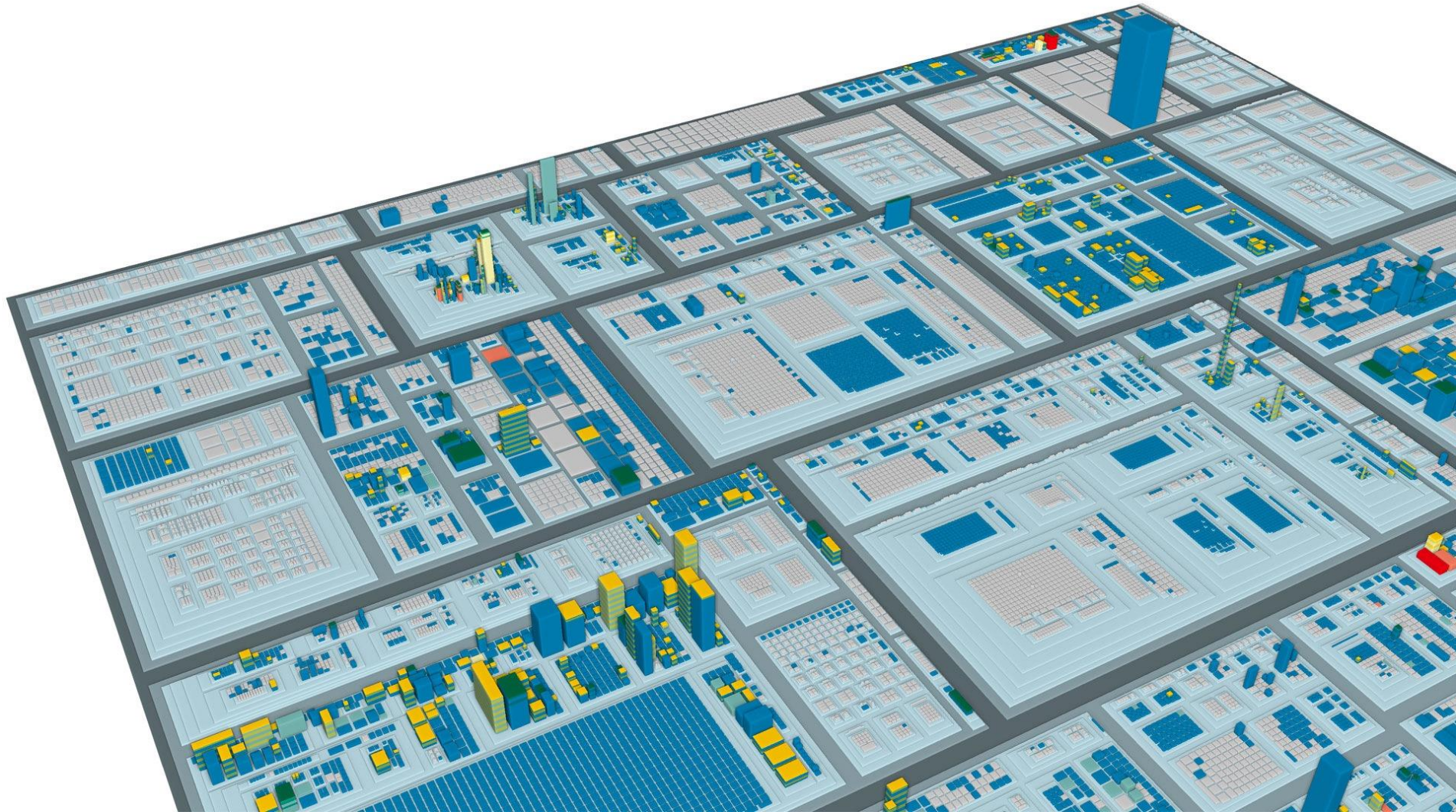
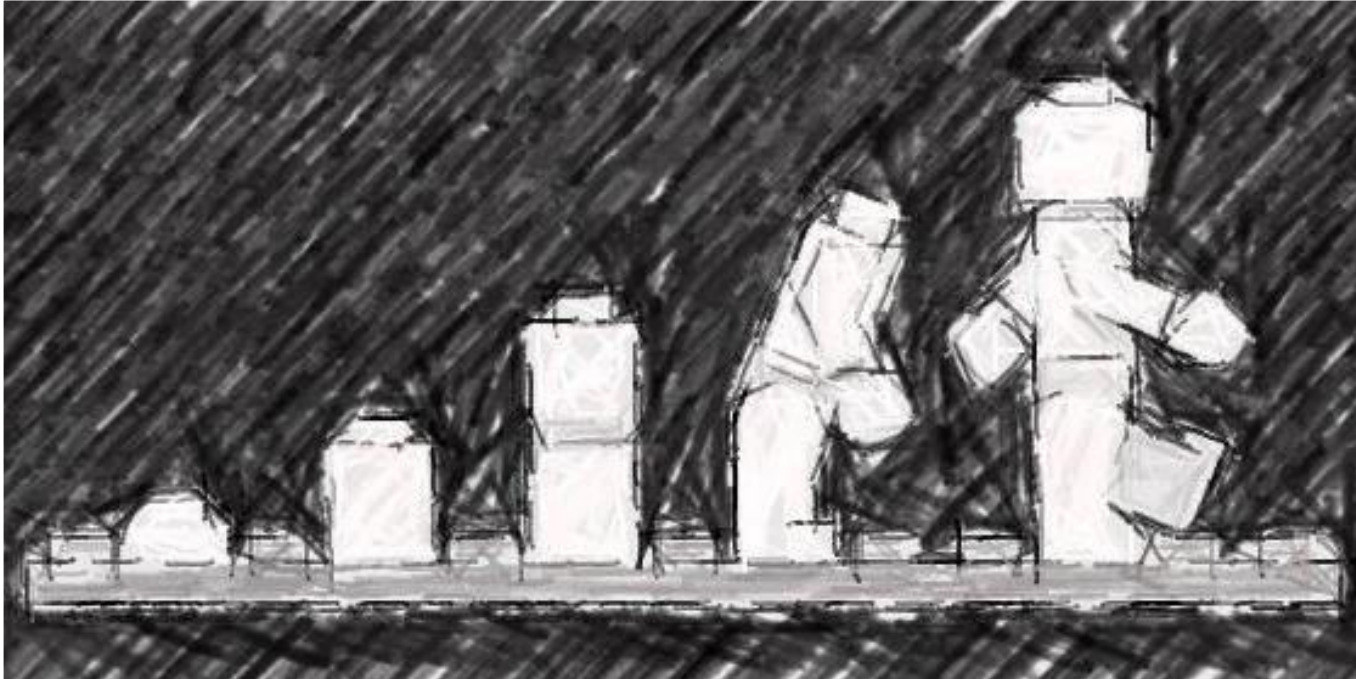


Automated Visual Software Analytics

OpenHPI – Prof. Dr. Jürgen Döllner – 2015



Version Control Systems

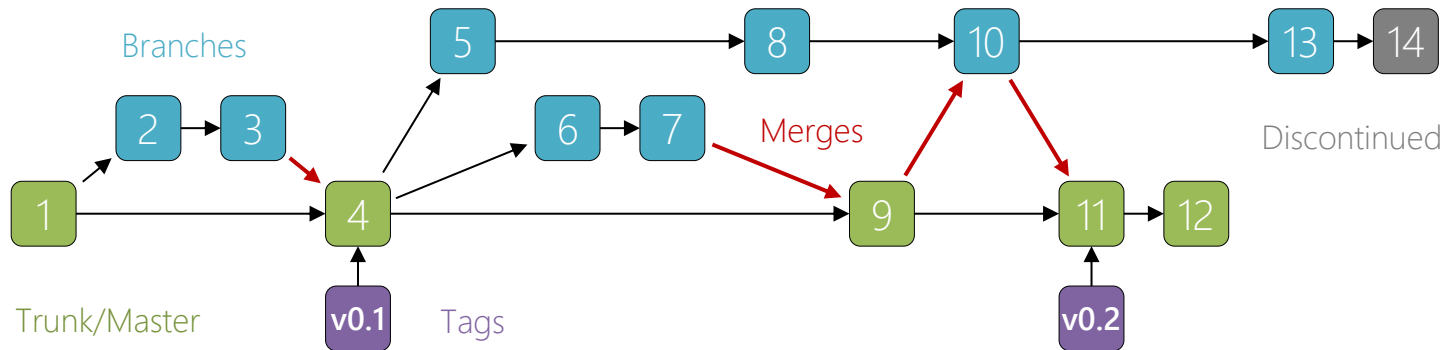


Principles of Version Control Systems
Example
Repository Data

Version Control Systems

- A *version control system* (VCS) manages changes applied over time to system artifacts such as source code, documents, models, web sites, and any other related representation of a system component.
- Version control systems represent a key type of **software repository**.

Principles of Version Control Systems



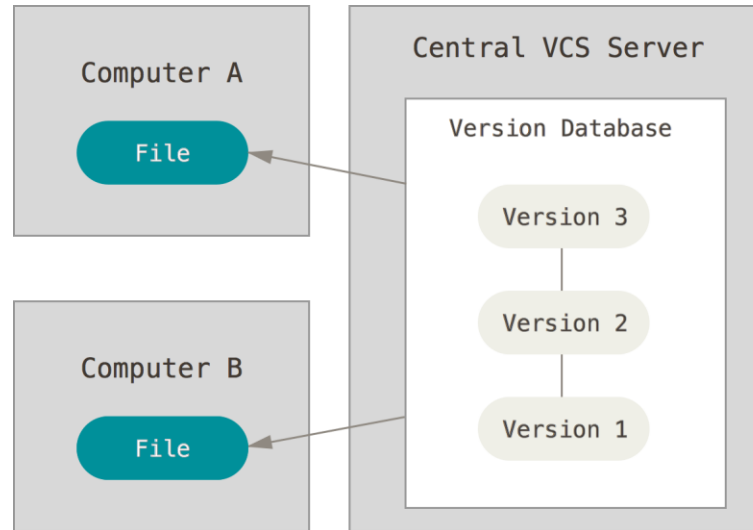
Features

- Before files can be edited, they have to be **checked out/cloned** by a user.
- When changes of one or more files are **checked in** (commit), the system records the date, the user that applied the change, and further information such as texts that describe the reason for the change (metadata).
- It allows to split the current state of the system, resulting in one or more **branches**. The main development is tracked in the **trunk/master**.
- The transition from a state to a new state due to a commit causes a new **revision**.
- **Tags** can be used to mark special revisions (e.g., releases).

Applications of Version Control Systems

- Systematically and completely reconstruct all changes, called the commit history, applied to artifacts of a system
- Record changes in terms of insertions, deletions or modifications applied to artifacts
- Recall versions and compare different versions of artifacts
- Identify who applied changes to a specific artifact and when
- Analyze and comprehend the introduction of bugs, bug fixing, and evolution of software features to support software maintenance tasks

Principles of Version Control Systems



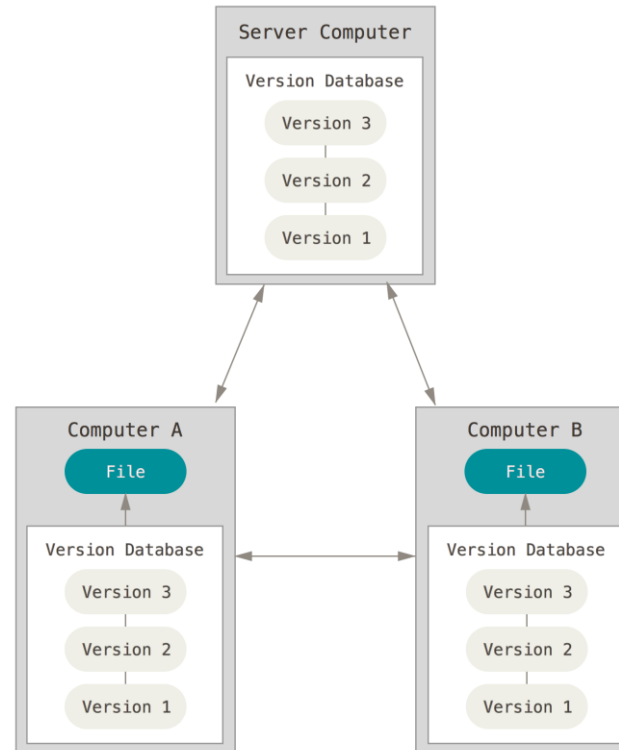
Centralized VCS

- One centralized server manages the repository.
- Each developer directly pushes and pulls from central repository.
- Server represents the "single point of failure".

– <http://git-scm.com/book/en/v2/>

Examples: CVS, SVN, Team Foundation Server, Perforce

Principles of Version Control Systems



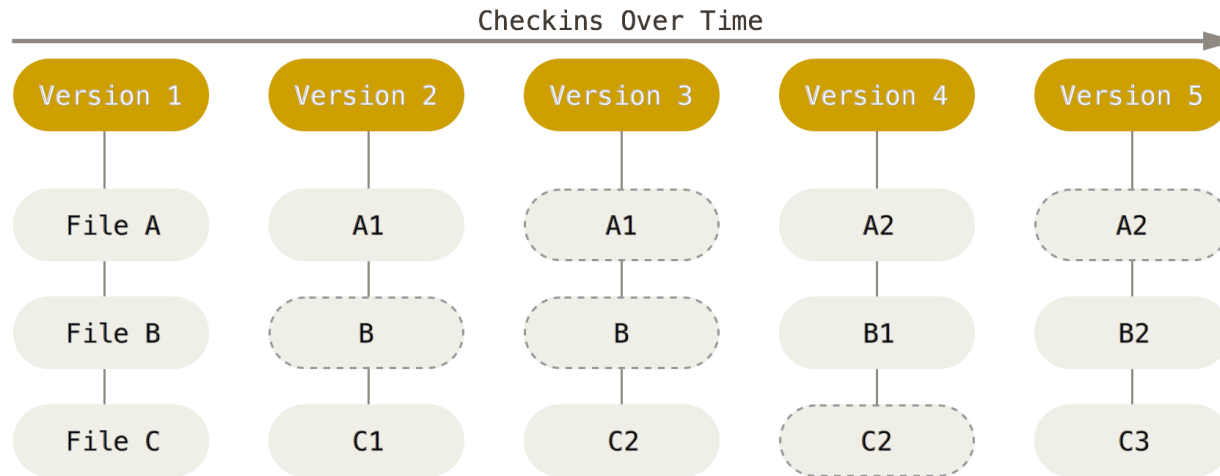
Distributed VCS

- Clients fully mirror the repository.
- If any server fails, any of the client repositories can be copied back to restore it.
- Every clone is actually a full backup of all the data.

– <http://git-scm.com/book/en/v2/>

Examples: Git, Bazaar, Mercurial, BitKeeper

Example

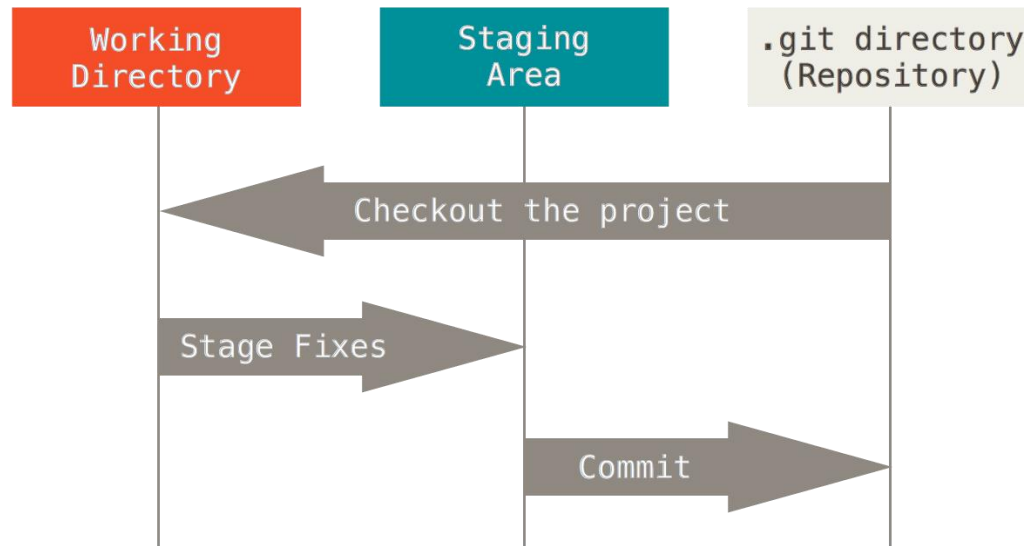


Git as a distributed version control system

- Data is stored as a “stream of snapshots”
- Each commit makes a snapshot of all files in the current state
- If a file has not changed, a reference to the previous identical file is used
- Checksums ensure integrity, implemented at the lowest system level

– <http://git-scm.com/book/en/v2/>

Example



Git as a distributed version control system

- Git directory: storage for metadata and objects
- Working directory: single check-out of one version of the project
- Staging area: storage for information about what will go into the next commit

– <http://git-scm.com/book/en/v2/>

Versions of System Artifacts

- Files and directories
- Source code, documentations, configuration files, etc.

Difference Data

- Differences between different versions of an artifact (often implicit, e.g., git)

Metadata about Software Change

- List of changed files
- Date of commits – “when”
- Author data (name, email, etc.) – “who”
- Description of subject – “why”
- Successor(s) and predecessor(s) forming a list or directed acyclic graph

Parsing Repository Data

Repository and revision traversal allow for gathering information.

