored in objects of the `CellTuple` class. Unlike in the model of the Topology Module, to keep things simple, the coordinates of geometry are directly stored in the `Node` class (there is no intermediate `Point3D` class any more).

A `Simplex` interface is derived from the `Cell` interface. This means that in this model, *a simplex is a cell*, which more corresponds to the terms of the mathematical definition. The interfaces `Cell` and `Simplex` are realized by concrete cell and simplex classes. Node, edge, triangle, and tetrahedron are modelled as simplices. Thereby, they are indirectly also cells. Face and solid are directly modelled as cells. Thereby, they are not simplices (since they do not realize `Simplex`). This is a sensible predicate, since faces and solids can be of any shape in this model.

If a face consists of only three nodes/points, then it is a triangle. In such cases faces can be modelled by the `Triangle` class, which is a subclass of `Face` class, saying that a triangle is a specialized face. The situation is similar concerning solids. If a solid consists of only four points, then it is a tetrahedron, modelled by the `Tetrahedron` class (as subclass of `Solid`).

All cells (and thereby all simplices) have to provide the topological methods `getNeighbour<Cell>`, `countNeighbour<Cell>`, and `getBoundary()`. These methods are internally working similar for all types of cells. A specialized method of the `Face` class is `getArea()` that calculates the surface size of the arbitrarily shaped face. Similarly, the `Solid` class has a `getVolume()` method for volume size calculation. The area and volume methods can be overridden by the `getArea()` and `getVolume()` methods of the `Triangle` and `Tetrahedron` class. By this, the simplex properties of the

triangle and tetrahedron can be utilized to employ simpler, more cost-effective algorithms for area and volume calculation.

It is assumed that the direct integration of this model into the kernel will render even cleaner and consolidated source code since it will remove several currently necessary code doublings.

5.2.3 Approach to Dimension-independent Cell Model

The present cell model of the Topology Module is limited to dimensions *0* to *3*. This limitation is “hard wired” by the class model, since the naming of the concrete cell classes is dimension-dependent (`Node` for 0-d, `Edge` for 1-d, `Face` for 2-d, and `Solid` for 3-d see Fig. 40). The algorithms of the Topology Module reflect the class model, so that the dimension-dependency is also reflected in most of the topological algorithms of the module. A limitation of the considered dimensions is a reduction of the conceptual complexity that is helpful for the first steps. It helps to develop a first impression of the interactions between class model and algorithms.

However, now that extensive experience could be gained with the model, it becomes feasible to create a dimension-independent cell class model in the DB4GeO Kernel. During the development of the Topology Module, it was remarkable that many algorithms of individual cell types had close similarity. To be precise, many algorithms could have been reused in different cell classes if there had been no fixed naming of cell types. It is assumed that the utilization of a dimension-independent cell class model will additionally reduce model and code complexity (see Fig. 107).

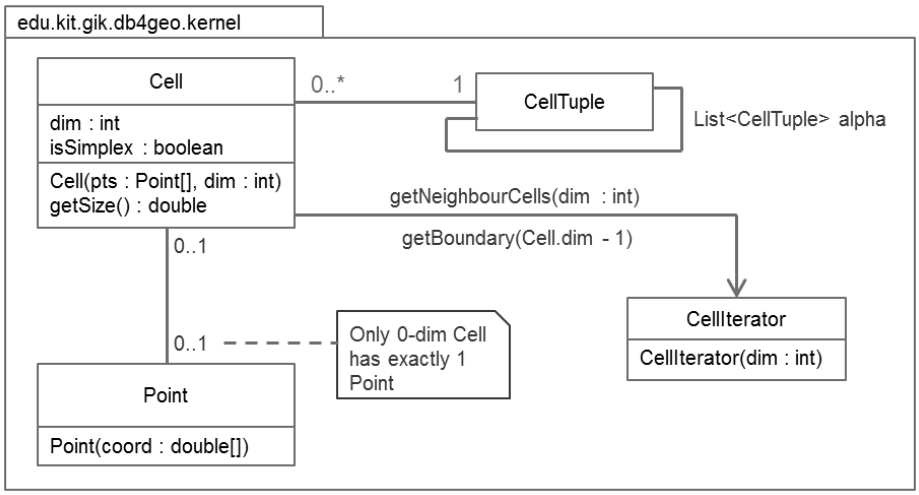


Fig. 107: Classes of dimension-independent cell model approach

At a first glance it is already apparent that the class model in Fig. 107 is more concise than the preceding model in Fig. 106, though it serves the same purpose.

