ViSiTR: 3D Visualization for Code Visitation Trail Recommendations

Roy Oberhauser
Computer Science Dept.
Aalen University
Aalen, Germany
email: roy.oberhauser@hs-aalen.de

Abstract—**The rapid digitalization occurring in our society depends on massive amounts of data and software running on various devices. This, in turn, entails the creation and maintenance of an ever-increasing volume of computer program code. This situation is exacerbated by a limited pool of trained human resources that must quickly comprehend various sections of program code. Thus, effective and efficient automated tutor systems or recommenders for program comprehension are imperative. Furthermore, advances in game engine and PC performance have hitherto been insufficiently utilized by software engineering tools to leverage the potential that 3D visualization of code structure and navigation can provide. This paper introduces ViSiTR (3D Visualization of code viSitation Trail Recommendations), an approach that utilizes program code as a knowledge base to automatically recommend code visitation trails with visual 3D navigation to support effective and efficient human program code comprehension. A case study with a prototype demonstrated the viability of the approach but found scalability issues for large projects.**

Keywords - program code comprehension; recommendation systems; learning models; intelligent tutoring systems; knowledge-based systems; software engineering; engineering training; computer education; software visualization.

I. INTRODUCTION

This is an extended paper of [1]. The increasing demand for and utilization of software throughout society and industry results in soaring volumes of (legacy) program code and associated maintenance activity. Although the total lines and growth of program code worldwide is untracked and unknown, it's been estimated that well over a trillion lines of code (LOC) exist with 33bn added annually [2], while a study of 5000 active open source software projects shows code size doubling on average every 14 months [3].

This has ramifications on the amount of relatively expensive labor involved in software development and maintenance. Approximately 75% of technical software workers are estimated to be doing maintenance [4]. Moreover, program comprehension may consume up to 70% of the software engineering effort [5]. As an example, the Year 2000 (Y2K) crisis [6] with global costs of $375-750 billion provided an indicator of the scale and importance of program comprehension. Moreover, the available pool of programmers to develop and maintain code remains limited and is not growing correspondingly. This is exacerbated by high employee turnover rates in the software industry, for example 1.1 years at Google [7].

Thus, there is resulting pressure on programmers to rapidly come up to speed on existing code or comprehend and maintain legacy code (a type of knowledge) in a cost-effective manner. It thus becomes imperative that programmers be supported with automated tutors and recommenders that efficiently and effectively support program code comprehension. In this space, recommendation systems for software engineering provide information items considered to be valuable for a software engineering task in a given context [8].

However, such recommenders often lack integrated visualization support, hampering their ability to achieve more comprehensive program comprehension support. This relates to an essential difficulty of software construction asserted by F. P. Brooks Jr., namely the invisibility of software, since the reality of software is not embedded in space [9]. The most common formats used for the comprehension of program code include text or the two-dimensional Unified Modeling Language (UML).

In prior work, we introduced ReSCU [1], a knowledge-centric recommendation service and planner for program code comprehension. This was enhanced with support for cognitive learning models in C-TRAIL [10]. Separately, we developed a 3D flythrough visualization approach called FlyThruCode [11] that offers opportunities for grasping software program structures utilizing information visualization to support exploratory, analytical, and descriptive cognitive processes.

This paper introduces ViSiTR (3D Visualization of code viSitation Trail Recommendations), an approach for the 3D visualization of automatically recommended program comprehension code trails in alignment with various cognitive learning model styles and the "holey quilt" theory [12]. ViSiTR can be viewed as an intelligent tutor system with a 3D visual interface, applying a practical form of granular computing [13] and concepts like knowledge distance [14] to automatically recommend knowledge navigation as a Hamiltonian cycle [15], a special case of the traveling salesman problem (TSP) [16], in an unfamiliar knowledge landscape consisting of program code.

The paper is organized as follows: the next section discusses related work. This is followed by Section III, which provides background information. Section IV then describes the solution concept that is followed by a description of a prototype implementation. Section VI then presents a case study, which is followed by a conclusion.