Example process:

- Find path through commit graph
- Iterate over revisions and files
- For each revision: analyze author, date, message, ...
- For each file: analyze and extract metric values

```
commit eab31f228fd1872753c55fb438e3d88e51c0e9b2 d937b74b8f2f46e4943535d371e3f333ddf491a4
Author: Camilla Berglund <elmindreda@elmindreda.org>
Date: Tue Dec 31 20:06:18 2013 +0100
```

Added description of 3.0.4 release.

```
commit d937b74b8f2f46e4943535d371e3f333ddf491a4 0e8dace18c9d592a6f37a5238137edd1a6d41f53
Author: Camilla Berglund <elmindreda@elmindreda.org>
Date: Sat Dec 28 20:56:57 2013 +0100
```

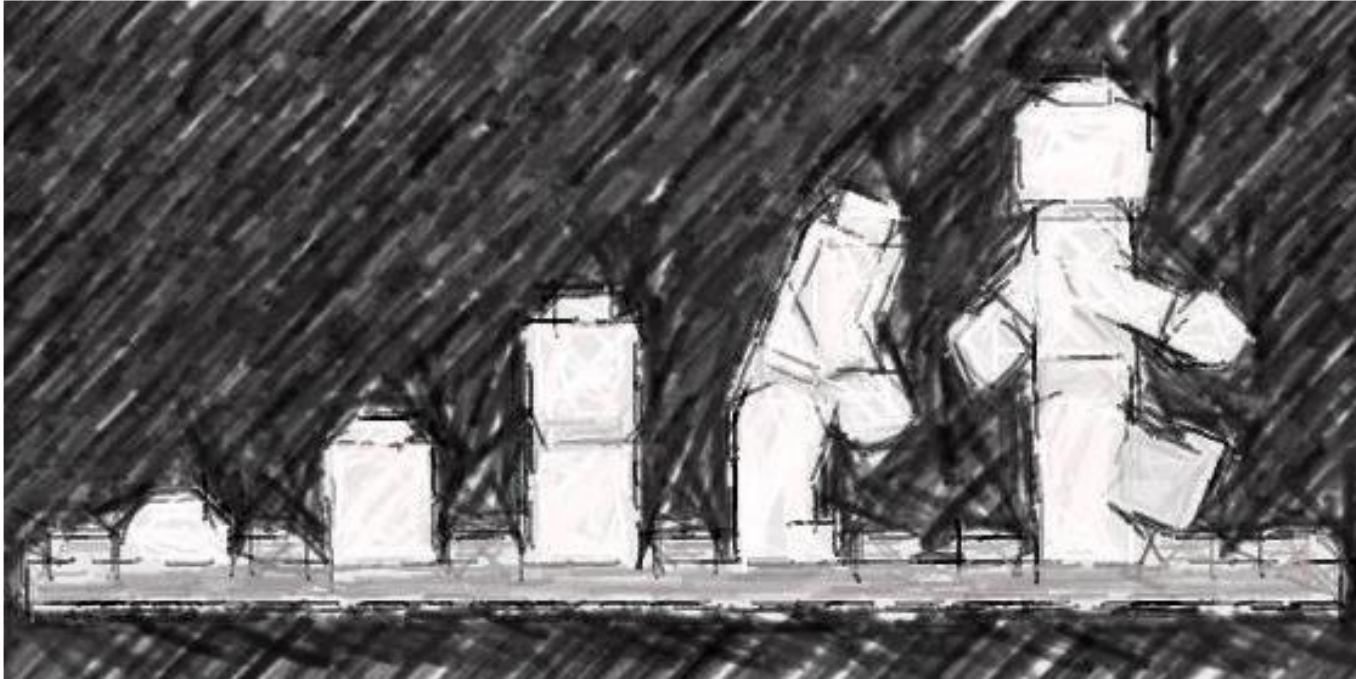
Fixed glfwSetWindowSize stealing focus.

Version control systems are ubiquitous tools to manage the development of large-scale, long-term complex computer-based systems. This kind of software repositories represents not only the source code base but also a primary source of information about the system's evolution and developer activities over time.

The direct use of version control systems includes the **management of changes to the artifacts of the system** including recall and comparison of different versions, revision management, and documentation of changes.

Revision control systems also have an indirect use for software analytics: They possess **valuable information about the system's evolution** in terms of applied changes, branches, trunks as well as the individual developer activities. Mining techniques for software repositories allow software analytics to take advantage of that information.

Version Control Systems



Principles of Version Control Systems
Example
Repository Data

Version Control System

A *version control system* manages changes applied over time to files such as source code, documents, models, web sites and any other related representation of system components. It can be based on a centralized database or distributed databases.

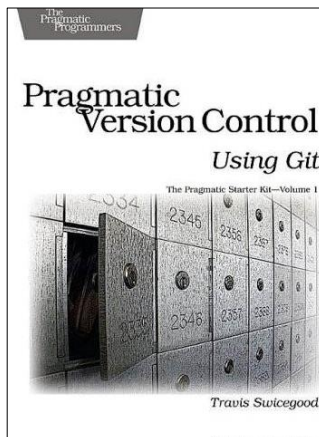
Commit

A *commit* is a conformed, applied change to one or more system artifacts, recorded by the version control system along with metadata.

Further Reading



Marco Biazzini, Martin Monperrus, Benoit Baudry: "On Analyzing the Topology of Commit Histories in Decentralized Version Control Systems", ICSME 2014, 2014.



Travis Swicegood: "Pragmatic Version Control Using Git (Pragmatic Starter Kit)", The Pragmatic Programmers, 2009.

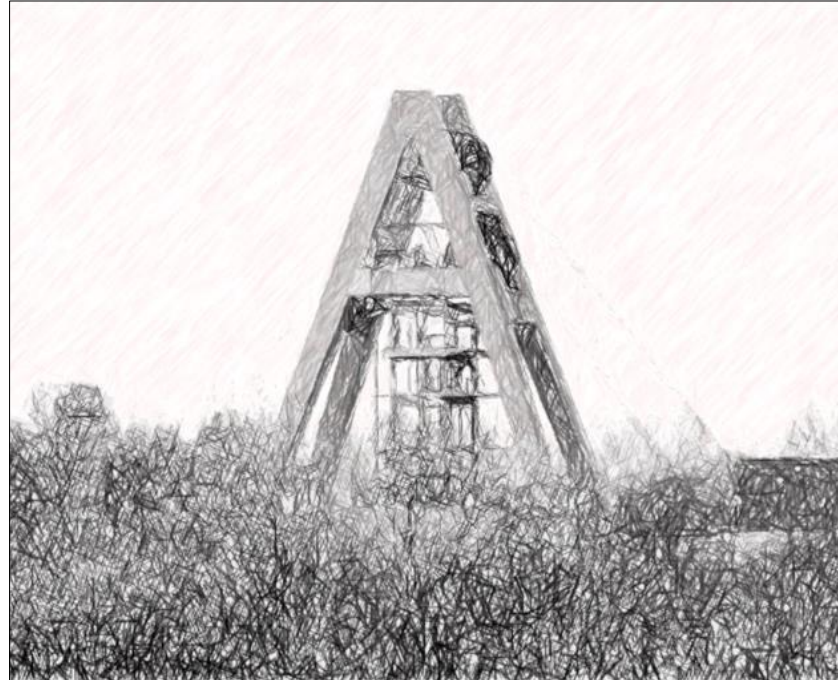
Regardless of the team size (even in the case of a single developer) version control systems (VCS) are essential elements in software development processes. They precisely store the history of system evolution in terms of committed changes and allow for efficient management of development activities.

VCS data tends to be "big software engineering data" throughout the lifecycle of a complex computer-based system.

Software analytics processes and transforms VCS data to elicit information about software evolution, developer involvement, and change histories, that is, it fundamentally creates added values out of VCS.

Software analytics seamlessly integrates VCS information with other system-related information such as measures of software metrics. This way, it offers insights over the system evolution.

Mining Software Repositories



Types of Software Repositories
Mining of Software Repositories
Mining Objectives

Types of Software Repositories

Software Repositories

A *software repository* represents a record-keeping database that stores data about artifacts of a complex computer-based system. It tracks changes applied to the artifacts and stores corresponding meta data.

Types of Software Repositories

History-Recording Software Repositories

Version control systems, bug/defect/faults repositories and tracking systems, communication archives (email, wiki, ...)

```
Welcome to Git (version 1.7.11-preview20120710)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

C:\Users\User\Desktop
$ cd "C:\Projects\MUCSample.Web"

C:\Projects\MUCSample.Web
$ git init
Initialized empty Git repository in c:/Projects/MUCSample.Web/.git/

C:\Projects\MUCSample.Web (master)
$ git config --global user.email "username@example.com"

C:\Projects\MUCSample.Web (master)
$ git config --global user.name "your-user-name"

C:\Projects\MUCSample.Web (master)
$ git add .

C:\Projects\MUCSample.Web (master)
$ git commit -m "Initial commit"
```

Source: <http://ulita.ms.mff.cuni.cz/>

Types of Software Repositories

Run-Time Data Software Repositories

Deployment logs, trace data about execution behavior, application deployment data



Anonymous | [Login](#) | [Signup for a new account](#) 2015-03-04 05:16 EST Project: [All Projects] [Switch] [RSS]

[My View](#) | [View Issues](#) | [Change Log](#) | [Roadmap](#) [Issue #] [Jump]

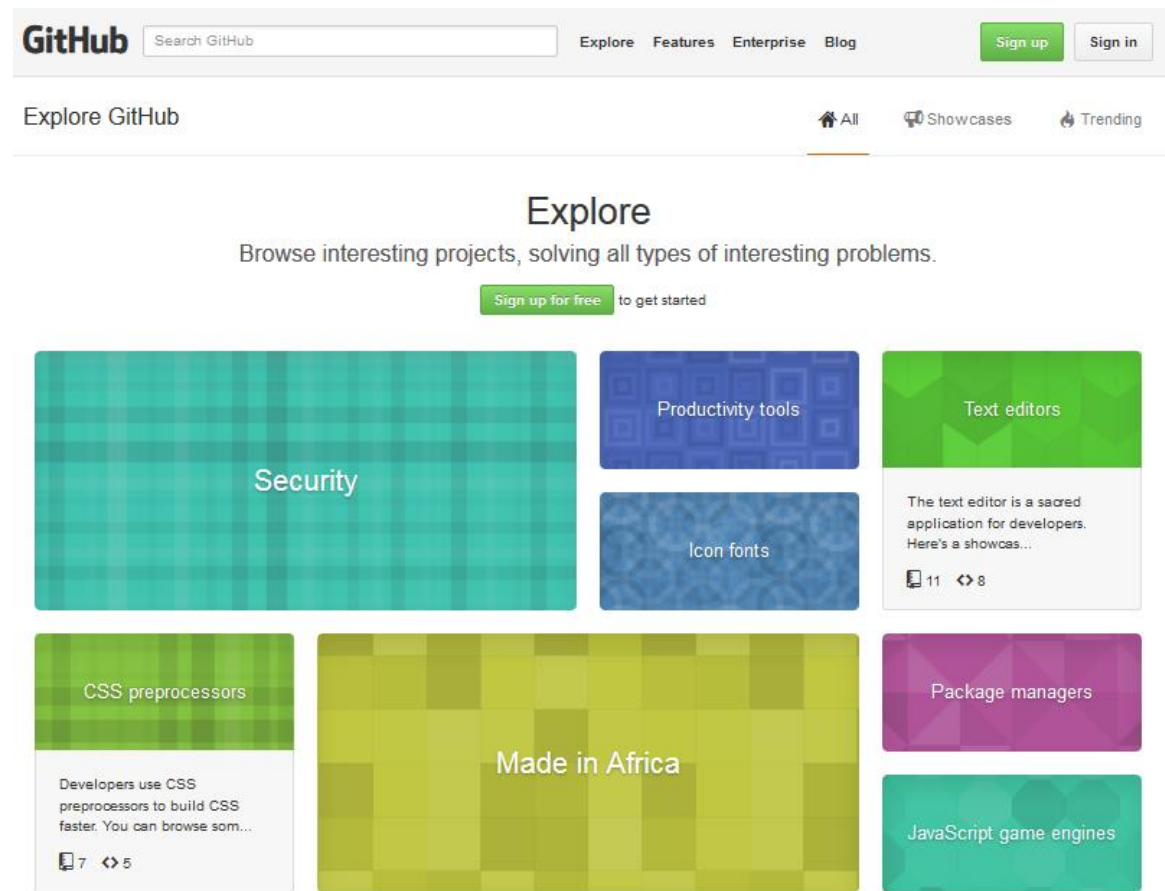
View Issue Details [[Jump to Notes](#)] [[Issue History](#)] [[Print](#)]

ID	Project	Category	View Status	Date Submitted	Last Update		
0001260	CMake	CMake	public	2004-10-18 10:05	2014-01-16 09:07		
Reporter	Bojan Resnik						
Assigned To							
Priority	normal	Severity	feature	Reproducibility	always		
Status	backlog	Resolution	open				
Platform		OS		OS Version			
Product Version							
Target Version		Fixed in Version					
Summary	0001260: Support for precompiled headers						
Description	Many popular C/C++ compilers, including VC++, Borland and GCC support creation and usage of precompiled headers. Currently, CMake does not directly support precompiled headers, so the only way to deal with them is to use SET_SOURCE_FILES_PROPERTIES with appropriate compiler-specific switches. This, however, requires conditional code for each supported compiler and has an additional drawback that it doesn't map well into some development environments. For instance, VC++ DSP/VCPROJ files have a special setting related to precompiled files, which is not affected by the current workaround. It would be a great benefit if CMake were to support precompiled headers in a generic and compiler-independent way, perhaps with special commands for creating and using precompiled headers on source file level, and not on target level. For compilers that do not support this feature, precompiled headers can be silently ignored, because a project that uses precompiled headers can always be compiled without them (usually slower).						
Tags	No tags attached.						
Attached Files	 PCHSupport.cmake [^] (7,798 bytes) 1969-12-31 19:00  NativePrecompiledheaders.support.zip [^] (4,813 bytes) 2007-08-17 11:46  PCHSupport_rodlima.cmake [^] (7,750 bytes) 2008-03-19 22:08  PCHSupport_26.cmake [^] (10,689 bytes) 2008-05-13 09:38  PCH_GCC4.cmake [^] (1,222 bytes) 2010-07-26 01:37						

Types of Software Repositories

Code Software Repositories

SourceForge, Google Code, or GitHub that contain large numbers of independently developed software projects

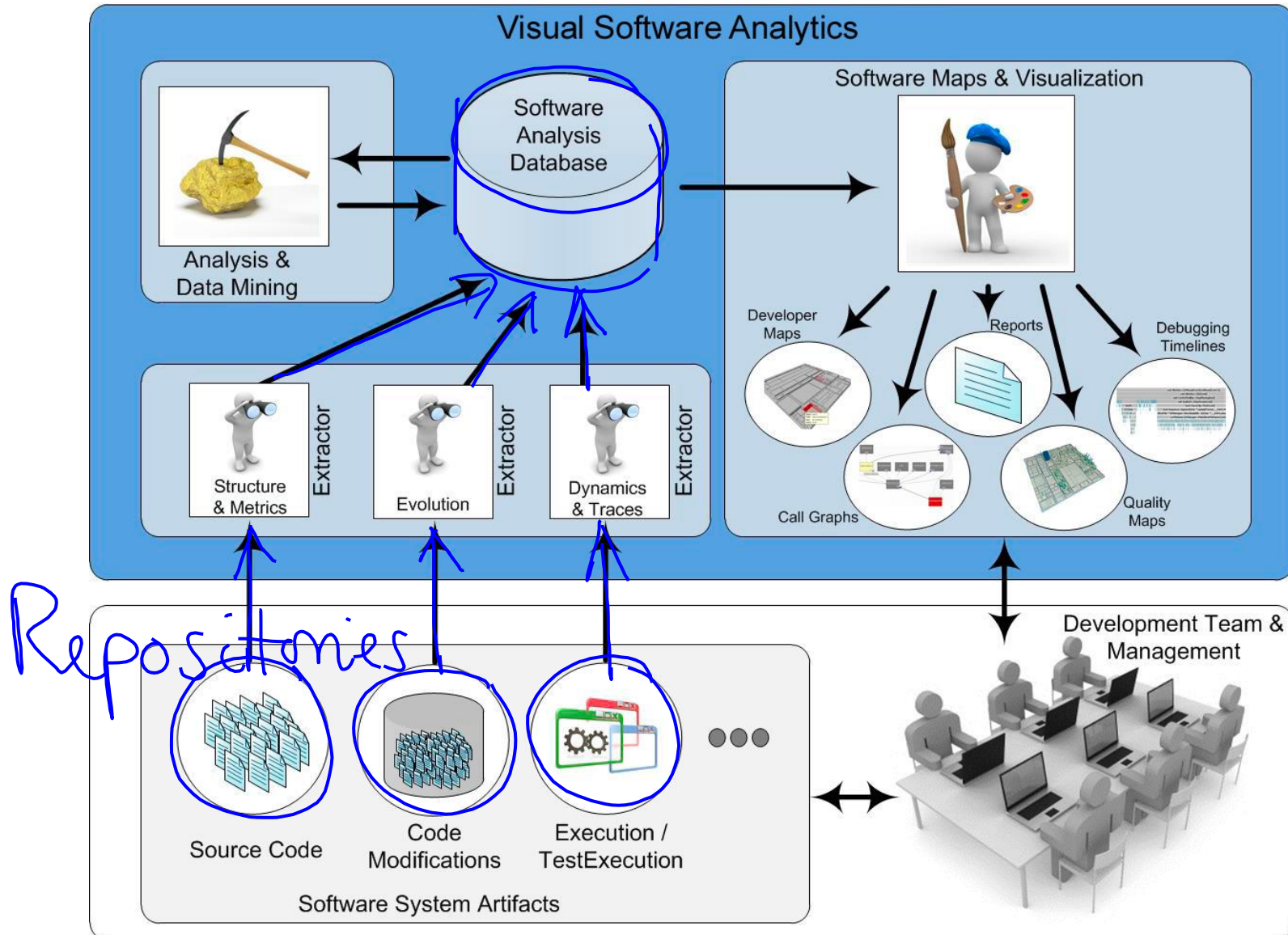


- The field of **Mining Software Repositories** aims at examining and analyzing *"the rich data available in software repositories to uncover interesting and actionable information about software systems and projects"*.
– J. Whitehead, T. Zimmermann: "Introduction to the Special Issue on Mining Software Repositories", Empir. SW Engineering 2012
- If focuses on uncovering pertinent information, relationships, and trends about a particular evolutionary characteristic of a system.
– Adapted from H. Kagdi, M. Collard, J. Maletic: "A survey and taxonomy of approaches for mining software repositories in the context of software evolution"

Role of software repositories

- software repositories contain a **wealth of valuable information** about software projects, that is, they are a first-class source of information
- software engineering becomes as a **more and more data-driven discipline**, lowering the dependency on intuition and experience of developers
- software analytics emerges as a key discipline that takes advantage of and applies **mining techniques** on software repositories

Mining of Software Repositories



Understanding Software Systems

- Software repositories server as a kind of **group memory** for software development projects.
- Understanding and comprehension of complex computer-based systems is often hindered by rare or out-of-date documentations and by frequently changing developer teams.