Again, `Cell` is the key class of the model. The cell itself is indeed a class, not an interface anymore, since it can realize all the needed functionality itself.[156] The `Cell` class has a *dimension* (`dim`) property of *integer* type, so that the dimension can be set as required. Since G-Maps are used for topology modelling, the `Cell` class is associated to one `CellTuple`. `CellTuples` have the ability to manage an arbitrary number of self-referencing `alpha` associations in a `List`. Practically, the number of `alpha` associations equals the maximum `dim` of all cells. Since the dimension of cells and cell-tuples is not fixed, the topological algorithms can also be written dimension-independent.

The `dim` property of the `Cell` class has to be provided by the API user who intends to instantiate a `Cell` object. He provides it as a parameter value of the class constructor. Additionally, the API user also provides an *array of support points* for the cell's geometry through `pts` constructor parameter value of type `Point`. Though, it is possible to provide any number of points as support points, one `Cell` object can associate *not more than one* `Point`. The idea behind this construct is that if a cell of higher dimension with more than one support points (e.g. edge or face) shall be created, then the constructor implicitly has to create all incident cells of all lower dimensions. In this process, *only the 0-dimensional cells* (nodes) are the carriers of the geometry information, i.e. only they get a reference to a `Point` object. Each `Point` object has an array of geometry coordinates in the `coord` class field property. Finally, the constructor sets the 0-dimensional cells as part of the higher dimensional cells.

If the API user invokes the `Cell` constructor with a *minimum set* of support points for a given dimension (e.g. two support points for a 1-cell or three support points for a 2-cell), then the constructor automatically sets the `isSimplex` class property to `true`, indicating that the cell is a simplex. The `Cell` class also provides a `getSize` method that returns the size of spatial extent of the cell. The meaning of the return value of `getSize` depends on the dimension of the cell, e.g. length for 1-cell, area for 2-cell, or volume for 3-cell. Additionally, the calculation algorithm has to consider whether the `isSimplex` flag is set, since the costs for calculation of spatial extent size can be reduced in the case of a simplex.

Similar to the preceding model in Fig. 106, the dimension-independent model provides a `getNeighbourCells` method in `Cell` class. But in this case, the method and return type are dimension-independent. Thus, only one method and only one return type (`CellIterator`) instead of four is needed. The `getNeighbourCells` method requires a `dim` parameter of type `int` in order to return the neighbouring cells of the demanded dimension. For example if the API user has a face (2-cell) and needs to retrieve all edges (1-cells) of the face, then `getNeighbourCells(1)` has to be invoked on the 2-cell. Then the resulting `CellIterator` will be of dimension *1*, i.e. it will iterate over all neighbouring 1-cells. Analogously to the Topology Module API, the method returns *incident* cells if the dimension of the cell iterator is *different* than the dimension of the invoking cell (`Cell.dim != CellIterator.dim`), and it returns *adjacent* cells if the

156 The implementation of concrete cell classes (Node, Edge etc.) can be omitted.

dimension of the cell iterator *equals* the dimension of the invoking cell (`Cell.dim == CellIterator.dim`). The `getBoundary` method of `Cell` is based on the functionality of `getNeighbourCells` method and simply invoked `getNeighbourCells` with a dimension value that is *1* dimension lower than the dimension of the given cell (`Cell.dim – 1`).

The fundamental classes, properties, and methods of a proposal for a dimension-independent model have been presented. First observations suggest that such an approach will provide a leaner cell class model with reduced and cleaner code. However, the usefulness of higher-dimensional cell-tuples is debatable, since the growth of the cell-tuple structure is strong in relation to dimension. The number of darts grows exponentially with each higher dimension.