II. Related Work

An overview of recommendation systems in software engineering is provided by [8]. In the Eclipse Integrated Development Environment (IDE), NavTracks [17] recommends files related to the currently selected files based on their previous navigation patterns. Mylar [18] utilizes a degree-of-interest model in Eclipse to filter our irrelevant files from the File Explorer and other views. The interest value of a selected or edited program element increases, while those of others decrease, whereby the relationship between elements is not considered. In support of developers with maintenance tasks in unfamiliar projects, Hipikat [19] recommends software artifacts relevant to a context based on the source code, email discussions, bug reports, change history, and documentation. The eRose plugin for Eclipse mines past changes in a version control system repository to suggest what is likely also related to this change based on historical similarity [20]. To improve navigation efficiency and enhance comprehension, the FEAT tool uses concern graphs either explicitly created by a programmer [21] or automatically inferred [22] based on navigation pathways utilizing a stochastic model, whereby a programmer confirms or rejects them for the concern graph. With the Eclipse plugin Suade [23], a developer drags-and-drops related fields and methods into a view to specify a context, and Suade utilizes a dependency graph and heuristics to recommend suggestions for further investigation. To support the usage of complex APIs in Eclipse, the Prospector system [24] recommends relevant code snippets by utilizing a search engine in combination with Eclipse Content Assist. Strathcona [25] analyzes structural facts of an incomplete code selection and utilizes heuristic matches to determine the most similar example. The Eclipse plugin FrUiT [26] supports example framework usage via association rule mining of applications that utilize a specific framework. Codetrail [27] connects source code and hyperlinked web resources via Eclipse and Firefox. Yin et al. [28] propose applying coarse-grained call graph slicing, intra-procedural coarse-grained slicing, and a cognitive easiness metric to guide programmers from the easiest to the hardest non-understood methods. Cornelissen et al. [29] survey work on program comprehension via dynamic analysis.

Although we attempted to provide a more detailed practical comparison with many of the above program comprehension tools, we abandoned our effort since the tool software was mostly either inaccessible or after download we were unable to get it to successfully execute. Our comparison is thus based on research paper descriptions. In contrast to the related work above, various facets differentiate the ViSiTR approach. ViSiTR is able to recommend code region visitations and plan a code trail order without necessitating an explicit context or prior history, without requiring the intervention or confirmation of a human expert. Furthermore, the approach is unique in applying a conceptual mapping of geographical points of interest (POI) and the traveling salesman problem/planning (TSP) to source code and the generation of code trail planning, with a Hamiltonian cycle to avoid unnecessary revisitations.

With regard to 3D software visualization tools across the various software engineering areas, an overview and survey is given by Teyseyre and Campo [30]. Software Galaxies [31] provides a web-based visualization of dependencies among popular package managers and supports flying. Every star represents a package that is clustered by dependencies. CodeCity [32] is a 3D software visualization approach based on a city metaphor and implemented in SmallTalk on the Moose reengineering framework. Buildings represent classes, districts represent packages, and visible properties depict selected metrics. Wettel et al. [33] showed a significant increase in terms of task correctness and decrease in task completion time. Rilling and Mudur [34] use a metaball metaphor combined with dynamic analysis of program execution. X3D-UML [35] provides 3D support with UML in planes such that classes are grouped in planes based on the package or hierarchical state machine diagrams. A case study of a 3D UML tool using Google SketchUp showed that a 3D perspective improved model comprehension and was found to be intuitive [36].

In contrast to the above work, ViSiTR supports visual code trails that can be recommended, captured, and replayed via 3D fly-thru visitation using multiple and dynamically switchable metaphors, custom and automatic annotation/tagging, and the display of localized contextually-relevant program code data (code, metrics, UML) in a heads-up display, thereby intermixing 2D data while flying through the 3D space. Because the source code is transformed into an XML description, various programming languages can be easily supported. Its plugin architecture permits the integration of program code data from various separate tools.

III. Background

For our purposes, it may be helpful to view program code comprehension from the holey quilt theory perspective [12], based on [37] and [38]. According to this metaphor "novice programmers' early comprehension models can be characterized by a pattern of 'holey knowledge' (i.e., an incomplete patchwork of fabric, with empty cells and missing stuffing)" [38] quoted in [12].