Examples:

- Revealing details about the history of a specific artifact
- Uncovering rationales for the current state or design of an artifact

Predicting and Identifying Bugs

- Software quality metrics (e.g., software complexity metrics) can be used predictors for buggy or risky source code.
- In software repository mining, there is a different approach: Best predictors of bugs are assumed to be prior bugs and prior changes, that is, “code that has bugs in the past is likely to have bugs in the future and bugs are not likely to appear in unchanged code”.

– Adapted from: A. E. Hassan, "The Road Ahead for Mining Software Repositories", FoSM, 48-57, 2008

Propagating Changes by Mining Co-Changes

- If source code is changed in one module, that change most likely impacts other modules, too (e.g., due to changed assumptions and conditions for module use). Hence, the change has to be propagated to these modules to ensure the system's integrity and consistency.
- Change propagation, however, still relies to a significant degree on empirical knowledge of experienced developers and communication. Dependency graphs provide a formal mean to identify modules for change propagation but they are typically incomplete.
- In software repository mining it is assumed that modules co-changing in the past are likely to co-change in the future. Based on this assumption, we can identify this kind of “historical dependencies”.

– Adapted from: A. E. Hassan, "The Road Ahead for Mining Software Repositories", FoSM, 48-57, 2008

Understanding Team Dynamics

- Teams communicate through a number of systems such as email, mailing lists, instant messaging, blogs, etc. about different issues including
 - design alternatives and decisions
 - project policies
 - bug reports
 - code reviews
- The communication information can be mined to understand the dynamics of the software development team and to take advantage of it for future tasks
- Example: Mining the most qualified team member for handling a bug issue by mining prior bug reports and historic communication information.

Limitations

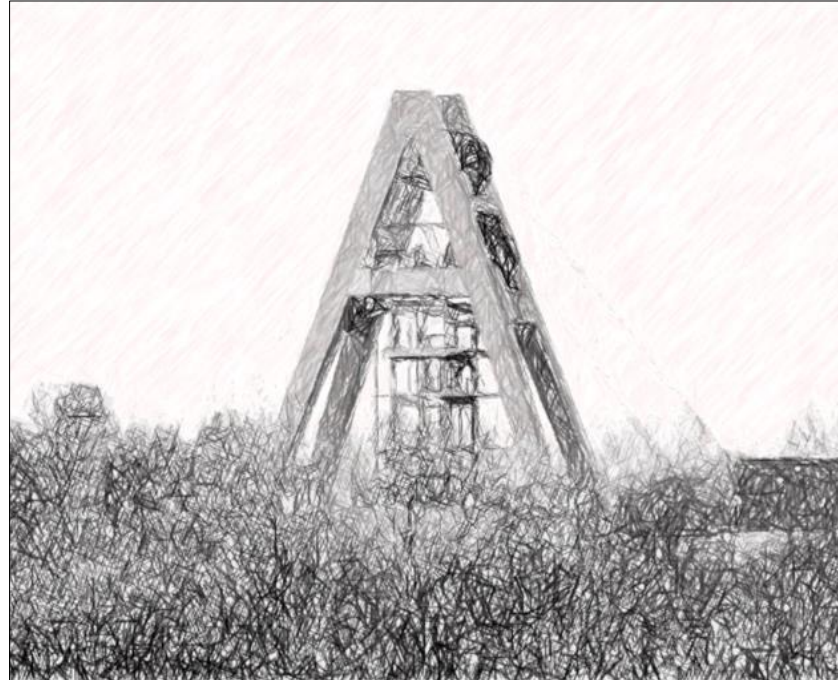
- Version control systems have not been design with automated mining in mind
- High complexity associated with extraction and analysis of data of software repositories
- No common or standardized format for repository records and operations
- Extraction of high-quality data is faced by partially unstructured, non-formal information in software repositories (e.g., emails, bug descriptions)
- Linking across different types of repositories is rarely used
- Quality of data stored in software repositories has to be improved and assured
- “Repository data cannot be used to conclude causation instead it can only show correlation.”

– Ahmed E. Hassan: “The Road Ahead for Mining Software Repositories”, FoSM, 48-57, 2008

Software repositories support different kinds of software maintenance tasks. They store and archive a **wealth of information** about a software project over time. They can store not only software-related artifacts, applied changes, and related meta data but also data about run-time execution, test executions, bug handling, developer communication and their activities.

Mining software repositories is an essential task in software analytics as it reveals valuable information about evolution, dynamics, and management of large-scale, long-term software development projects. In particular, it provides substantial information for system understanding and comprehension, identifies potentially buggy or risky system components, assists in managing change propagation, and reveals insights into team dynamics.

Mining Software Repositories



Types of Software Repositories
Mining of Software Repositories
Mining Objectives

Software Repository

A *software repository* represents a record-keeping database that stores data and metadata about artifacts of a complex computer-based system and changes applied to them.

Mining Software Repositories

Mining software repositories denotes concepts, methods, techniques and tools that examine and analyze data of software repositories to uncover pertinent information, relationships, and trends about a particular evolutionary characteristic of a system.

Further Reading

The Road Ahead for Mining Software Repositories

Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University, Canada
ahmed@cs.queensu.ca

Abstract

Source control systems, like repositories, archived communications, deployment logs, and code repositories are examples of software repositories that are commonly available for most software projects. The Mining Software Repositories (MSR) field analyzes and mines this rich data available in these repositories to answer interesting and actionable information about software systems. By transforming these repositories from data record keeping into data analysis, we can gain decision processes in modern software projects. For example, data on previous commit dependencies, traditionally used to analyze code, could be mined with data in bug repositories to help predict the probability of a bug being introduced. We have seen about 1000 code fixes in prior changes and logs. In this paper, we present a brief history of the MSR field and discuss several recent achievements and trends of using MSR techniques to support software research and practice. We also discuss the current opportunities and challenges that lie in the road ahead for this important and emerging field.

1 Introduction

Prior experiences and dominant patterns are the driving force for most decision processes in modern software development. Practitioners often rely on their experiences, intuition, and gut feeling to making important decisions. Managers often drive development and testing processes based on their experience in previous projects and their intuition about the complexity of the new project relative to prior projects. Developers commonly use their experience when deciding a new feature or fixing a bug. Modern history provides the testing of features that are known to be error prone based on field and log inputs. Software repositories contain a wealth of valuable information about software projects. Using the information stored in these repositories, practitioners can improve how they manage and reproduce, and depend more on historical and field data.

The Mining Software Repositories (MSR) field analyzes and mines this rich data available in these repositories to answer interesting and actionable information about software systems and projects. Examples of software repositories are:

Historical repositories such as version control systems, bug repositories, and archived communications record several information about the evolution and progress of a project.

Real-time repositories such as deployment logs contain information about the execution and the usage of an application at a single or multiple deployment times.

Code repositories such as SourceForge.net and Google code contain the source code of various applications developed by various developers.

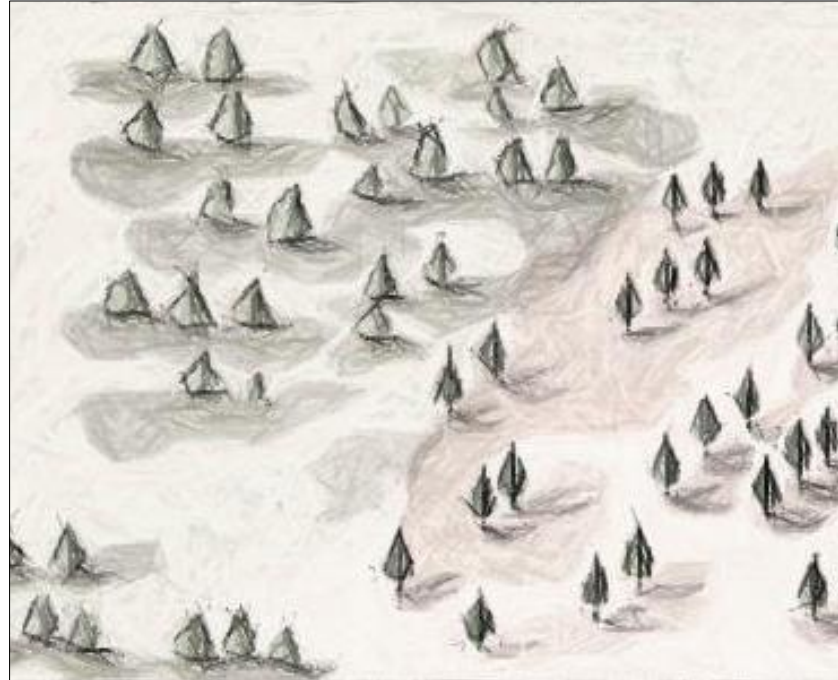
Software repositories are commonly used in practice to record keeping repositories and are rarely used to support decision making processes. For example, historical repositories are used to track the history of a bug or a feature, but are not commonly used to determine the expected maintenance time of an application based on the execution time of previously closed bugs.

MSR researchers aim to transform these repositories from data record keeping into data analysis processes which can gain decision processes in modern software projects. Mining these historical, real-time and code repositories, we can answer useful and important patterns and information. Historical repositories capture important historical dependencies [1] between project artifacts, such as functions, documentation files, or configuration files. The subject can use the dependencies to anticipate changes to related artifacts. Instead of only using static or dynamic code repositories which are just to capture information dependencies, for example, a change to the code which refers data in a file may require changes to the code which needs data from the file. Although these data are not contained dependencies (e.g., data that control flow between both pieces of

Ahmed E. Hassan: "The Road Ahead for Mining Software Repositories",
Frontiers of Software Maintenance (FoSM), 48-57, 2008.

- The wealth and benefits of information stored commonly in software repositories are obvious. A key limitation results from the fact that these software repository systems initially have not been designed to become parts of software analytics systems or serve data-driven software engineering. Often they operate in isolation. That is, it is up to specialized tool solutions to close the “tool gap” in the future.
- Mining software repositories provides valuable information to software analytics – from developer-centric, evolution, and history perspectives.
- Advanced mining techniques such as for change propagation automate and implement what would be otherwise only provided by the experience of individual developers. Consequently, mining complements software analytics by data-driven insights about project history and team dynamics.

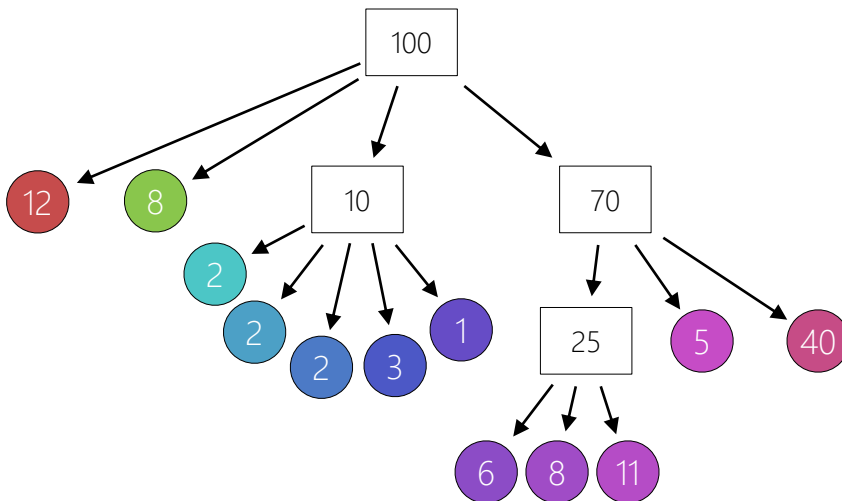
Treemaps



Principles of Treemaps
Treemap Layout Algorithms
Applications of Treemaps

Principles of Treemaps

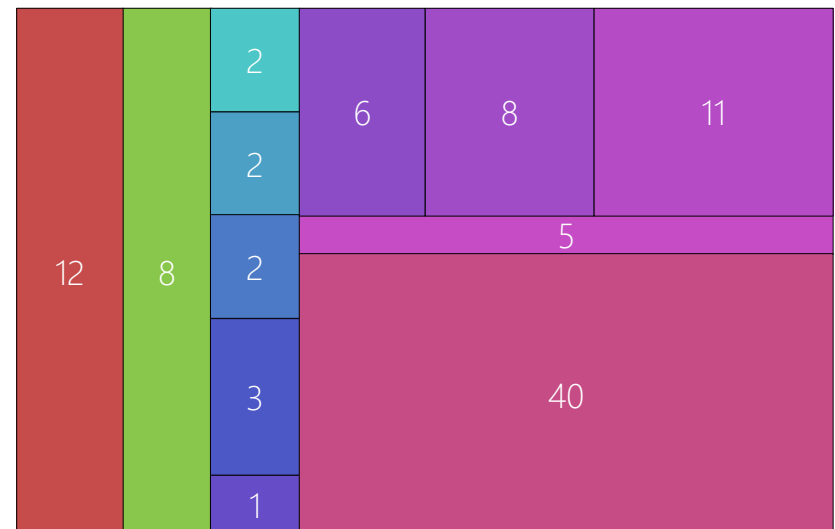
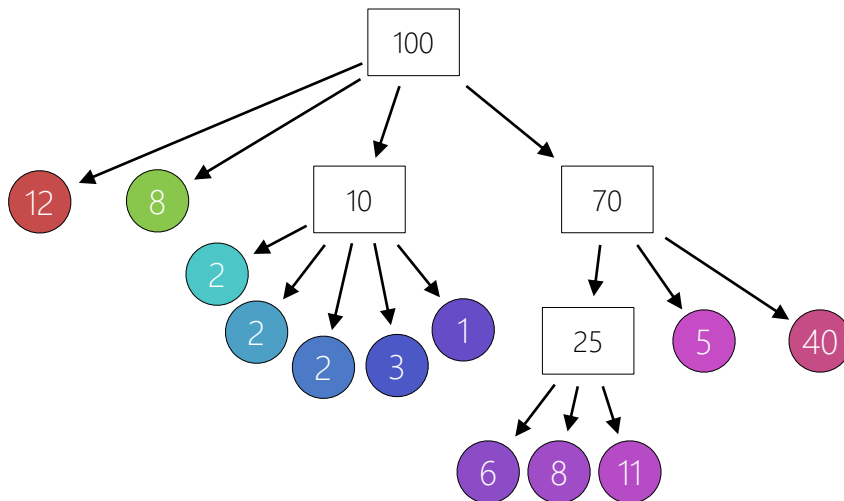
Depicting tree-based data using graphs: Leaf nodes are colored and have numerical weights, which can represent any associated attribute, e.g., KLOC. The weight of a parent node is defined by the sum of the weights of its child nodes.



Principles of Treemaps

Depicting tree-based data using graphs: Leaf nodes are colored and have numerical weights, which can represent any associated attribute, e.g., KLOC. The weight of a parent node is defined by the sum of the weights of its child nodes.

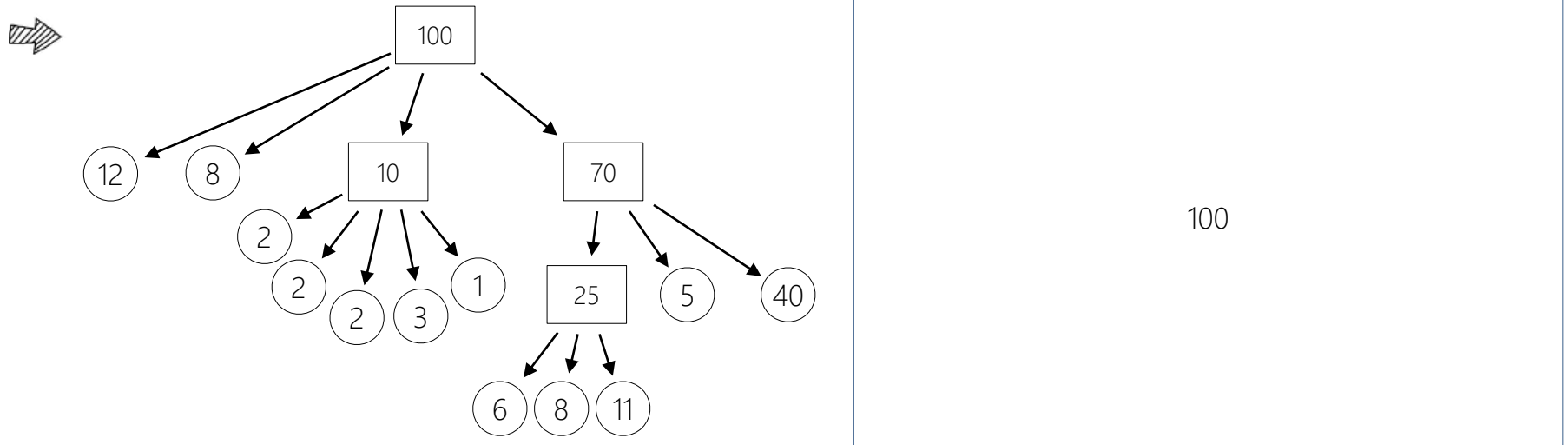
Treemap (right): The root node corresponds the outer rectangle. Child nodes are represented by nested rectangles; the rectangles' size is proportional to the their weights.