5.2.4 Comparison of the Topological Index with Classical Indices

The performance metrics in Ch. 4 already give some interesting insights into the advantages and disadvantages of using the Topology Module, considering its runtime and memory usage. However, the metrics only cover a comparison between data structures of the Topology Module on the one side and unordered sequences of DB4GeO on the other side, in order to demonstrate the index-like behaviour of the Topology Module. The resulting metrics suggest to consider the Topology Module as a “topological index”. In the next step, it would also be interesting to compare the runtime behaviour of the topological index vs. the indices that are already built in into DB4GeO Kernel such as the R*-tree and the Octree, in order to determine the performance of the topological data structure as an index compared to “classical” indices. Test queries could be typical spatial DB queries that search for neighbouring, incident, and adjacent geo-objects.

Bibliography

Abran, A., and J. W. Moore. 2001. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. Los Alamitos, California: IEEE Computer Society Press, U.S.

Alexandroff, P., and H. Hopf. 1935. *Topologie I*. Edited by R. Courant, W. Blaschke, F. K. Schmidt, and B. L. Van der Waerden. Vol. 45. Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen mit besonderer Berücksichtigung der Anwendungsgebiete. Berlin: Verlag von Julius Springer.

Alms, R., O. Balovnev, M. Breunig, A. B. Cremers, T. Jentzsch, and A. Siehl. 1998. “Space-Time Modelling of the Lower Rhine Basin Supported by an Object-Oriented Database.” *Physics and Chemistry of the Earth* 23 (3): 251–60.

Andenmatten, N., and T. Kohl. 2002. *Rapport Pour La Commission Géophysique Suisse, 2002*. http://www.sgpk.ethz.ch/jahresbericht/2002/Kohl.htm.

Anders, K.-H., and J. Bobrich. 2004. “MRDB Approach for Automatic Incremental Update.” In *Proceedings of ICA Workshop on Generalisation and Multiple Representation*. Leicester.

Andrae, C. 2008. *Spatial Schema. ISO 19107 und ISO 19137 vorgestellt und erklärt*. OpenGIS essentials. Die Geo-Standards von OGC und ISO im Überblick. Heidelberg: Wichmann.

Balovnev, O., T. Bode, M. Breunig, A. B. Cremers, W. Müller, G. Pogodaev, S. Shumilov, J. Siebeck, A. Siehl, and A. Thomsen. 2004. “The Story of the GeoToolKit - An Object-Oriented Geodatabase Kernel System.” *GeoInformatica* 8 (1): 5–47.

Bär, W. 2007. “Verwaltung geowissenschaftlicher 3D Daten in mobilen Datenbanksystemen.” PhD thesis, University of Osnabrück.

Baumgart, B. G. 1975. “A Polyhedron Representation for Computer Vision.” In *Proceedings of the National Computer Conference and Exposition*, 589–96. AFIPS ’75. New York, NY, USA: ACM.

Bayer, R., and E. M. McCreight. 1972. “Organization and Maintenance of Large Ordered Indexes.” *Acta Informatica* 1 (3): 173–89.

Blasby, D. 2001. “Building a Spatial Database in PostgreSQL.” presented at the Open Source Database Summit. http://postgis.refractions.net/files/OSDB2_PostGIS_Presentation.ppt.

Breunig, M. 2001. *On the Way to Component-Based 3D/4D Geoinformation Systems*. Lecture Notes in Earth Sciences 94. Berlin: Springer.

Breunig, M., B. Broscheit, A. Thomsen, E. Butwilowski, M. Jahn, and P. V. Kuper. 2009. “Towards a 3D/4D Geo-Database Supporting the Analysis and Early Warning of Landslides.” In *Cartography and Geoinformatics for Early Warnings and Emergency Management: Towards Better Solutions*, 100–110. Prague, Czech Republic.

Breunig, M., E. Butwilowski, D. Golovko, P. V. Kuper, M. Menninghaus, and A. Thomsen. 2013. “Advancing DB4GeO.” In *Progress and New Trends in 3D Geoinformation Sciences*, edited by J. Pouliot, S. Daniel, F. Hubert, and A. Zamyadi, 193–210. Lecture Notes in Geoinformation and Cartography. Berlin Heidelberg: Springer.