For the programmer, her or his program comprehension knowledge base can be viewed as a block model as shown in Figure 1 both functionally (the "what" in green) and structurally (the "how" in gold). Structure includes both the text surface of the program (right column), including its syntax, semantics, and style as well as its control structure (middle column). Its function goals (left column) considers its intent or goals at various levels. The finest granularity considered is on the atom (bottom) row, considering language elements and the result of any statement. The third row is labeled block, and consists of grouping within some region of interest (ROI). The second row labeled relations deals with the relations between method calls. The top row deals with the macro-structure of the overall program. The knowledge level (depth dimension) about any element within this structure can vary from fragile to moderate to deep, and is often correlated with the time on task (depth dimension), which can vary from low to medium to high.

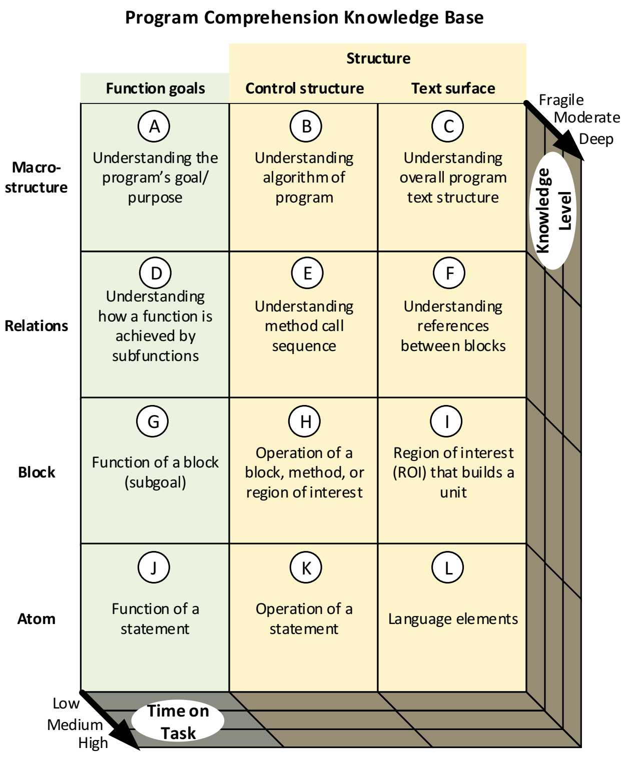

Program Comprehension Knowledge Base
Structure
Function goals
Control structure
Text surface
Fragile
Moderate
Deep
Knowledge Level
Macro-structure
Relations
Block
Atom
A
Understanding the program's goal/ purpose
B
Understanding algorithm of program
C
Understanding overall program text structure
D
Understanding how a function is achieved by subfunctions
E
Understanding method call sequence
F
Understanding references between blocks
G
Function of a block (subgoal)
H
Operation of a block, method, or region of interest
I
Region of interest (ROI) that builds a unit
J
Function of a statement
K
Operation of a statement
L
Language elements
Low
Medium
High
Time on Task

Figure 1. Depiction of the "holey quilt" theory of program comprehension, adapted from [12].

Given this perspective, a hermeneutic view of program comprehension is assumed, consisting of a dynamic process of program code interpretation that involves recurrent transitions between some overall picture down to various myopic code snippets and back to the overall picture again, successively assembling a (hopefully) coherent and consistent picture based on an interpretation of its syntax, semantics, and intention.

In the constructivist theory of human learning, humans actively construct their knowledge [39]. We thus view program comprehension as individualistic for aspects such as capacity, speed, motivation, and how mental models are constructed. Additionally, programmers possess different application-independent general and application-specific domain knowledge. Information processing habits of an individual are known as cognitive learning styles. ViSiTR provides individual and automated support for various learning model (*M:*) styles, primarily ordering or adjusting concept location (code area) visitation scope.

M:Bottom-Up: in this learning model, chunking [40] is used with the program model being correlated with a situation model [41]. Microstructures are mentally chunked into larger macrostructures as comprehension increases, as depicted by the row ordering in Figure 1 from bottom up. ViSiTR assumes a package hierarchy.