Definition

- A *treemap* defines a visual mapping of a tree hierarchy to space-filling nested rectangles; the size of the rectangles are proportional to numerical weights associated with tree nodes.
- **Mapping:** Each node of given tree hierarchy is represented by a *rectangle* of the treemap. Rectangles of child nodes are contained in the rectangle of their corresponding parent node. The specific *treemap layout algorithm* determines the spatial subdivision.
- **Efficient Space Usage:** In contrast to graph-based approaches, the treemap efficiently uses the given rectangular space, that is, the hierarchy is completely mapped inside the given root rectangle.

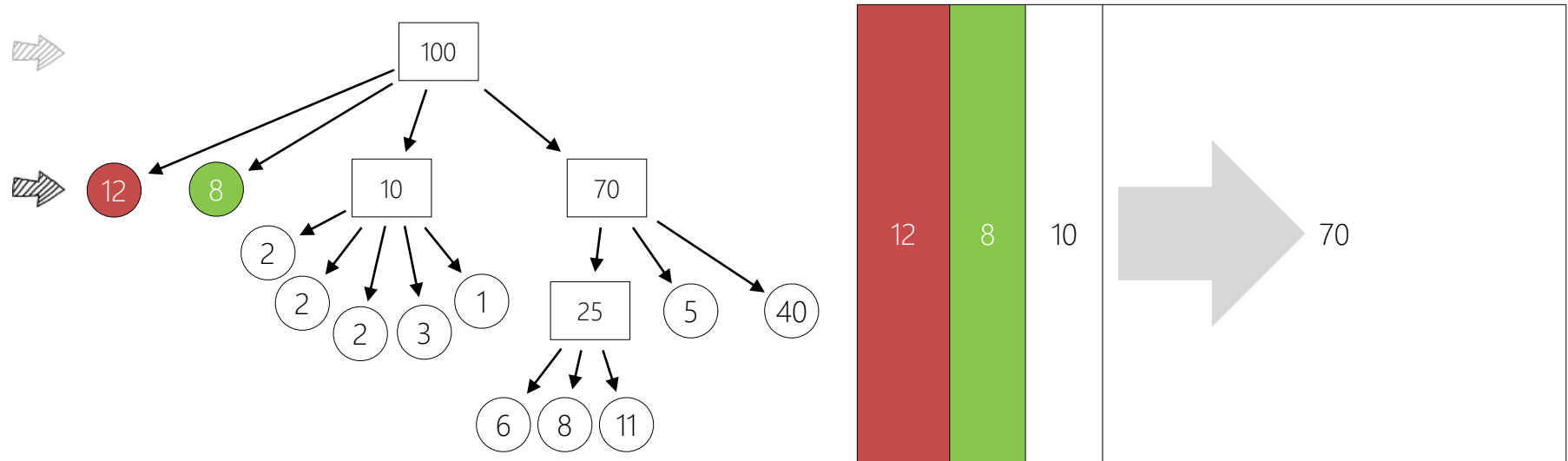
Treemap Layouts: Slice+Dice



“Splitting the screen into rectangles in alternating horizontal and vertical directions
as you traverse down the levels”

– Ben Shneiderman: “Treemaps for space-constrained visualization of hierarchies”

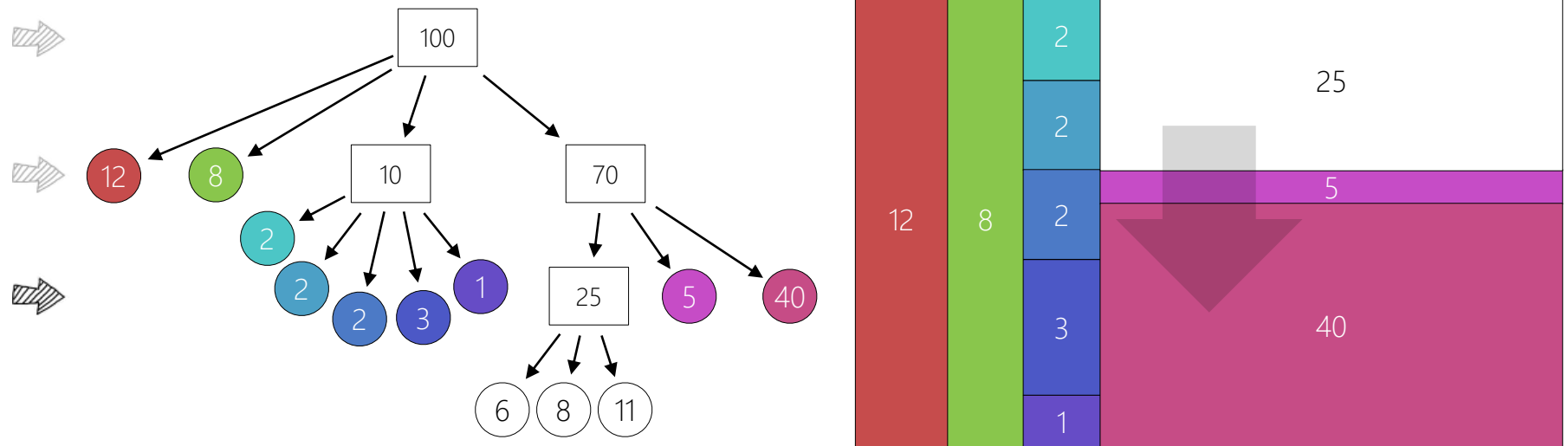
Treemap Layouts: Slice+Dice



“Splitting the screen into rectangles in alternating horizontal and vertical directions
as you traverse down the levels”

– Ben Shneiderman: “Treemaps for space-constrained visualization of hierarchies”

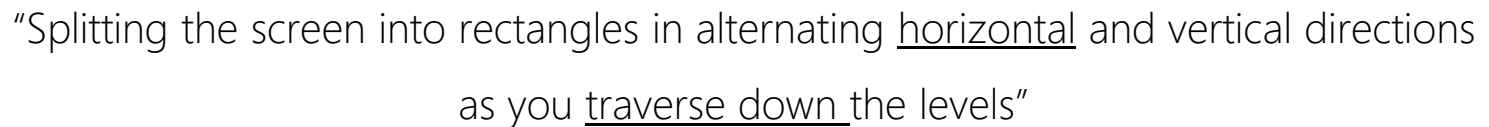
Treemap Layouts: Slice+Dice



“Splitting the screen into rectangles in alternating horizontal and vertical directions
as you traverse down the levels”

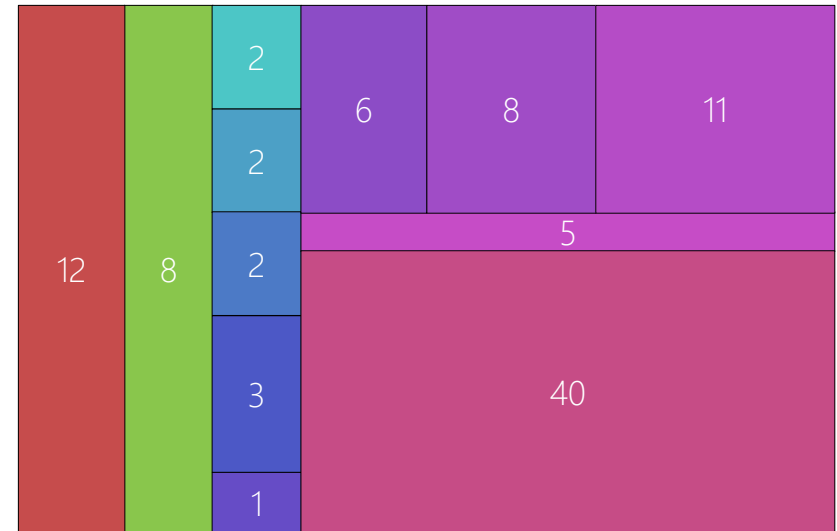
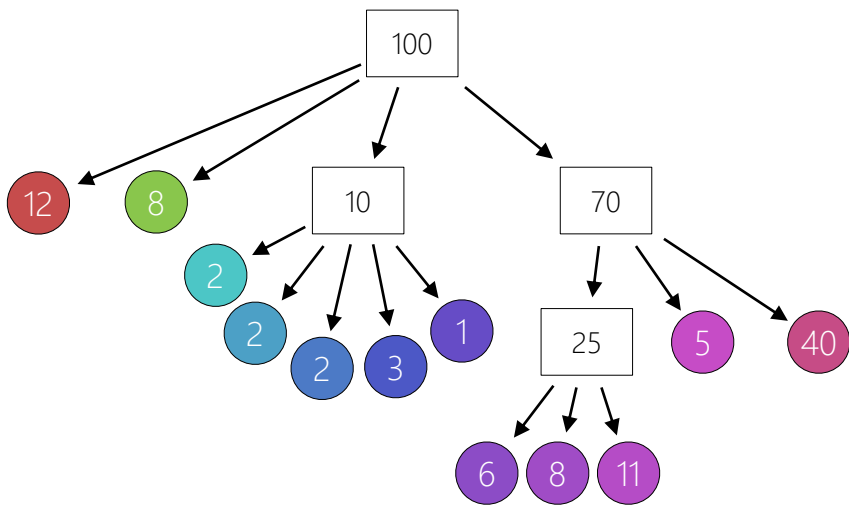
– Ben Shneiderman: “Treemaps for space-constrained visualization of hierarchies”

HPI Hasso Plattner Institut



OpenHPI - Automated Visual Software Analytics - Prof. Dr. Jürgen Döllner - 2015

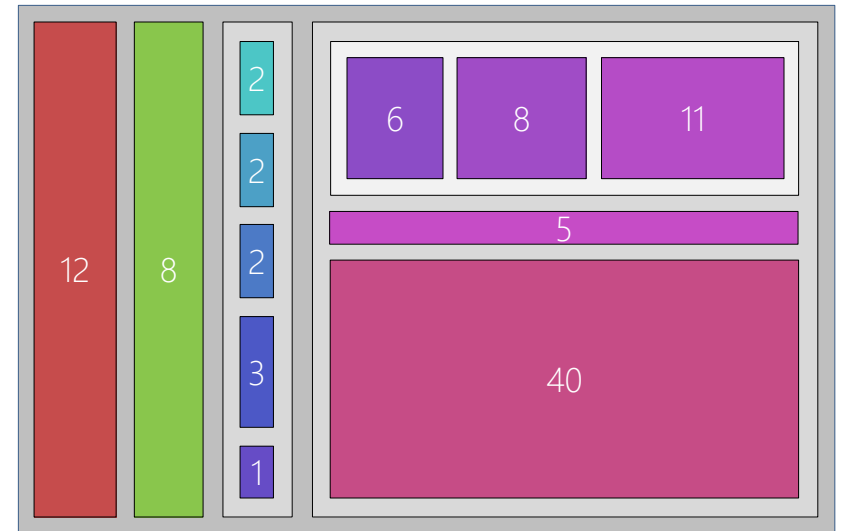
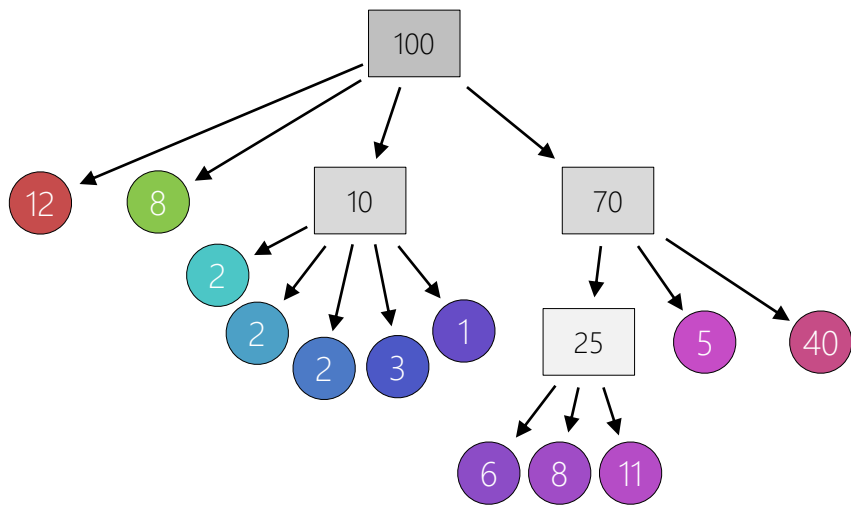
Treemap Layouts: Slice+Dice



“Splitting the screen into rectangles in alternating horizontal and vertical directions
as you traverse down the levels”

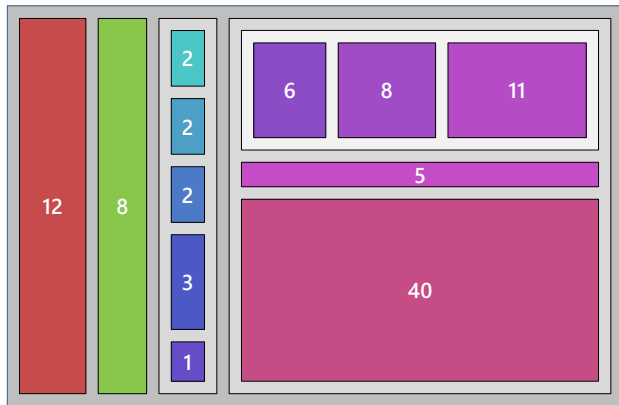
– Ben Shneiderman: “Treemaps for space-constrained visualization of hierarchies”

Treemap Layouts: Slice+Dice

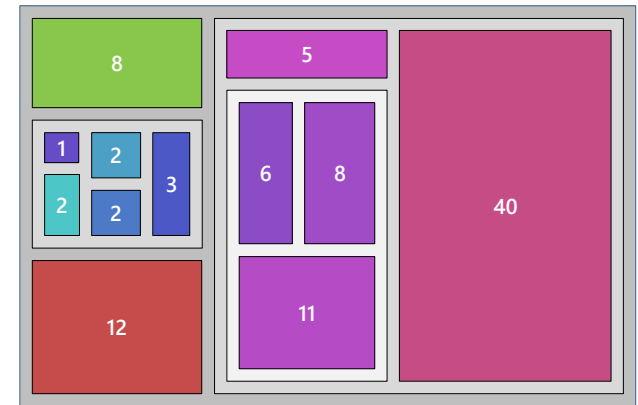


To ease recognition of the tree hierarchy, treemap layout algorithms can apply a **margin** (padding) to explicitly flag parent nodes and, thereby, depicting the nesting levels.

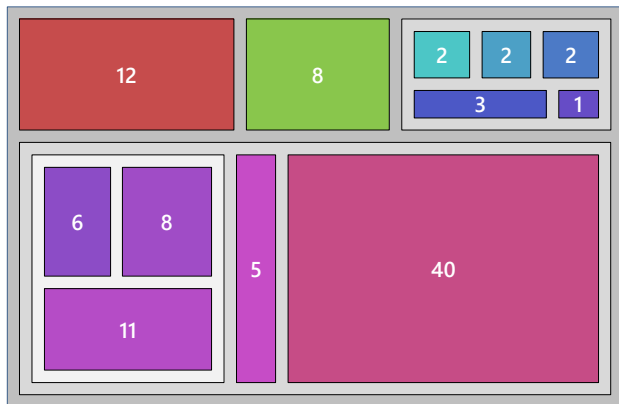
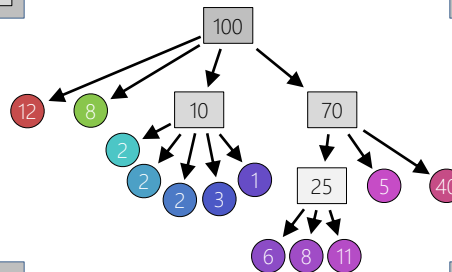
Treemap Layouts



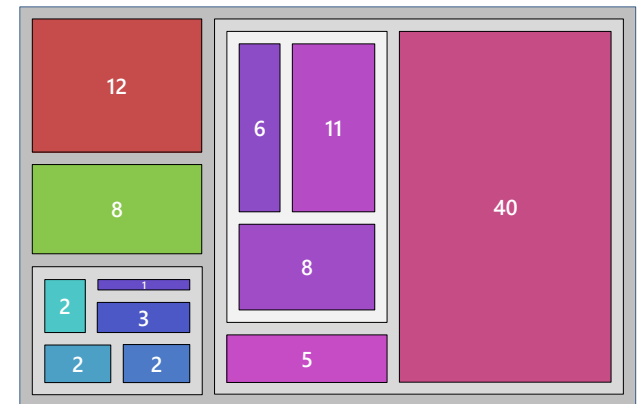
Slice+Dice



Squarified



Strip



Hilbert/Moore

Software engineering data is generally organized by hierarchies

- System hierarchy (system, subsystems, modules, classes, function, namespaces, etc.)
- Execution traces (root call, call graph, etc.)
- Development process (process, subprocesses, stages, steps, etc.)
- Development team (team leader, team groups, team members)

The system's implementation represents an evolving but relatively stable base hierarchy, which serves as a **reference structure for most of the software engineering data**. Hence, treemaps serve as a fundamental tool for visual software analytics.

Treemaps offer a number of specific **advantages and properties** for visualizing tree hierarchies:

- Space-efficient, space-filling, and robust
- Scale with respect to the hierarchy size; they can be used for “big hierarchies”
- Extensible to 3D (“cascaded tree maps”, “2.5D treemaps”)
- Mapping associated information onto a number of visual variables such as size, color, texture, height (3D)
- Support of detecting structures, patterns, and outliers with respect to mapped data.