Breunig, M., E. Butwilowski, P. V. Kuper, D. Golovko, and A. Thomsen. 2013. “Topological and Geometric Data Handling for Time-Dependent Geo-Objects Realized in DB4GeO.” In *Advances in Spatial Data Handling*, edited by S. Timpf and P. Laube, 1–13. Advances in Geographic Information Science. Berlin Heidelberg: Springer.

Breunig, M., E. Butwilowski, P. V. Kuper, N. Paul, A. Thomsen, S. Schmidt, and H.-J. Götze. 2011. “Handling of Spatial Data for Complex Geo-Scientific Modelling and 3D Landfill Applications With DB4GeO.” In *Geoinformatik 2011. Geochange*, edited by A. Schwering, E. Pebesma, and K. Behnke, 15–20. Schriftenreihe des Instituts für Geoinformatik. Heidelberg: Akademische Verlagsgesellschaft AKA GmbH.

Breunig, M., B. Schilberg, A. Thomsen, P. V. Kuper, M. Jahn, and E. Butwilowski. 2009. “DB4GeO: Developing 3D Geo-Database Services.” In *Proceedings of the Fourth International 3DGeoInfo Workshop*, 45–52. Ghent, Belgium.

———. 2010. “DB4GeO, a 3D/4D Geodatabase and Its Application for the Analysis of Landslides.” In *Geographic Information and Cartography for Risk and Crisis Management*, edited by M. Konecny, S. Zlatanova, and T. L. Bandrova, 83–101. Berlin, Heidelberg: Springer Berlin Heidelberg.

Brinkhoff, T. 2008. *Geodatenbanksysteme in Theorie und Praxis: Einführung in objektrelationale*