M:Top-Down: this model [42] is typically applicable when familiarity with the code, system, domain, or similar system structures already exists. Beacons and rules of discourse are used to hierarchically decompose goals and plans, as depicted by the rows in Figure 1 from top down. To automate support, ViSiTR assumes a cluster hierarchy and starts trails from the highest hierarchy.

M:Topics/Goal: when programmers are given a specific task, they tend to utilize an as-needed strategy to comprehend only those portions relevant for the task [43]. This correlates with ROIs of Figure 1. To support this simply, ViSiTR supports investigating a limited code subset via topic filtering. Topic filters (positive and negative) can be shared and support a goal (e.g., optimize memory) or apply to a specific topic (e.g., security, database access, user interface).

M:DynamicPath: in this model, ordering is oriented on actual invocation execution traces [44], which correlates with block E of Figure 1.

M:Exploratory: this model supports either discovery or analysis to confirm a hypothesis, with the learner actively deciding and controlling the navigation. It is supported by default, since a user can deviate at any time.

IV. SOLUTION APPROACH

The ViSiTR solution approach, incorporating various concepts from [1], [10], and [11], focuses on supporting the learning, understanding, and navigation of unfamiliar program source code by programmers in an automated, systematic way, without requiring additional knowledge, historical information, or human expert assistance. In alignment with the holey quilt theory, we hold the view that a programmer's view of any complex team-based software project given limited time constraints is unlikely to ever be comprehensive, leaving knowledge level "holes" from a knowledge level scale between none to deep knowledge. Thus, a major intent is to provide efficient code trails that focus on the important methods to comprehend given some limited timeframe.

A. Principles

The ViSiTR solution approach is based on these principles (P:):

P:POI: program source code locations are identified and viewed as Points-of-Interest (POI) (or knowledge entities), analogous to geographical locations in navigational systems and ROI in the holey quilt theory. Each POI is identified by some unique name, for instance in Java its fully qualified name (FQN) consisting of the concatenation of a package name, class name, colon, and method name. A POI can be viewed as a granule or information entity of interest in a knowledge "landscape", but this could be a function in non-object-oriented languages, an object method, a class, or a package.

P:POIRanking: to determine the importance of a POI (or knowledge granule) for human comprehension, they are ranked relative to each other. The algorithm MethodRank described below exemplifies such a ranking that fulfills this principle.

P:POILocality: POI locality, which can be conceptually viewed as knowledge closeness from the perspective of knowledge distance [14], is taken into consideration. This is intended to address the cognitive burden of context switches to a human when viewing program source code, by ordering POIs such that the number of unnecessary switches in a POI visitation order is reduced. The POI Distance calculation described later is an example for applying this principle.

P:Timeboxing: the amount of time available for concentrated comprehension and learning is assumed to be limited, and we assume learning is chunked into one or more sessions. Thus, the visitation time for POIs is estimated, and only the subset of priority ordered POIs that can be feasibly visited in the given timebox (deadline is midnight if no other time is provided) is first selected, and this prioritized subset is then reordered according to locality for that session. We assume that a session will not be interrupted, but that following sessions may not occur, therefore we use priority sorting first, and then resort the session subset by locality to limit jumping or thrashing.

P:CodeTrails: the recommendation service provides code trails as output with a navigation and visitation order recommendation for the POIs, whereby POI locality is taken into account. A mapping of the TSP and related planning algorithms are applied to these granules (the POIs) and the associated knowledge distance between them. While the path suggested may not necessarily be the most optimal path, it provides an efficient path nonetheless through the knowledge landscape (source code). In ViSiTR, POI visitation planning via the generated code trails focuses on invocation relationships rather than class relationships. Not following class relationships can be viewed as supported by an empirical eye-tracking study finding that "software engineers do not seem to follow binary class relationships, such as inheritance and composition" [45]. Two modes are supported: initial trail mode that generates a trail from scratch, and refactor trail mode that dynamically incorporates user actions and re-optimizes the trail based on the visited POI and the session time left. Visited POIs (including deviations) are detected via events and automatically removed from the next suggested trail.

P:User profile: user's knowledge level (e.g., familiar vs. unfamiliar) and competency level (junior vs. senior) are taken into consideration.