Treemaps offer a number of specific **advantages and properties** for visualizing tree hierarchies:

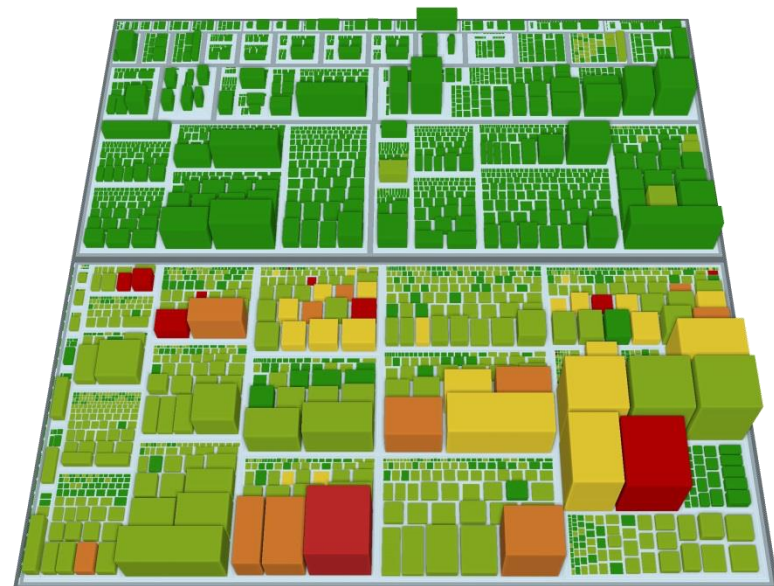
- Treemaps provide context and are sensitive to outliers
- “It moves the burden of reasoning partially to the visual system, freeing other parts of the cognitive system for higher-level tasks. This can lead to immediate insight in complex phenomena that otherwise would be hard to grasp. Also, visualization can provide context to search results and make outliers visible. Context and outliers are present but often not perceivable in tables; aggregation condenses information, leaving out context and outliers altogether.”

– R. Vliegen, J.J. van Wijk, E.-J. van der Linden: “Visualizing Business Data with Generalized Treemaps,
IEEE Transactions on Visualization and Computer Graphics, 12(5), 789-796, 2006.

- Treemaps make use of the **metaphor of a geographic map** – a common and familiar information visualization technique

Visual software analytics uses treemaps as one base mapping technique

- In visual software analytics the concept of “**software map**” has been developed based on treemaps, extended to 3D.
- It allows different stakeholders in software engineering to explore, analyze, and monitor different views on complex, big software engineering data.
- Example:

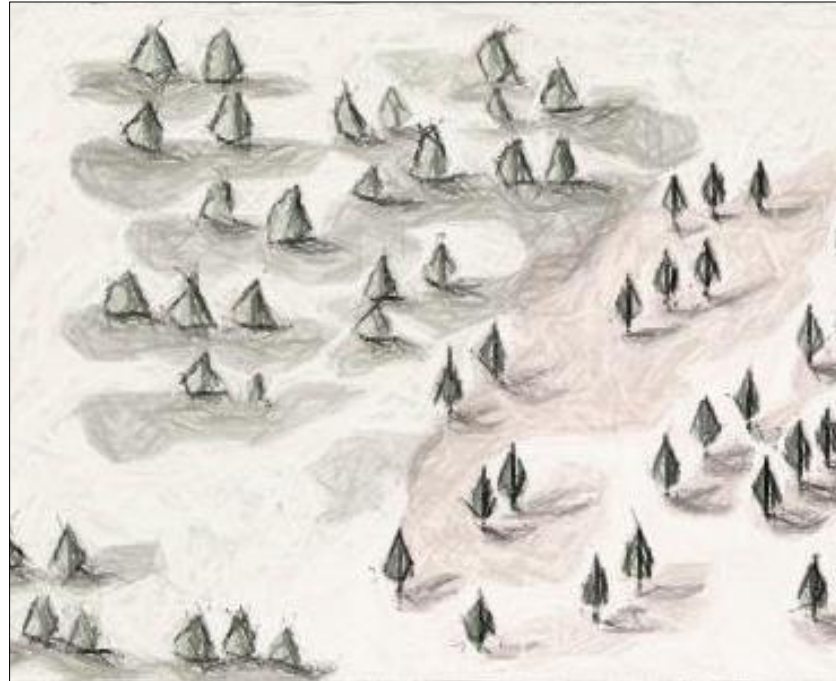


To map large hierarchical data, **treemaps provide a space-efficient, space-filling, and robust approach** for visualizing that data. Treemaps provide context, are sensitive to outliers, and facilitate detecting patterns in large tree hierarchies with respect to the mapped attributes.

The **treemap layout is easy to understand**: children are represented by nested rectangles inside the rectangle of their parent. Different layout algorithms exist that have different properties regarding proportions of rectangles and stability in the case of hierarchy changes.

In visual software analytics the **treemap-based software map** is used as one key instrument to explore, analyze, and monitor complex big software engineering data.

Treemaps



Principles of Treemaps
Treemap Layout Algorithms
Applications of Treemaps

Treemap

A *treemap* defines a visual mapping of a tree hierarchy to space-filling nested rectangles; the size of the rectangles are proportional to numerical weights associated with tree nodes.

Treemap Layout Algorithm

A *treemap layout algorithm* calculates geometry and position of the visual representations of nodes. Common treemap layout algorithms include "Slice+Dice", "Squarified", "Strip", and "Hilbert/Moore".

Further Reading



R. Vliegen, J.J. van Wijk, E.-J. van der Linden:

"Visualizing Business Data with Generalized Treemaps",

IEEE Transactions on Visualization and Computer Graphics, 12(5), 789-796, 2006.

- Treemaps and software engineering data are a good team: Hierarchical data is predominant, and treemaps are effective means to visualize them.
- Treemaps take advantage of several aspects: 1) They provide context and show all items, that is, they do not aggregate, generalize, or abstract items. 2) They can be efficiently processed by the human visual system. 3) They are close to geographic maps we are all familiar with.
- Technically, treemap rendering implementations allow for visualizing virtually unlimited hierarchies in real-time, that is, they are efficient elements in software analytics user interfaces.

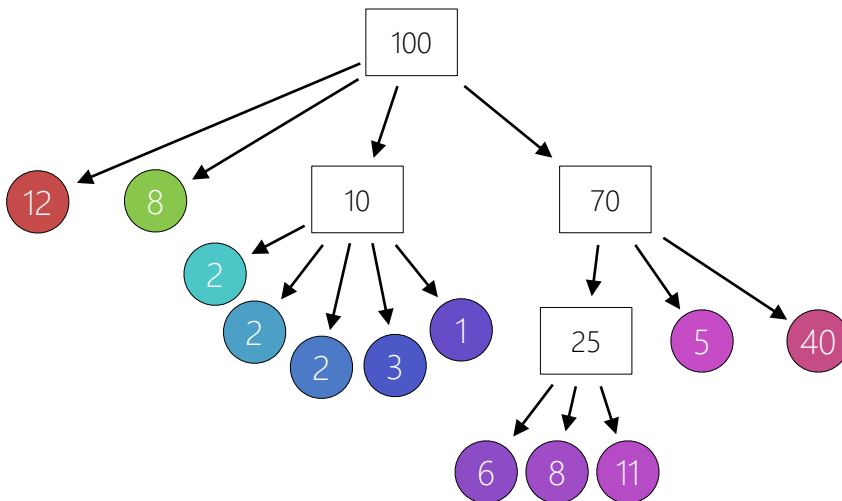
2.5D Treemaps



Principles of 2.5D Treemaps
Visual Variables of 2.5D Treemaps
2.5D Treemap Example

Principles of 2.5D Treemaps

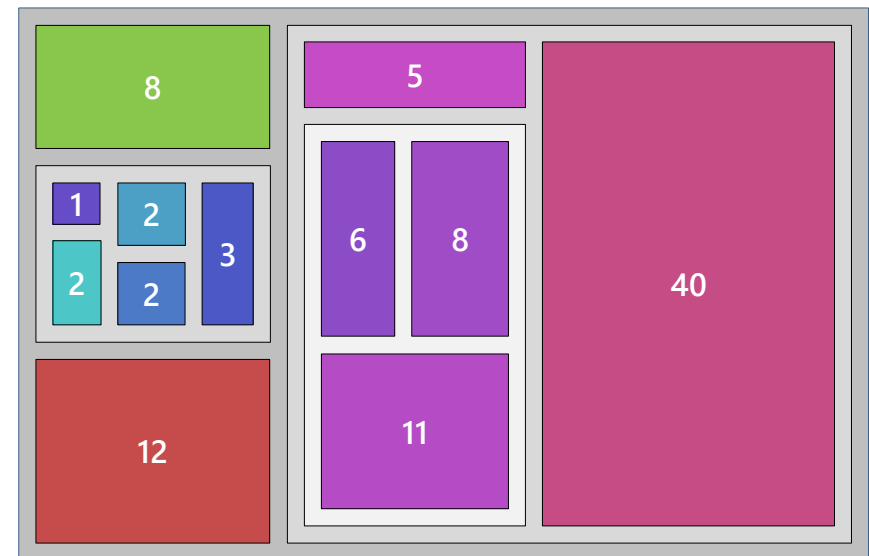
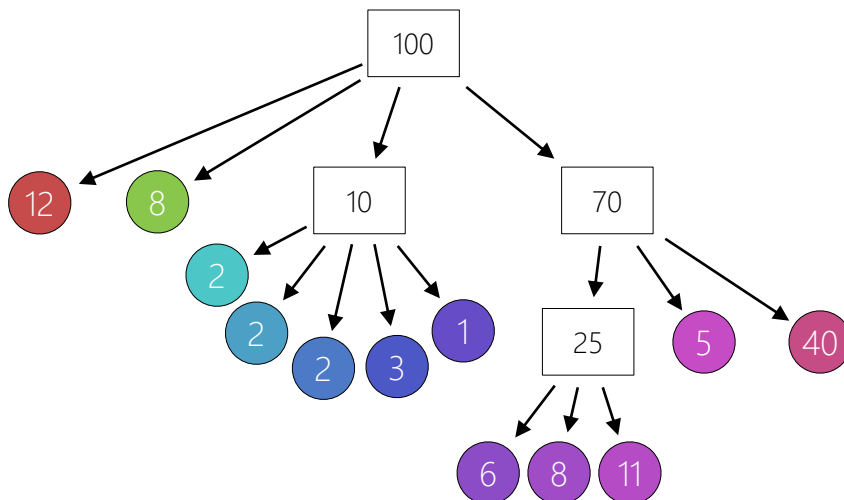
Depicting tree-based data using graphs: Leaf nodes are colored and have numerical weights, which can represent any associated attribute, e.g., KLOC. The weight of a parent node is defined by the sum of the weights of its child nodes.



Principles of 2.5D Treemaps

Depicting tree-based data using graphs: Leaf nodes are colored and have numerical weights, which can represent any associated attribute, e.g., KLOC. The weight of a parent node is defined by the sum of the weights of its child nodes.

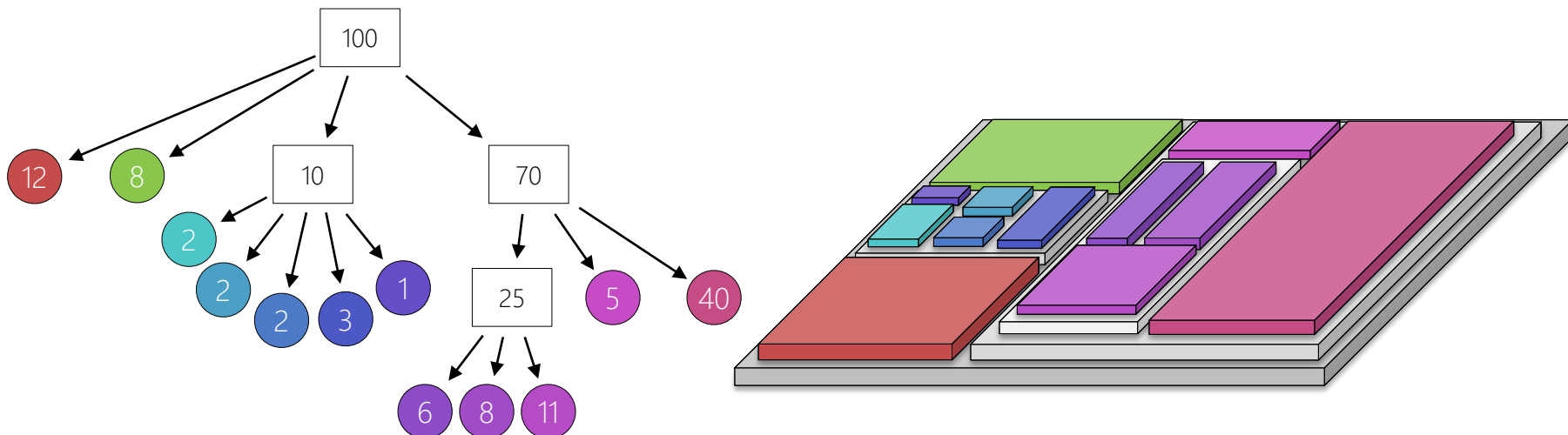
Treemap (right): The root node corresponds the outer rectangle. Child nodes are represented by nested rectangles; the rectangles' size is proportional to the their weights (using a squarified treemap layout).



Principles of 2.5D Treemaps

Depicting tree-based data using graphs: Leaf nodes are colored and have numerical weights, which can represent any associated attribute, e.g., KLOC. The weight of a parent node is defined by the sum of the weights of its child nodes.

2.5D Treemap (right): The visual representation uses cuboids (3D blocks) to represent nodes (using a squarified treemap layout).



Definition

- A 2.5D treemap defines a visual mapping of a tree hierarchy to cuboids. The ground areas of cuboids are nested on a root ground rectangle in a space-filling way; the ground area size of cuboids is proportional to numerical weights associated with tree nodes.
- **Mapping:** Each node of a given tree hierarchy is represented by a cuboid of the 2.5D treemap. The ground area of cuboids of child nodes is contained in the ground area of their corresponding parent node. The specific treemap layout algorithm determines the spatial subdivision.
- **Efficient Space Usage:** In contrast to graph-based approaches, 2.5D treemaps efficiently use the given ground area and extend the original treemaps in 3D space.

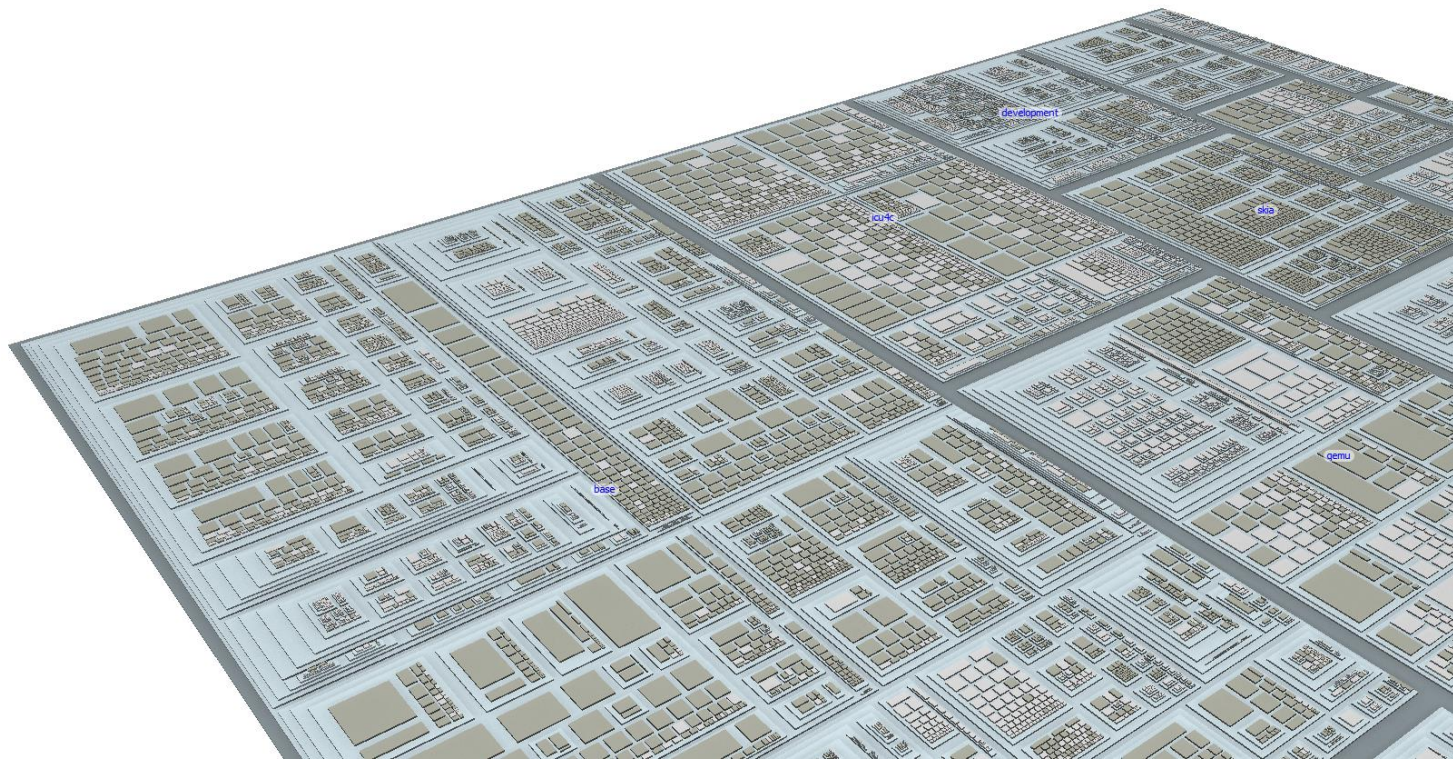
– This type of treemap is called “2.5D” (two-and-a-half dimensional) in analogy to heightfields, which can be defined by a 2D map and an additional height, leading to a “2.5D” surface representation. 2.5D treemaps are also known as Step Trees.

Visual Variables

- Parameters of the visualization technique that can be used to map thematic information.
 - Rectangles of 2D treemaps: size, color
 - Cuboids of 2.5D treemaps: size, color, height
- 2.5D treemaps offer additional partially usable parameters such as
 - cuboid geometry (sharp or rounded corners etc.)
 - texture of side and top faces of cuboids.