Geodatenbanken unter besonderer Berücksichtigung von Oracle Spatial. 2nd ed. Wichmann.
Brisson, E. 1989. “Representing Geometric Structures in D Dimensions: Topology and Order.” In *Proceedings of the Fifth Annual Symposium on Computational Geometry*, 218–27. SCG ’89. New York, NY, USA: ACM.
Bruegger, B. P., and W. Kuhn. 1991. *Multiple Topological Representations*. Ongoing research report. Orono: University of Maine. http://www.ncgia.ucsb.edu/Publications/Tech_Reports/91/91-17.pdf.
Brugman, B. 2010. “3D Topological Structure Management within a DBMS. Validating a Topological Volume.” MSc thesis, Delft: TU Delft.
Buttenfield, B. P. 1993. *Research Initiative 3: Multiple Representations, Closing Report*. Closing report. Buffalo, NY: National Center for Geographic Information and Analysis.
Butwilowski, E. 2007. “Topologische Fragestellungen bei der Kombination von 3D-Stadtmodellen mit 2D-Karten in einer Räumlichen Datenbank.” Diploma thesis, Osnabrück: University of Osnabrück.
Cremers, A. B., A. Siehl, W. Förstner, O. Balovnev, M. Breunig, M. Pant, S. Shumilov, J. Flören, M. Hammel, and W. Müller. 2000. *Teilprojekt D4 - Objektorientiertes 3D/4D-Geoinformationssystem*. http://www.geo.informatik.uni-bonn.de/publications/2000/sfb- d4/D4-Bericht-2000.pdf.
Dadam, P., V. Lum, and H.-D. Werner. 1984. “Integration of Time Versions into a Relational Database System.” In *Proceedings of the 10th International Conference on Very Large Data Bases*, 509–22. Singapore.
Dijkstra, E. W. 1959. “A Note on Two Problems in Connexion with Graphs.” *Numerische Mathematik* 1 (1): 269–71.
Dorffner, L., M. Ludwig, and G. Forkert. 2006. “Adding Another Dimension to Municipal Management.” http://www.esri.com/news/arcuser/0706/3d-planning1of2.html.
EEA GmbH. 2012. “Earth Energy Aanalytics & Development GmbH.” Accessed March 8. http://beyondwind.net/.
Ellul, C., and M. Haklay. 2006. “Requirements for Topology in 3D GIS.” Edited by J. P. Wilson, A. S. Fotheringham, and D. O’Sullivan. *Transactions in GIS* 10 (2): 157–75.
ESRI. 2010. “GIS Mapping Solutions for Industry.” *Esri - The GIS Software Leader*. Accessed April 15. http://www.esri.com/industries.html.
Fang, Y., M. Friedman, G. Nair, M. Rys, and A.-E. Schmid. 2008. “Spatial Indexing in Microsoft SQL Server 2008.” In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 1207–16. SIGMOD ’08. New York, NY, USA: ACM.
Fielding, R. T., and R. N. Taylor. 2002. “Principled Design of the Modern Web Architecture.” *ACM Transactions on Internet Technology* 2 (2): 115–50.
Fradin, D., D. Meneveaux, and P. Lienhardt. 2005. *Hierarchy of Generalized Maps for Modeling and Rendering Complex Indoor Scenes*. 2005-04. Rapport de Recherche. Poitiers: University of Poitiers. http://xlim-sic.labo.univ-poitiers.fr/publications/files/publi790.pdf.
Gabriel, P., J. Gietzel, H. H. Le, and H. Schaeben. 2011. “A Network Based Datastore for Geoscience Data and Its Implementation.” In *Proceedings of the 31st Gocad Meeting 2011*. Nancy, France.
Gabriel, P., J. Gietzel, and H. Schaeben. 2010. “A Framework for a Networkbased Datastore for Spatial and Spatio-Temporal Geoscience Data.” In *Proceedings of the 30th Gocad Meeting 2010*. Nancy, France.
Götze, H., and B. Lahmeyer. 1988. “Application of Three-dimensional Interactive Modeling in Gravity and Magnetics.” *Geophysics* 53 (8): 1096–1108.
Hägerstrand, T. 1975. “Space, Time and Human Conditions.” In *Dynamic Allocation of Urban Space*, edited by A. Karlsqvist, L. Lundqvist, and F. Snickars, 3–14. Farnborough: Saxon House.
Haunert, J.-H., and M. Sester. 2005. “Propagating Updates between Linked Datasets of Different Scales.” In *Proceedings of the XXII International Cartographic Conference*, 11–16. A Coruña, Spain.
Hoppe, H. 1996. “Progressive Meshes.” In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, 99–108. New York, NY: ACM.
IAI (Institut für Angewandte Informatik Karlsruhe). 2011. “Semantische Datenmodelle für die Geothermie.” Accessed September 9. http://www.iai.fzk.de/www-extern/index.php?id=1810.
ISO. 2003. “ISO 19107: Geographic Information - Spatial Schema.”
———. 2011. “ISO/IEC 13249-3: Information Technology -- Database Languages -- SQL Multimedia and Application Packages -- Part 3: Spatial.”
Kolbe, T., and G. Gröger. 2003. “Towards Unified 3D City Models.” In *Proceedings of the ISPRS Comm. IV Joint Workshop on Challenges in Geospatial Analysis, Integration and Visualization II*.