B. Visualization Principles

ViSiTR includes these visualization principles (P:V:):

P:V:Multiple 3D metaphors: The input for a model instantiation is an import of project source code. One of the first issues faced in visualization is how to best model and visualize the program code structures. Because of the lack of any standardization or norms in this area, and to support the spectrum of individual preferences, support is provided for modeling and switching between *multiple visualization metaphors*, analogous to the concept of skins. Our initial model focuses primarily on modeling and visualizing object-oriented packages, classes, and their relationships such as associations and dependencies. Initially, we support two metaphors "out-of-the-box" to provide examples of skins, and custom mappings to other objects types are possible. In the *universe* metaphor, each planet represents a class with planet size based on the number of methods, and solar systems represent a package. Multiple packages are shown by layer solar systems over one another. In the *terrestrial* metaphor, buildings can represent classes, building height can represent the number of methods, and glass bubbles can group classes into packages. Relationships are modeled visually as light beams or pipes by default.

P:V:Cockpit: analogous to an airplane cockpit, this provides information to the user on the border of the screen, and has input fields for searching for a class or method or navigating directly to a class. Buttons can be depressed to indicate preferences. A minimap on the upper right of the screen provides a high-level overview of the entire landscape and one's relative location in a small area.

P:V:Heads-Up Display (HUD): This provides a transparent glass on the screen with additional context-specific information. The type of information displayed can be changed via left/right arrows on the screen edges. The transparency level can be adjusted in the cockpit to provide a less opaque background if desired (e.g., to view code better). Various HUD screens are provided: *Tags* for automatic and manual persistent annotations/tags; *Source Code* where the program text is shown in scrollable form; *UML* where UML diagrams are dynamically generated in 2D; *Metrics* which shows text-based metrics due to the large number of possible metrics (any of which may be of interest to the user); *Project Management* to manage the metaphor, load a project record, or import a new project; and *Filtering* that provides selectors for adjusting the visibility of packages by interest.

P:V:Flythrough navigation: both mouse and keyboard support for 3D navigation (motion) in all directions is provided, as is autopilot or lockon to navigate to a specific class.

P:V:Intermixing 3D/2D: support for dynamically generated 2D UML is integrated in the 3D environment, enabling the usage of this standard notation to support the understanding of a particular area of interest.

P:V:CodeTrails: We provide the ability to capture and record a visitation trail as well as provide a playback ability, displaying the previous, current, and the next visitation node. Furthermore, the trail can be recommended and adjusted adhoc by the ViSiTR service. The HUD features can be used to view the code for any visited class.

C. ViSiTR Solution Architecture

ViSiTR consists of a visual client that utilizes a recommender service. The architecture for the recommender service is shown in Figure 2 and consists of four primary modules: *Cognitive Learning, Knowledge Processing, a Database Repository, and Integration*.

The *Cognitive Learning* module supports various program code learning Models, Goals, Topics, execution Traces, and visitation History. The *Knowledge Processing* module includes the components POI Prioritizer for ranking POIs, a POI Filter that filters based on visitations or topics, a Trail Estimator for visitation times, and a Trail Planner for planning the POI visitation time and order. The *Database*

Repository utilizes appropriate database types to retain metadata, knowledge, or data in forms such as a graph database for modeling the source code as a graph of nodes with properties, and a relational/NoSQL database for dealing with non-graph-related knowledge related to source code. The *Integration* module includes a Web Service API (application programming interface) for development tool integration, an Input Processor to process inputs, transformations, and events (such as a POI visit) including analysis and tracing inputs, and a Trail Generator for generating a planned trail into a desired format.

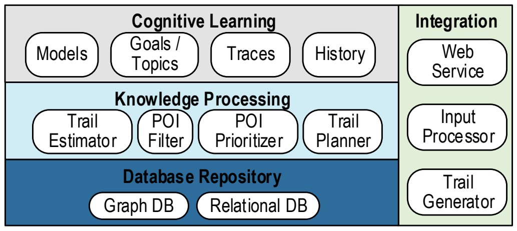


Figure 2. ViSiTR recommender service architecture.