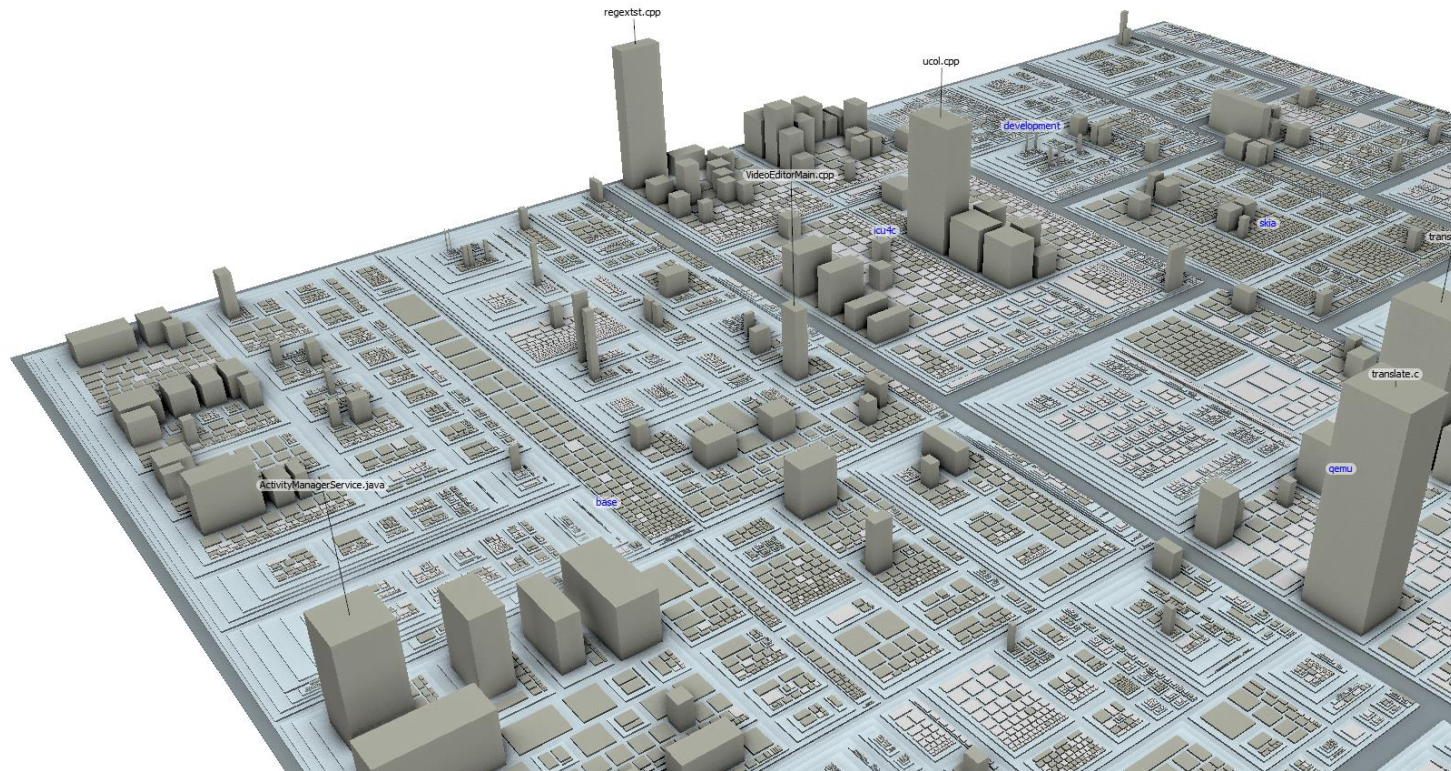
2.5D Treemap Example

Size: The basic treemap layout algorithm maps a given weight to the (footprint) size of each rectangle.



2.5D Treemap Example

Height: An additional data attribute is mapped as height assigned to the corresponding cuboid.



2.5D Treemap Example

Color: Similar to 2D treemaps, data attributes can be mapped to cuboid color.



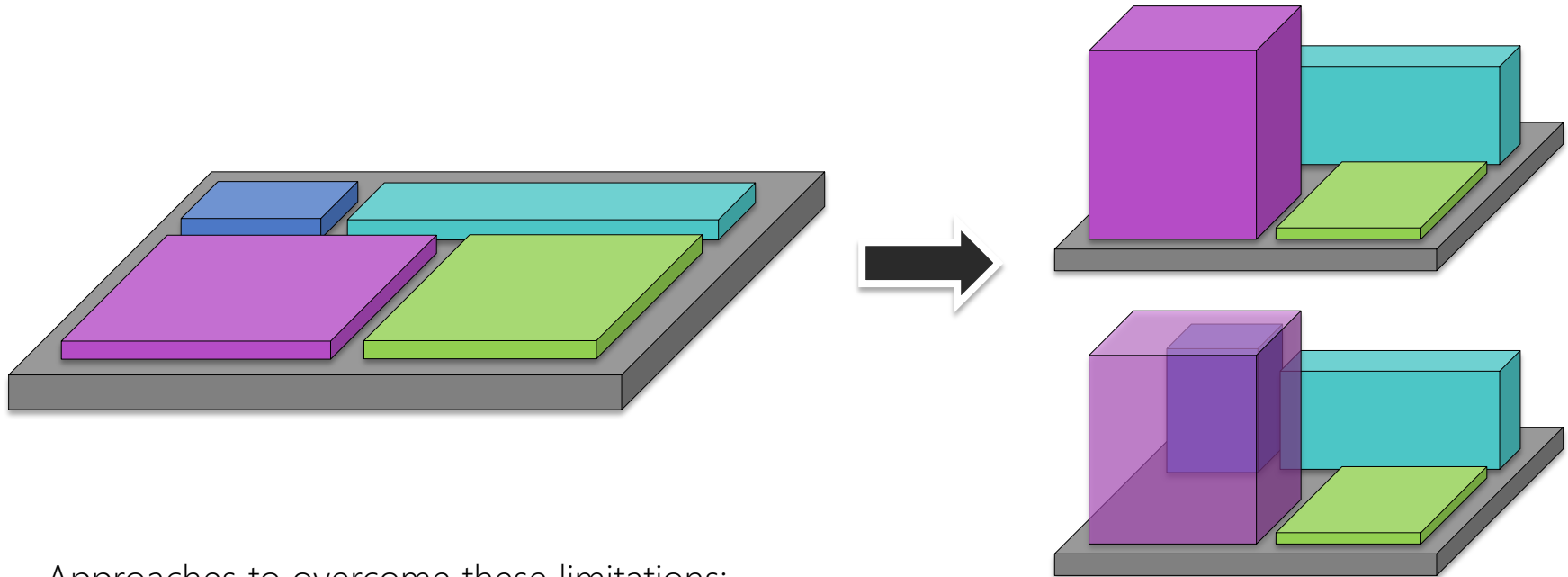
2.5D Treemap Example

Texture: Additional attributes can be mapped to cuboid's top-face and side-faces.



Limitations

2.5D treemap cuboids can occlude parts of the treemap:



Approaches to overcome these limitations:

- Interactive 3D exploration, that is, changing the viewpoint
- Automatically using transparency to cuboids that are occluders

2.5D treemaps extend treemaps towards a 3D representation. As a key advantage, they offer **cuboids** that allow for mapping an additional data attribute to the cuboid height. To depict a 2.5D treemap, an orthogonal view is typically used to avoid perspective distortion.

2.5D treemaps possess **three strong visual variables**: size, color, and height. Software analytics can take advantage of 2.5D treemaps for visualizing (big) tree data because color and height serve as two independent “information carriers” while a fixed attribute such as LOC is typically mapped to size.

Technically, 2.5D treemaps benefit from today's 3D capabilities of devices. They can be implemented in an efficient way using **GPU functionality**. For that reason, they can even be used to visualize “big tree data”, that is, 2.5D treemap rendering is scalable.

2.5D Treemaps



Principles of 2.5D Treemaps
Visual Variables of 2.5D Treemaps
2.5D Treemap Example

2.5D Treemap

A 2.5D treemap defines a visual mapping of a tree hierarchy to cuboids. The ground areas of cuboids are nested on a root ground rectangle in a space-filling way. The ground area size of cuboids is proportional to numerical weights associated with tree nodes, calculated by a treemap layout algorithm. Color and heights are two further and independent visual variables that can be used to map data attributes associated with tree nodes.

Further Reading



H.-J. Schulz, S. Hadlak, H. Schumann: "*The Design Space of Implicit Hierarchy Visualization: A Survey*", IEEE Transactions on Visualization and Computer Graphics, 04(07):393-411, 2011.

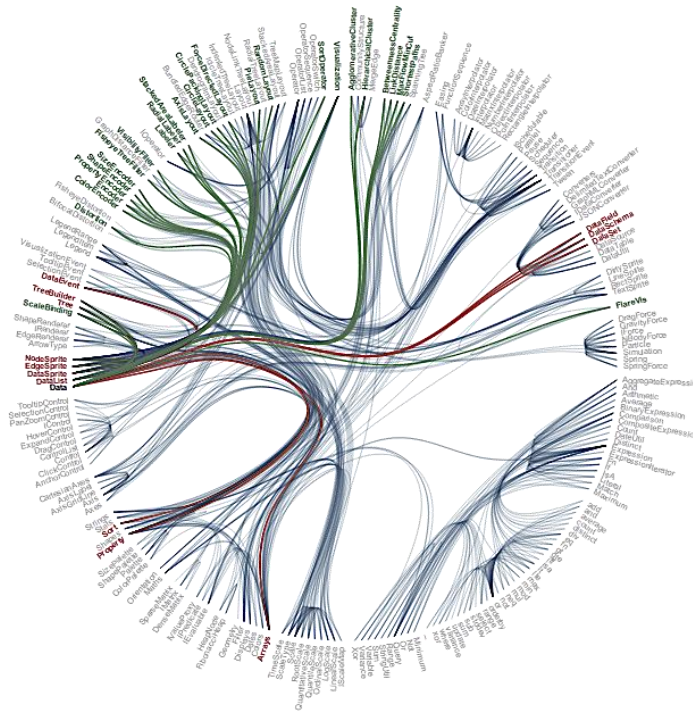
2.5D treemaps are straightforward extensions to treemaps and offer an additional strong visual variable, the cuboid height, for information visualization.

2.5D treemaps, if viewed from above using an orthogonal projection, practically are not faced by occlusions, that is, they have the same visual clarity and readability as treemaps.

As a 3D visualization technique, they can be efficiently implemented on today's devices using the GPU processing capabilities.

For software analytics, they are essential as they map and visualize tree data, offer two independent visual variables (color+height) to allow users to explore correlations between these two independent variables. The primary motivation for cuboids, thus, is not to emphasize the hierarchical structure or nesting.

Hierarchical Circular Bundle View



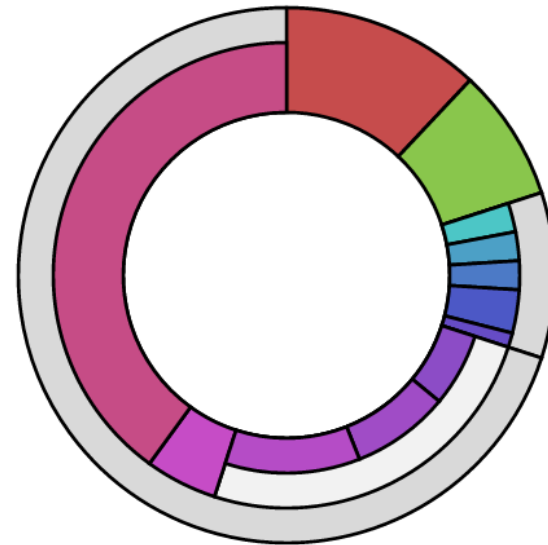
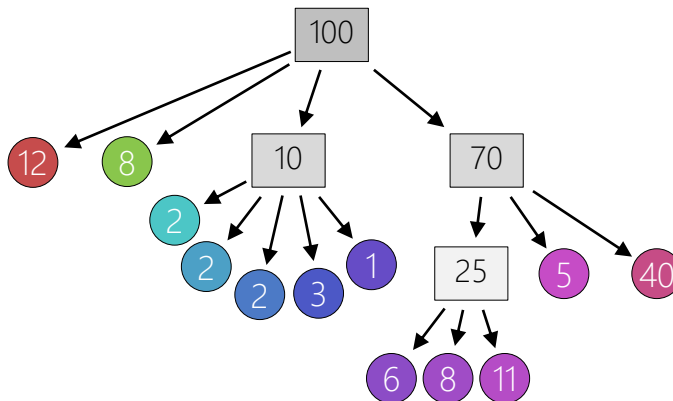
Principles of Circular Bundle Views

Layout Algorithm

Applications of Circular Bundle Views

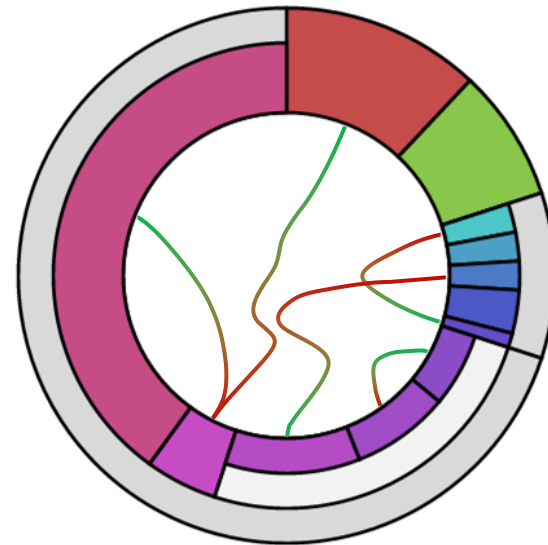
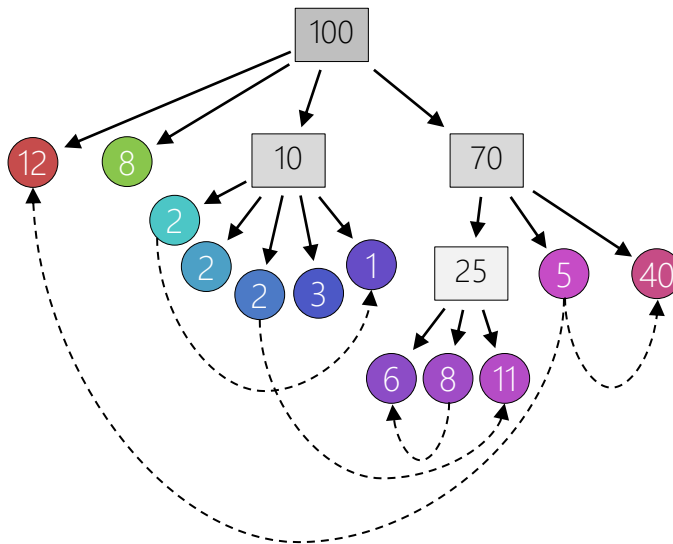
Principles of Hierarchical Circular Bundle Views

- **Tree hierarchy** (left): graph-based visualization. Leaf nodes are colored and have numerical weights, which can be any numerical attribute associated with a node, e.g., file size. The weight of a parent node is the sum of the weights of its child nodes.
- **Hierarchical Circular Bundle View** (right): The root node corresponds to the outer circle. Child nodes are represented by ring segments; the segments are proportional to the weights. Child segments are nested in the segment of their parent.



Principles of Hierarchical Circular Bundle Views

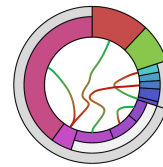
- **Compound Graph:** Represents the tree structure together with relations among their leaf nodes.
- **Relations between Hierarchy Components:** The inner hole allows for visualizing relations among nodes by directly connecting their corresponding ring segments. For each relation, a line connects the two associated segments. The line is bended according the hierarchical position of the connected nodes.



Hierarchical Circular Bundle View

A *hierarchical circular bundle view* (HCBV) defines a visual mapping of

- a tree hierarchy to a ring structure with nested ring segments, whereby the segments are proportional to numerical weights associated with tree nodes (annulus segments), and
- relations between tree nodes with bended lines, which are positioned inside the inner ring (annulus).



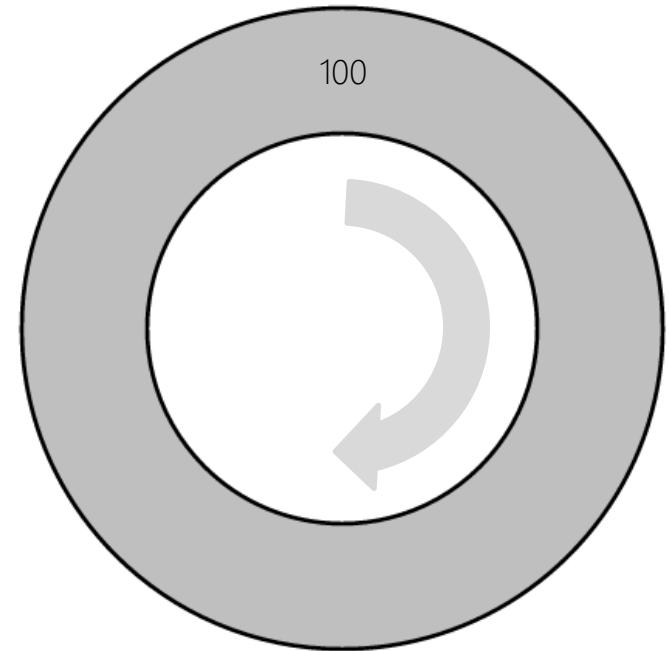
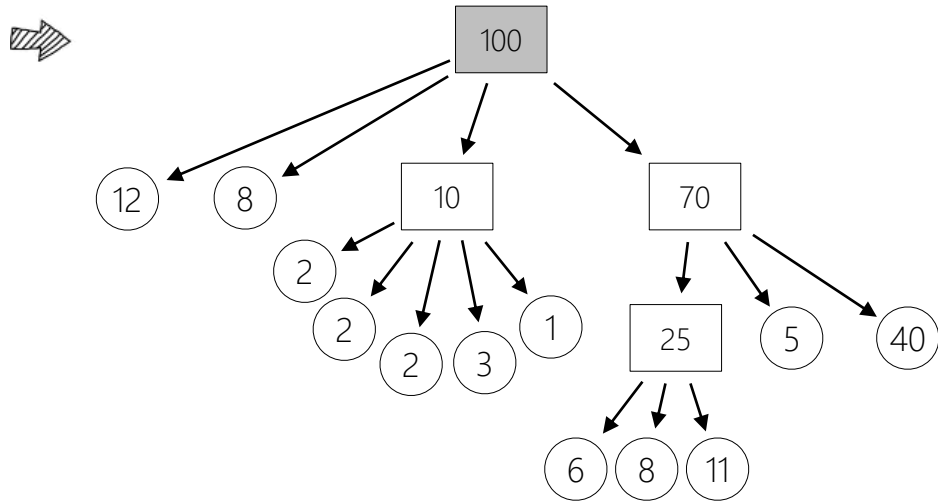
Layout

Each node of given tree hierarchy is represented by a *segment* of the HCBV. Segments of child nodes are contained in the segment of their corresponding parent node.