Stuttgart, Germany.
Krämer, M., T. Ruppert, E. Klien, and J. Kohlhammer. 2010. “DeepCity3D: Integration von 3D-Stadtmodellen und Untergrundinformationen.” In *Geoinformatik 2010. Die Welt Im Netz*, edited by A. Zipf, 72–80. Heidelberg: Akademische Verlagsgesellschaft Aka.
Krimmelbein, A. 2011. “Topologie in CityGML.” Master thesis, Karlsruhe: Karlsruhe Institute of Technology.
Kuper, P. V. 2010. “Entwicklung einer 4D Objekt-Verwaltung für die Geodatenbank DB4GeO.” Diploma thesis, Osnabrück: University of Osnabrück.
Langran, G., and N. R. Chrisman. 1988. “A Framework For Temporal Geographic Information.” *Cartographica: The International Journal for Geographic Information and Geovisualization* 25 (3): 1–14.
Lautenbach, S., and J. Berlekamp. 2002. *Datensatz Zur Visualisierung Der Zentraldeponie Piesberg in Osnabrück*. Osnabrück: Institut für Umweltsystemforschung, University of Osnabrück.
Lester, M. 1990. “Tracking the Temporal Polygon: A Conceptual Model of Multidimensional Time for Geographic Information Systems.” In *Proceedings of the The Temporal Workshop, NCGIA*, edited by R. Barrera, A. U. Frank, and K. Al-Taha. Orono.
Lévy, B., and J.-L. Mallet. 1999. *Cellular Modelling in Arbitrary Dimension Using Generalized Maps*. Technical report. ISA-GOCAD (Inria-Lorraine/CNRS). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.1323.
Lévy, B. 2000. “Topologie Algorithmique Combinatoire et Plongement.” PhD thesis, Centre National de la Recherche Scientifique.
Lienhardt, P. 1989. “Subdivisions of N-Dimensional Spaces and N-Dimensional Generalized Maps.” In *Proceedings of the Fifth Annual Symposium on Computational Geometry*, 228–36. SCG ’89. New York, NY: ACM.
Mäntylä, M. 1988. *An Introduction to Solid Modeling*. Rockville, Md: W.H. Freeman & Company.
Münchener Rück. 2004. “Erneuerbare Energien. Versicherung einer Zukunftstechnologie.” files.globalmarshallplan.org/muenchner_rueck_20040601_de.pdf.
Open Geospatial Consortium. 2008. “OpenGIS® City Geography Markup Language (CityGML) Encoding Standard.”
———. 2011. “OpenGIS Implementation Specification for Geographic Information - Simple Feature Access - Part 1: Common Architecture.”
Ott, T., and F. Swiaczny. 2001. *Time-Integrative Geographic Information Systems - Management and Analysis of Spatio-Temporal Data*. Heidelberg: Springer.
Patenge, K. 2010. “Oracle Spatial Goes Standard.” presented at the DOAG SIG Spatial 2010.
Peuquet, D. J. 1984. “A Conceptual Framework and Comparison of Spatial Data Models.” *Cartographica: The International Journal for Geographic Information and Geovisualization* 21 (4): 66–113.
Picavet, V. 2010. “State of the Art of FOSS4G for Topology and Network Analysis.” presented at the FOSS4G 2010, Barcelona. http://2010.foss4g.org/presentations/3555.pdf.
Polthier, K, and M. Rumpf. 1995. “A Concept for Time-Dependent Processes.” In *Visualization in Scientific Computing*, edited by M. Göbel, H. Müller, and P. Urban, 137–53. Heidelberg: Springer.
Pouliot, J., and F. Fallara. 2007. *3D Geological Model of Porcupine-Destor Fault (north Québec, Canada) Containing Surfaces and Tetrahedral Solids Built in Gocad©*.
Ravada, S., and J. Sharma. 1999. “Oracle8i Spatial: Experiences with Extensible Databases.” In *Advances in Spatial Databases*, edited by R. H. Güting, D. Papadias, and F. Lochovsky, 1651:355–59. Berlin, Heidelberg: Springer.
Raza, A., and W. Kainz. 1999. “Cell Tuple Based Spatio-Temporal Data Model: An Object Oriented Approach.” In *Proceedings of the 7th ACM International Symposium on Advances in Geographic Information Systems*, 20–25. GIS ’99. New York, NY, USA: ACM.
Refractions Research. 2012. “Using PostGIS: Data Management and Queries.” *PostGIS 1.5.3 Manual*. Accessed March 19. http://postgis.refractions.net/documentation/manual-1.5/ch04.html.
Remmert, R. 1964. “Topologie: Topologische Mannigfaltigkeit.” In *Das Fischer-Lexikon. Mathematik 1*, edited by H. Behnke, R. Remmert, H.-G. Steiner, and H. Tietz, I:307–8. Frankfurt am Main: Fischer Bücherei.
Rhind, D. W. 1973. “Generalisation and Realism Within Automated Cartographic Systems.” *Cartographica: The International Journal for Geographic Information and Geovisualization* 10 (1): 51–62.
Rolfs, C. 2005. “Konzeption und Implementierung eines Datenmodells zur Verwaltung von

zeitabhängigen 3D-Modellen in geowissenschaftlichen Anwendungen.” Diploma thesis, Osnabrück: University of Osnabrück.

Royer, J.-J. 2004. “3D Modeling and Visualization.” In *Proceedings of the 9th International CODATA Conference: Data Visualization–Earth and Geo Science*, 7–10. Berlin.