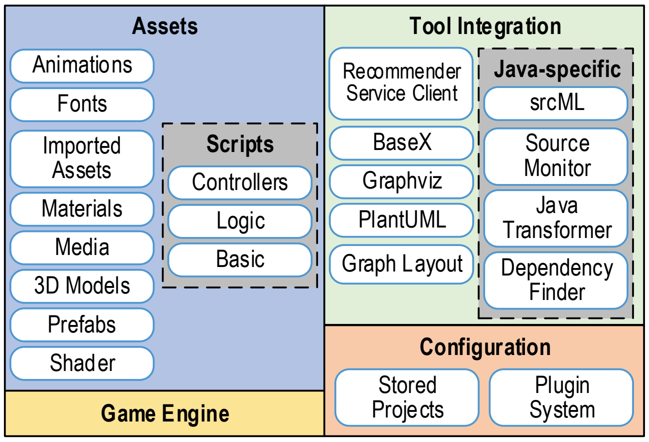


Figure 3. ViSiTR visual client architecture.

Figure 3 shows the visual client architecture which is based on a game engine and supports extensibility via plugin-ins. Assets are used by the game engine and consist of Animations, Fonts, Imported Assets (like a ComboBox), Materials (like colors and reflective textures), Media (like textures), 3D Models, Prefabs, Shaders (for shading of text in 3D), and Scripts. Scripts consist of Basic Scripts like user interface (UI) helpers, Logic Scripts that import, parse, and load project data structures, and Controllers that react to user interaction. Logic Scripts read Configuration data about Stored Projects and the Plugin System (input in XML about how to parse source code and invocation commands). Logic Scripts can then call Applications consisting of General and Java Applications. General Applications currently consist of BaseX, Graph Layout consists of our own version of the KK layout algorithm for positioning objects, Graphviz, PlantUML, and integration with for instance the recommender service as a web service client. Java Applications consist of Dependency Finder, Java Transformer that invokes Groovy scripts, Campwood SourceMonitor, and srcML. Via the designed Plugin system, additional tools and applications can be easily integrated. This was used to integrate the Recommender Service Client which invokes the Recommender Service.

D. *ViSiTR Service MethodRank Calculation*

With regard to *P:POIRanking*, it is assumed that in general, given no other knowledge source besides the source code and assuming limited learning time, it is more essential for the user to become familiar with the methods of a project that are used frequently throughout the code, rather than ones that are only sparsely utilized. Thus, a variation of the PageRank [20] algorithm call *MethodRank* is used to prioritize the POIs, whereby instead of webpages methods are mapped, and instead of hyperlinks, we map invocations. Thus, those methods that have the most references (invocations) in the code set are ranked the highest. While this does not consider runtime invocations (such as loops), it can be an indicator for a method with broader relative utilization and thus likely of greater interest for comprehension. One might argue that certain utility methods such as print or log would perhaps then be ranked highest, but we provide pattern matching mechanisms to include or exclude methods of no interest so the focus can be on domain-relevant methods. Or one might argue against PageRank for webpages, in that the highest ranked webpages are not necessarily the most important, since importance can be viewed differently by various individuals and their distinct perspective and intentions. However, *MethodRank* does provide an indicator of the methods that are heavily used throughout the static code and should thus be understood.

E. *ViSiTR Service POI Distance Calculation*

To address *P:POILocality*, an underlying assumption is that (sub)packages map vertically to (sub)layers and classes serve as a type of horizontal grouping of methods. Thus, the distance between any two POIs (given in (3)) A and B (analogous to geographical distance) is determined by their *vertical* (1) and *horizontal* (2) distance where *ld()* is a layer depth function.

$$VerticalDistance = ld(A) + ld(B) - 2(ld(common)) \quad (1)$$

For instance, given layer A = foo.a.b and layer B = foo.x.y.z (closest common package is foo) so *VerticalDistance* = 3 + 4 - 2(1) = 5.

$$HorizontalDistance = \begin{cases} 0 & if\ class(A) = class(B) \\ 1 & otherwise \end{cases} \quad (2)$$

For instance, the POIDistance between methods in the same class is 0, between classes in the same package 1.

$$POIDistance = VerticalDistance + HorizontalDistance \quad (3)$$

Depending on the implementation, a higher layer may only represent a greater abstraction (e.g., only interfaces) and not necessarily be that far in cognitive "distance".