Efficient Space Usage

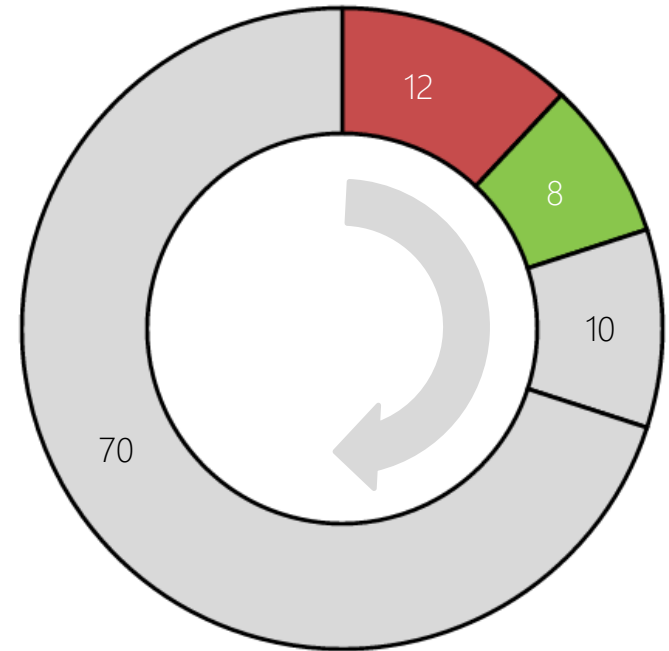
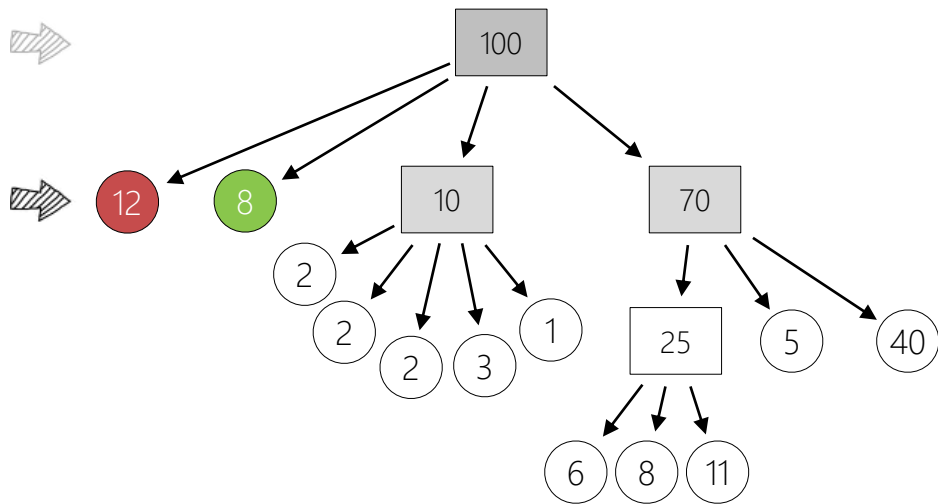
In contrast to graph-based approaches, the HCBV efficiently uses the given circular space, that is, the hierarchy is completely mapped inside the given root ring.

Layout Algorithm



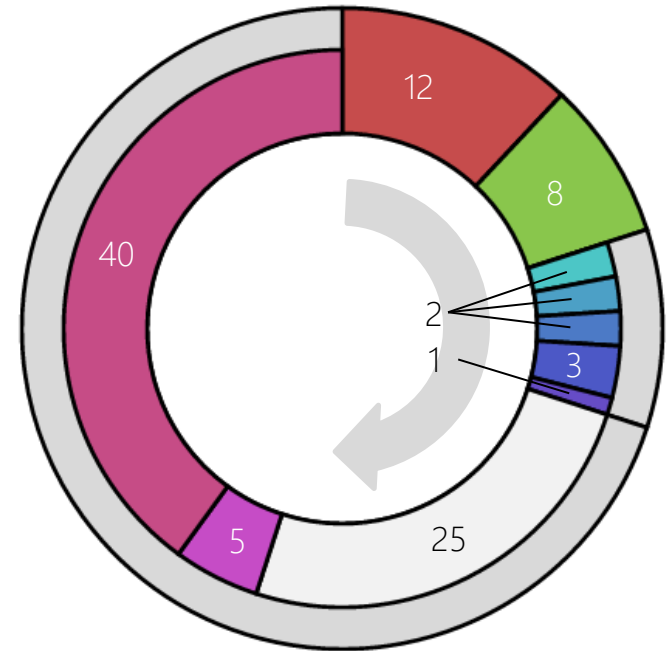
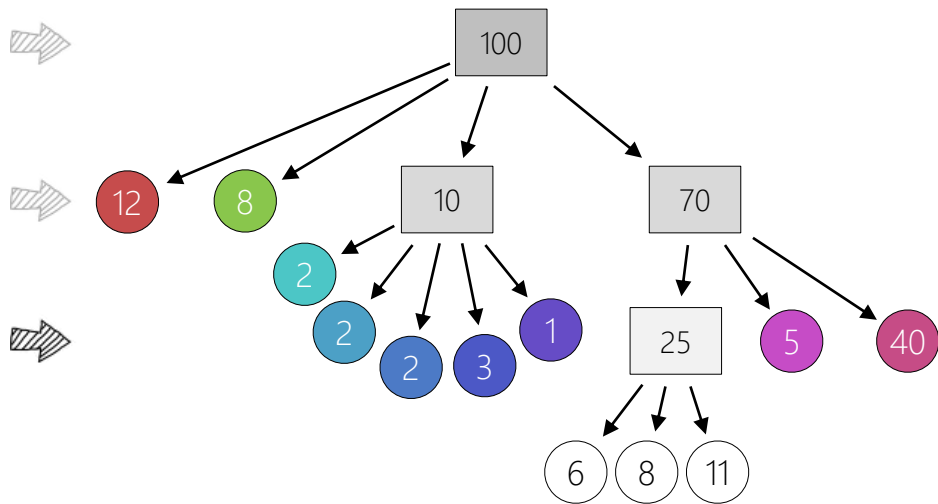
A ring, representing the root node, is recursively subdivided into segments for each of its children.

Layout Algorithm



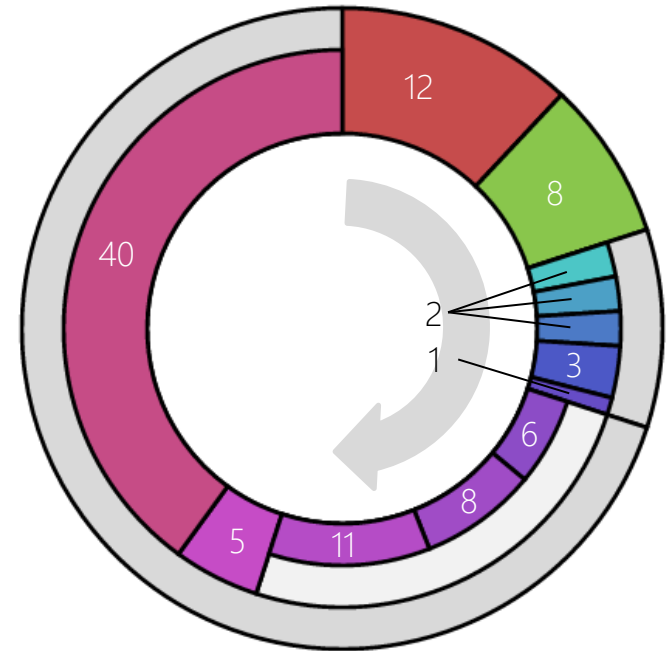
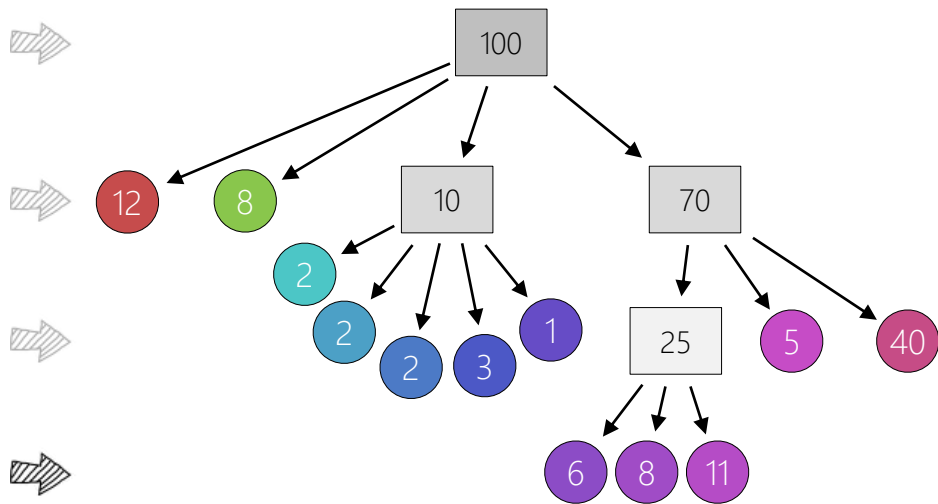
For each child a ring segment is created with an angle proportional to its weight.

Layout Algorithm



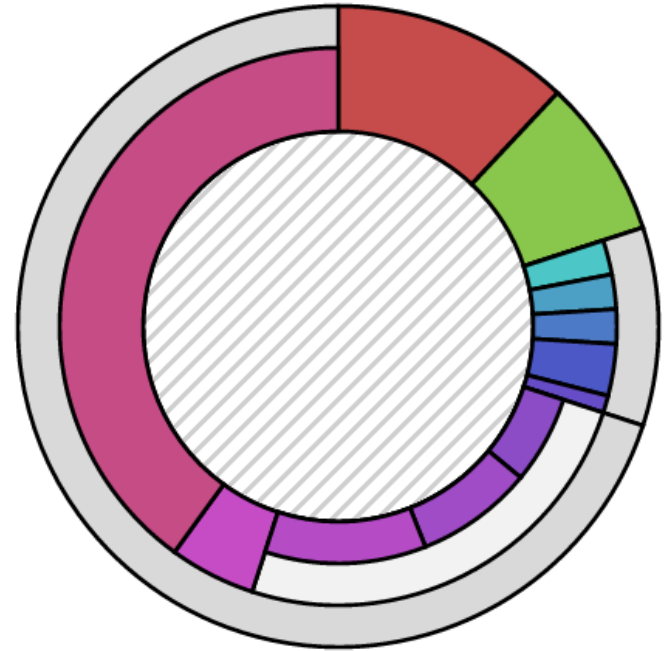
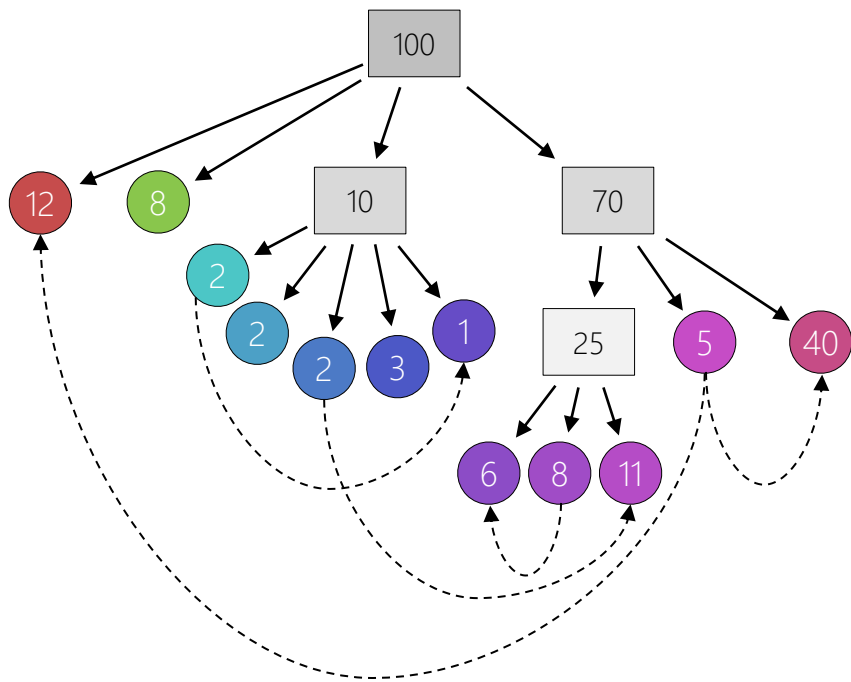
For each new depth in the hierarchy, the ring segments have a lower outer radius than their parents to express the parent-child relationship.

Layout Algorithm



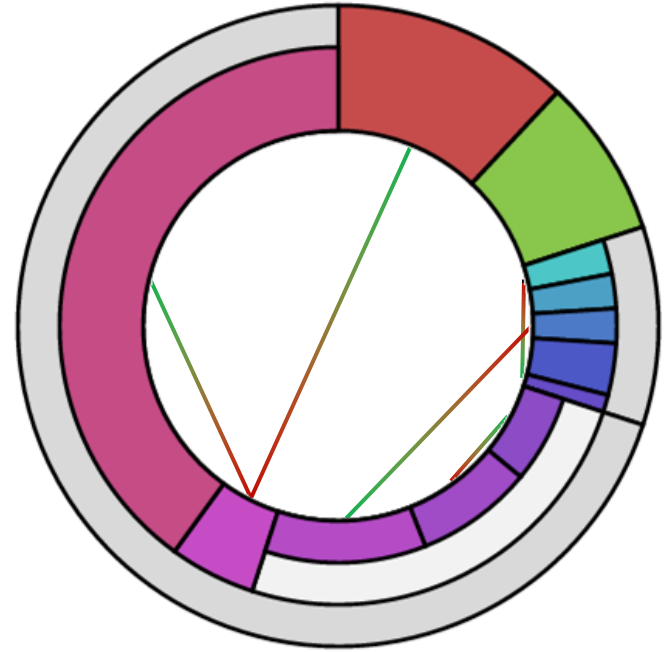
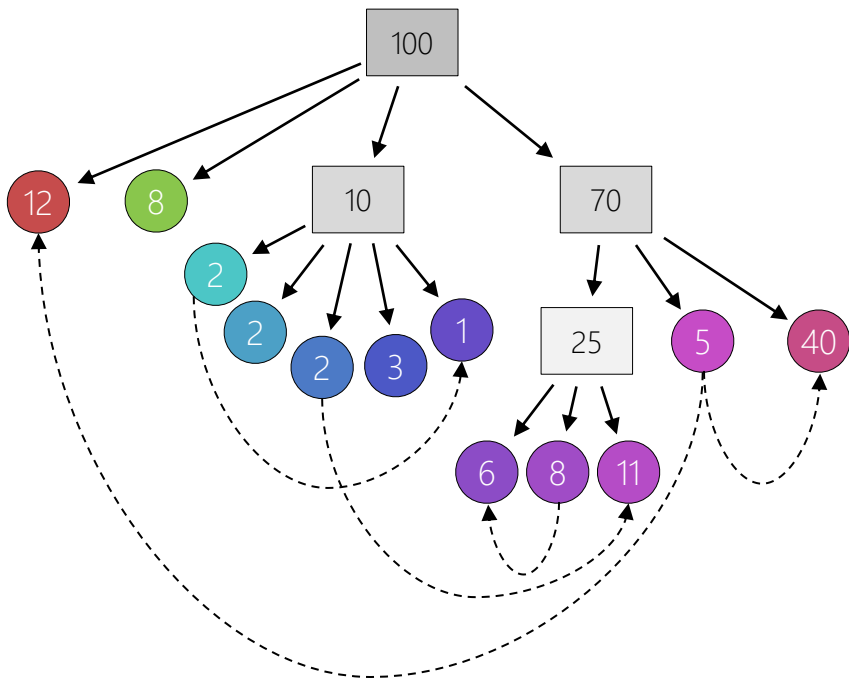
The final layout shows the hierarchical structure. All segments of leaf nodes have the same inner radius, that is, they end in the inner hole.