Santilli, S. 2011. “Topology with PostGIS 2.0.” *PostgreSQL Sessions. PostGIS Day Paris 2011 Conference*. http://www.postgresql-sessions.org/2/sandro_santilli_-_topology_with_postgis_2.0.

Schaeben, H., M. Apel, K. G. v. d. Boogaart, and U. Kroner. 2003. “GIS 2D, 3D, 4D, nD.” *Informatik-Spektrum* 26 (3): 173–79.

Schildt, H. 2011. *Java The Complete Reference*. 8th ed. New York City: McGraw-Hill Osborne Media.

Sester, M., C. Heipke, R. Klein, and H.-P. Bähr. 2008. “Editorial: Abstraktion von Geoinformation bei der multiskaligen Erfassung, Verwaltung, Analyse und Visualisierung.” *Photogrammetrie, Fernerkundung, Geoinformation* 2008 (3): 153–55.

Shoham, Y., and N. Goyal. 1988. “Temporal Reasoning in Artificial Intelligence.” In *Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence*, edited by H. E. Shrobe, 419–38. San Francisco, CA: Morgan Kaufmann Publishers Inc.

Shumilov, S., A. Thomsen, A. B. Cremers, and B. Koos. 2002. “Management and Visualization of Large, Complex and Time-Dependent 3D Objects in Distributed GIS.” In *Proceedings of the 10th ACM International Symposium on Advances in Geographic Information Systems*, edited by A. Voisard and S.-C. Chen, 113–18. GIS ’02. New York, NY, USA: ACM.

Siebeck, J. 2003. “Concepts for the Representation, Storage, and Retrieval of Spatio-Temporal Objects in 3D/4D Geo-Information-Systems.” PhD thesis, Bonn: Rheinische Friedrich-Wilhelms-Universität.

Snodgrass, R., and I. Ahn. 1985. “A Taxonomy of Time Databases.” In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, 236–46. SIGMOD ’85. New York, NY, USA: ACM.

Thomsen, A., and M. Breunig. 2007. “Some Remarks on Topological Abstraction in Multi Representation Databases.” In *Proceedings of 3rd Internat. Workshop on Information Fusion and Geographical Information Systems IF&GIS-07, St. Petersburg*, edited by V. V. Popovich, M. Schrenk, and K. V. Korolenko, 234–51. Berlin, Heidelberg: Springer.

Thomsen, A., M. Breunig, E. Butwilowski, and B. Broscheit. 2008. “Modelling and Managing Topology in 3D Geoinformation Systems.” In *Advances in 3D Geoinformation Systems*, edited by P. Oosterom, S. Zlatanova, F. Penninga, and E. M. Fendel, 229–46. Lecture Notes in Geoinformation and Cartography. New York, NY: Springer Science+Business Media.

University of Chicago Press, ed. 2010. *The Chicago Manual of Style: The Essential Guide for Writers, Editors and Publishers*. 16th ed. Chicago: University of Chicago Press.

Wachowicz, M. 1999. *Object-Oriented Design for Temporal GIS*. London: Taylor & Francis.

Waldura, R. 2007. “Dijkstra’s Shortest Path Algorithm in Java.” http://renaud.waldura.com/doc/java/dijkstra/.

Weiler, K. 1985. “Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments.” *IEEE Computer Graphics and Applications* 5 (1): 21–40.

———. 1988. “The Radial Edge Structure: A Topological Representation for Non-Manifold Geometric Boundary Modeling.” Edited by M.J. Wozny, H.W. McLoughlin, and J.L. Encarnaçao. *Geometric Modeling for CAD Applications*, 3–36.

Weisstein, E. W. 2014. “Cell.” *MathWorld - A Wolfram Web Resource*. Accessed December 1. http://mathworld.wolfram.com/Cell.html.

———. 2015. “Möbius Strip.” Text. *Wolfram MathWorld*. Accessed April 6. http://mathworld.wolfram.com/MoebiusStrip.html.

———. 2010. “Simplicial Complex.” Text. *MathWorld - A Wolfram Web Resource*. Accessed July 22. http://mathworld.wolfram.com/SimplicialComplex.html.

Worboys, M.-F. 1994. “A Unified Model for Spatial and Temporal Information.” *Computer Journal* 37 (1): 26–34.

Subject Index