Nevertheless, any sublayers between them should still be cognitively "closer".

F. *ViSiTR Service Hamiltonian POI Visitation Trail*

Assuming the principles of proper modularity and hierarchy are applied in a given project, a greater distance between POIs is equivalent to a larger mental jump. Thus, to reduce mental effort, once the distance for all pairs has been calculated, we desire the overall shortest trail that provides the visitation order for all POIs such that each POI is visited exactly once except that the starting point is also the end point, i.e., a Hamiltonian cycle. The calculation problem is equivalent to the well-known TSP.

G. *ViSiTR Service Knowledge Processing*

ViSiTR knowledge processing stages are shown in Figure 4 and described below.

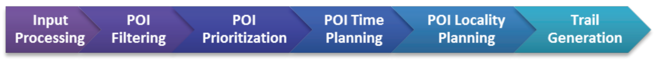


Figure 4. ViSiTR knowledge processing stages.

1) Input Processing: the source code as text files is imported and analyzed. A list of all the POIs in the project as FQNs is determined. The layer of each POI is determined by counting the subpackage depth of its FQN. Whether the project actually utilizes a layer structure or not is irrelevant. This is then used to apply the aforementioned POI distance calculation.

2) POI Filtering: POIs already visited by this user (either in the expected order or out of order) are filtered from the set for the initial planning or replanning.

3) POI Prioritization: the aforementioned MethodRank calculation is used to create an ordered list of POIs.

4) POI Time Planning: the actual POI visitation time is stored per user. Given no prior actual POI visitation time, a default visitation time can be estimated based on a user's profile utilizing a basis time per line of code in seconds, and factors correlated with the size and complexity of the current POI method, the knowledge level (stranger or familiar), and the competency level (junior or senior). Based on the limited session time available and the set of POIs, the POI Time Planner component limits the set to an ordered list by priority that is cut off at the point that the cumlative time exceeds the timeboxed session. This reduces the size of the FQN set for locality planning and traversal.

5) POI Locality Planning: from the resulting set, the POIs are then ordered using a planner for a Hamiltonian cycle and a TSP path that takes locality into account, such that those nearby are visited first before jumping to POIs at a further distance.

6) Trail Generation: the trail with the recommended POI visit order is generated.

H. *ViSiTR Client Visualization Process*

Enabling visualization in the ViSiTR client consists of: 1) *modeling* program code project constructs, structures, and artifacts as well as visual objects, 2) *mapping* these to a metaphor of visual objects, 3) *extraction* via tools of a concrete project's structure (via source code import and parsing) and metrics, 4) *visualization* of the model with alternative metaphors, and 5) supporting *navigation* through the model in 3 dimensional space (simulating movement by moving the camera based on user interaction).

V. IMPLEMENTATION

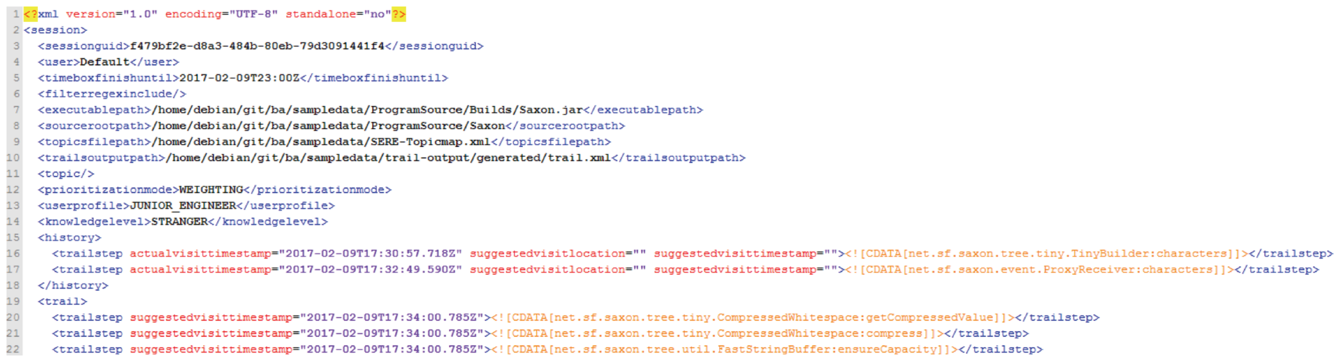
To support validation of the solution concept and architecture, a prototype was realized in Java that analyzes and generates code trails given Java program code as input. For simplification, only normal class methods are considered and method overloading is ignored (a single FQN is used for methods of the same name in trails), but this could be extended via more complex method signatures to handle any method type and overloading where only parameters differentiate methods.