Layout Algorithm



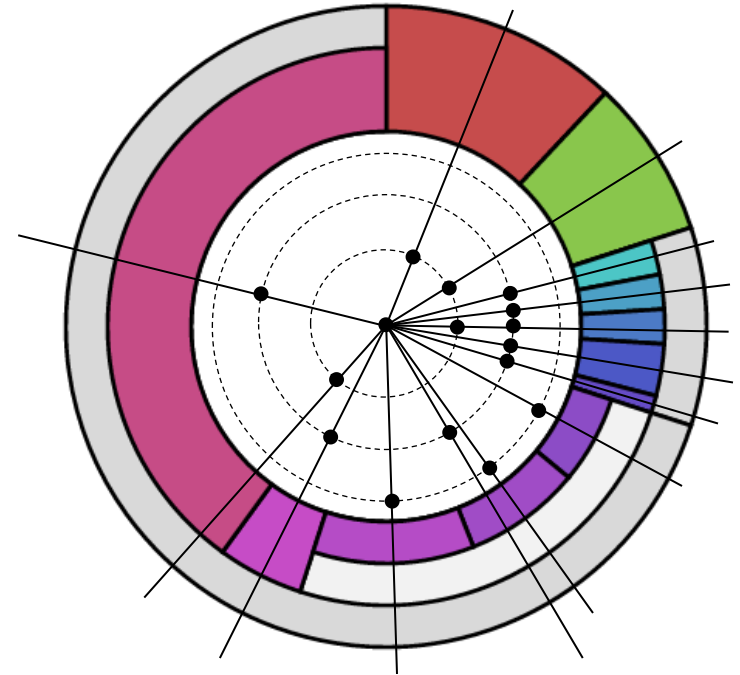
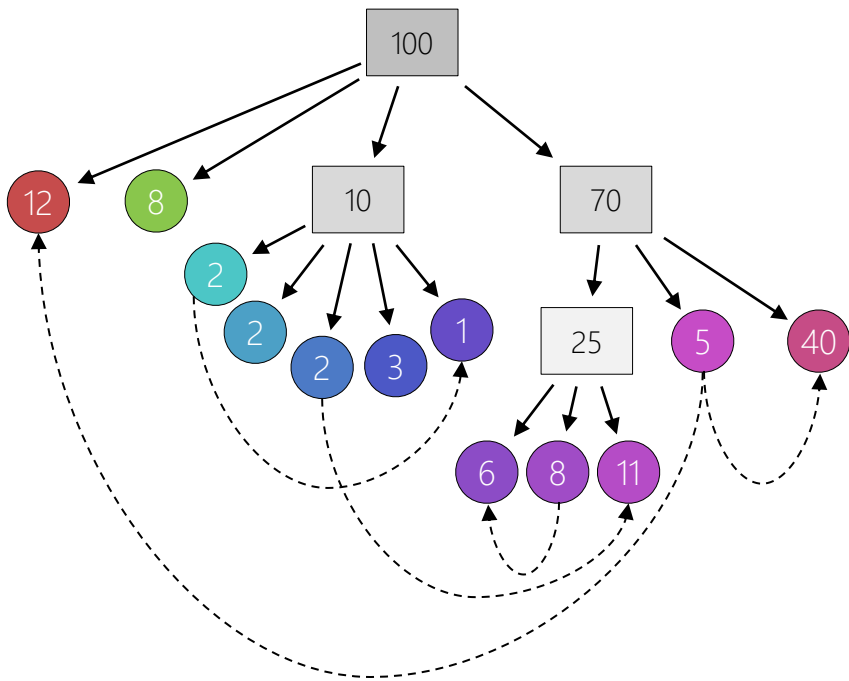
Relations among nodes can now be visualized inside the hole by drawing *edges*.

Layout Algorithm



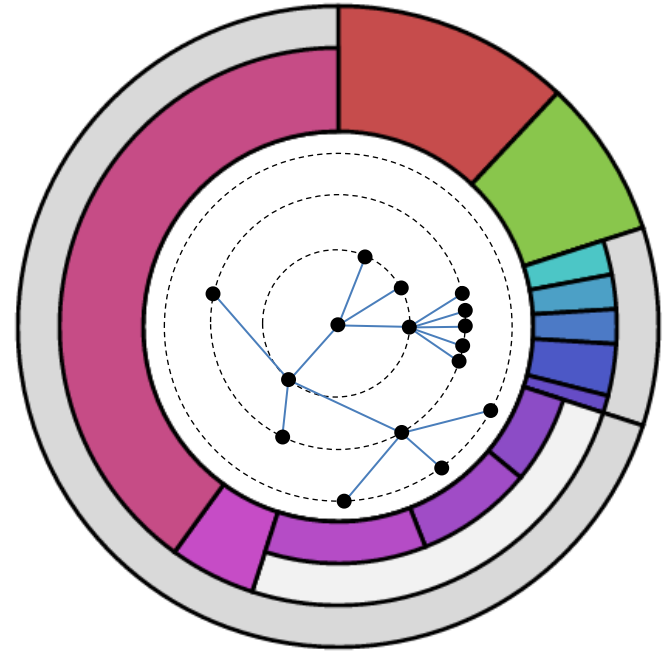
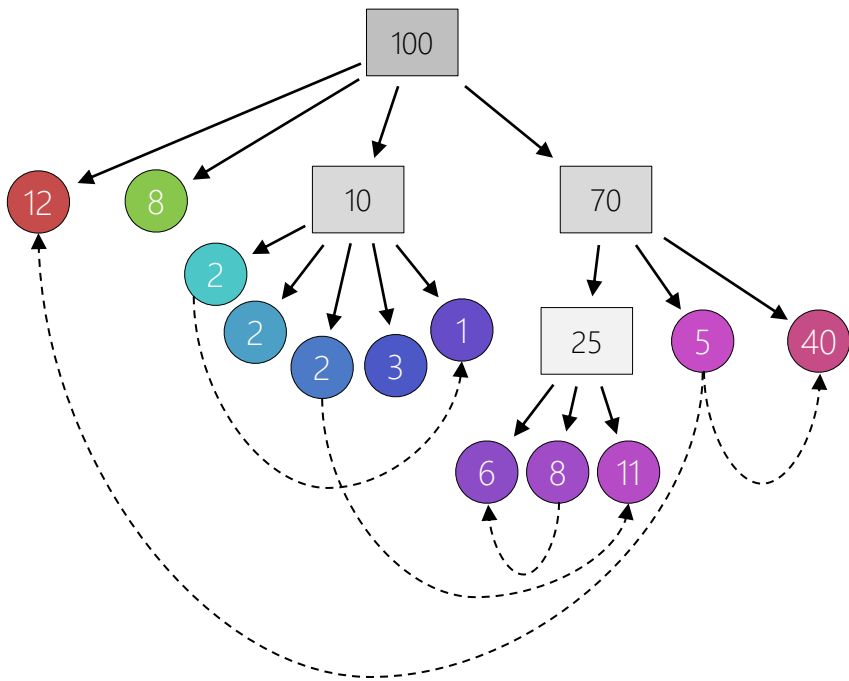
Visualizing edges by straight lines would lead to low readability for relations between small elements positioned near to each other.

Layout Algorithm



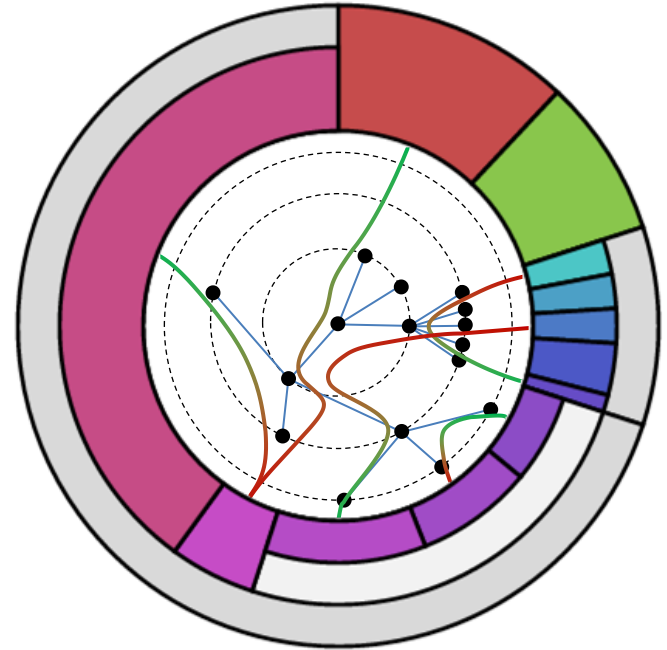
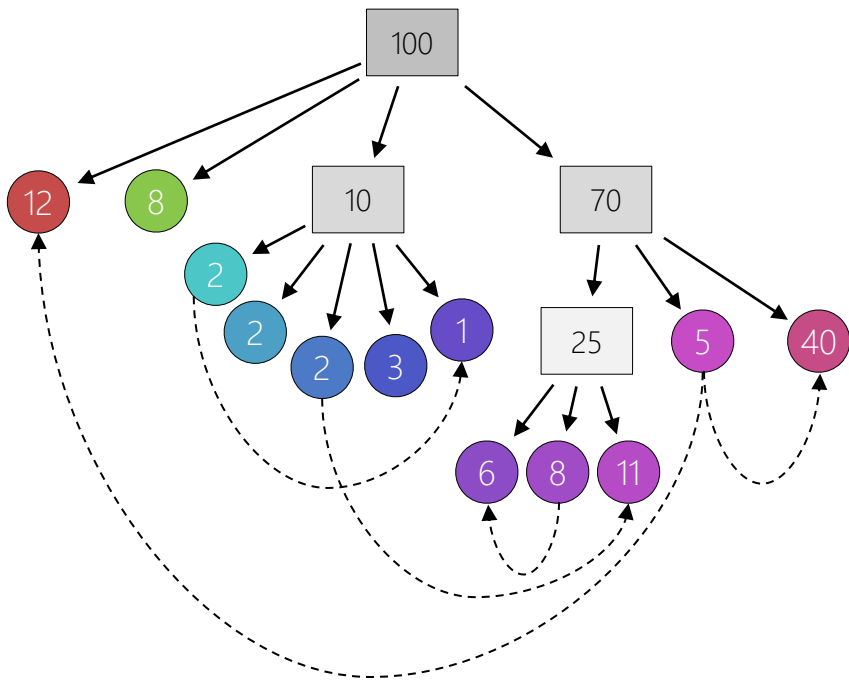
Edge bundling is based on the hierarchical layout,
which is first mirrored inside the circle by a radial tree.

Layout Algorithm



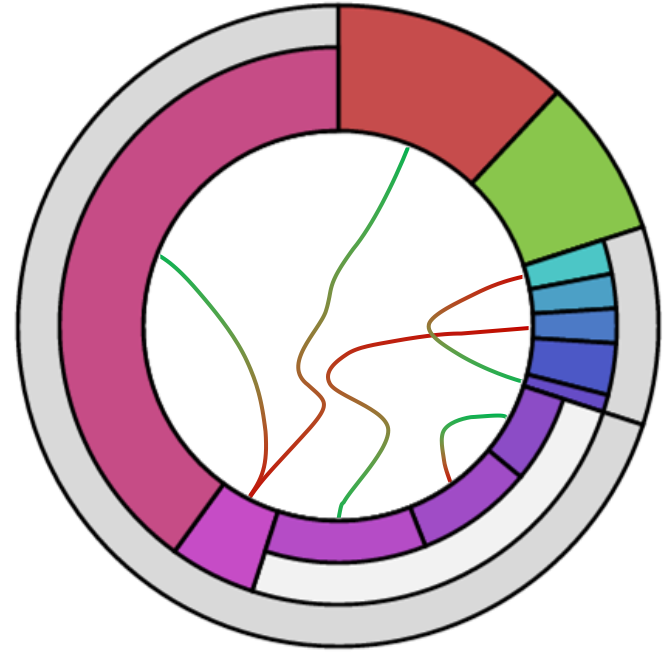
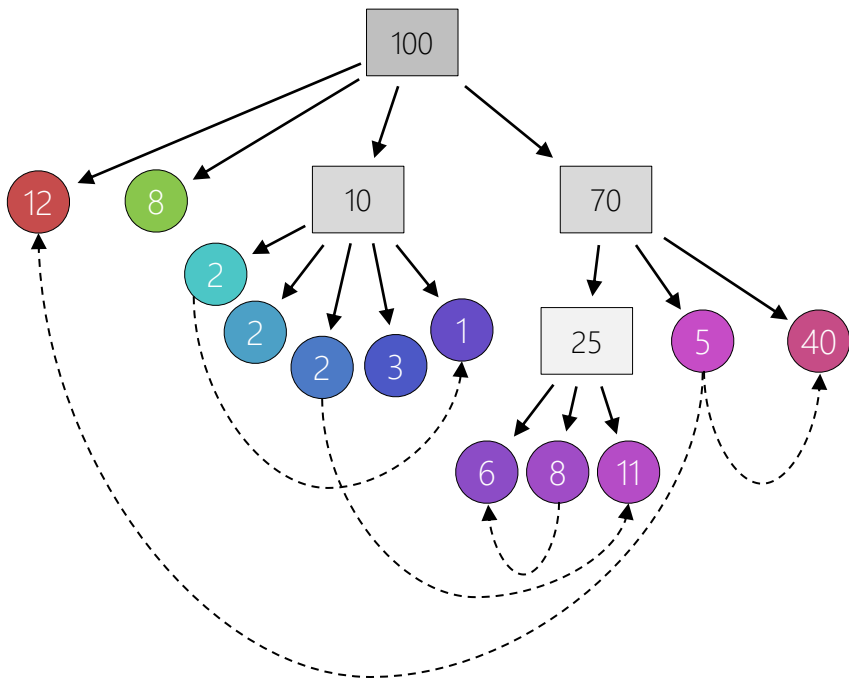
The node's positions in the radial tree are used as control points for Bezier-curves used to visualize edges.

Layout Algorithm



The node's positions in the radial tree are used as control points for Bezier-curves used to visualize edges.

Layout Algorithm



The final layout “bundles” bended lines implicitly by the mirrored points of their lowest common ancestor and helps to depict relations between small items that are located nearby.

- Visual software analytics uses hierarchical circular bundle views to visualize any type of relations defined among hierarchical system structures. The edges used to represent a relation are bended and bundled according to the hierarchy position of their corresponding nodes.
- Examples of relation types:
 - Include dependencies
 - Call dependencies
 - Deployment dependencies
 - ...

OpenHPI - Automated Visual Software Analytics - Prof. Dr. Jürgen Döllner - 2015

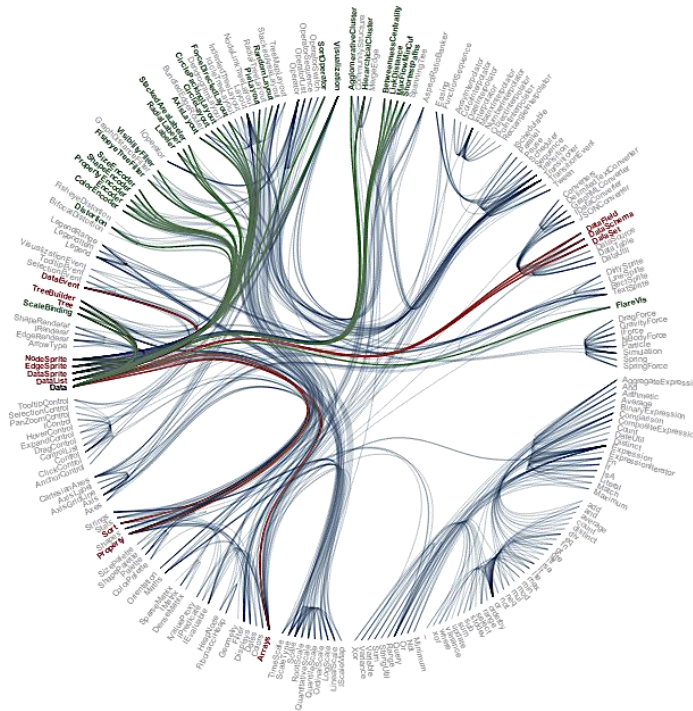
To map hierarchical data together with relations among their components, **hierarchical circular bundle views** provide a space-efficient, clear visualization approach. The edges, which represent relations, can be bundled and reflect the hierarchy setting.

HCBV provide context and facilitate identifying and exploring relations within a hierarchy.

In visual software analytics HCBV are one key instrument to explore, analyze, and monitor complex big software engineering data in form of **compound graphs**.

Applications of HCBV in software analytics include examining, analyzing, and comprehending inheritance relations, function call relations, module dependencies, etc.

Hierarchical Circular Bundle View



Principles of Circular Bundle Views

Layout Algorithm

Applications of Circular Bundle Views

Compound Graph

A hierarchical structure (tree) with a set of relations between leaf nodes.

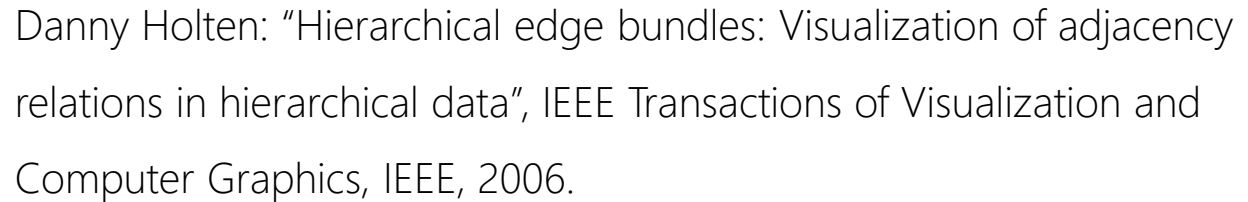
Hierarchical Circular Bundle View

A *hierarchical circular bundle view* (HCBV) defines a visual mapping of (1) a tree hierarchy to ring structure with nested ring segments, whereby the segments are proportional to numerical weights associated with tree nodes, and (2) a visual mapping of relations between tree nodes with bended lines, which are positioned in the inner hole of ring.

Edge Bundling

A rendering technique that determines a compact layout of bended edges.

HPI Hasso Plattner Institut



- Hierarchical circular bundle views and software engineering data are a good team: Compound graphs are frequent, and HCBVs are effective means to visualize them.
- HCBV have one big advantage: They can visually layout relations in an ordered way (in contrast to many graph-based visualization) as there is an explicit area (the hole in ring structure) that allows us to connect all leaf nodes of the hierarchy.
- HCBV can be used to interactively edit relations by moving leaf nodes, by reconnecting edges or edge bundles - a high-level approach to edit relations among hierarchy nodes.
- Technically, HCBV rendering implementations allow for visualizing large hierarchies in real-time, that is, they are perfect elements in software analytics user interfaces.