A. *ViSiTR Service Implementation*

To permit the code trail processing and generation to be location-independent (run anywhere, be it local, organization, or cloud and not necessarily burden client PCs) and easily integrate with with various integrated development environments (IDEs), the ViSiTR service was realized as a Web service. It is REST-based (Representational State Transfer), processing client events (e.g., visitations) and outputting updated trails. Thus, if larger projects require more processing, the service could be placed on a more powerful cloud-based server. The *Database Repository* used H2 as a relational and Neo4j as a graph database. To support flexible integration, the output trail format is XML.

The actual POI visitation time is tracked via navigation events received via the web service, with the table METHODRATING_TIMEONMETHOD storing MethodID, UserID, and visitation time (in seconds). POIs that were already visited (expected or not) are then filtered and removed from the replanned trail.

MethodRank requires a data structure with methods (as FQNs) and their target invocation relationships and counts. For this, static code analysis of a project's methods and invoke relationships is performed using jQAssistant 1.0.0 and the GraphAware Neo4j NodeRank plugin [47]. A Cypher query selects all Method FQNs and their invoked Method FQNs and the result is exported to a CSV file. Self-references (such as recursion) are ignored. A separate simplified graph is then created by importing the CSV file into the Static Analysis Program with FQN(Method)->INVOKES->FQN(TargetMethod) relationships in the Neo4j server. GraphAware NodeRank then provides NodeRanks (i.e., MethodRanks) for every node (Method) for the number of invocations with the NodeRank stored in each node's property (Figure 5 shows a partial graph in Neo4j). The result is retrieved via the Neo4j REST API in JSON



```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<session>
  <sessionguid>f479bf2e-d8a3-484b-80eb-79d3091441f4</sessionguid>
  <user>Default</user>
  <timeboxfinishuntil>2017-02-09T23:00Z</timeboxfinishuntil>
  <filterregexinclude/>
  <executablepath>/home/debian/git/ba/sampledata/ProgramSource/Builds/Saxon.jar</executablepath>
  <sourcerootpath>/home/debian/git/ba/sampledata/ProgramSource/Saxon</sourcerootpath>
  <topicsfilepath>/home/debian/git/ba/sampledata/SERE-Topicmap.xml</topicsfilepath>
  <trailsoutputpath>/home/debian/git/ba/sampledata/trail-output/generated/trail.xml</trailsoutputpath>
  <topic/>
  <prioritizationmode>WEIGHTING</prioritizationmode>
  <userprofile>JUNIOR_ENGINEER</userprofile>
  <knowledgelevel>STRANGER</knowledgelevel>
  <history>
    <trailstep actualvisittimestamp="2017-02-09T17:30:57.718Z" suggestedvisitlocation="" suggestedvisittimestamp=""><![CDATA[net.sf.saxon.tree.tiny.TinyBuilder:characters]]></trailstep>
    <trailstep actualvisittimestamp="2017-02-09T17:32:49.590Z" suggestedvisitlocation="" suggestedvisittimestamp=""><![CDATA[net.sf.saxon.event.ProxyReceiver:characters]]></trailstep>
  </history>
  <trail>
    <trailstep suggestedvisittimestamp="2017-02-09T17:34:00.785Z"><![CDATA[net.sf.saxon.tree.tiny.CompressedWhitespace:getCompressedValue]]></trailstep>
    <trailstep suggestedvisittimestamp="2017-02-09T17:34:00.785Z"><![CDATA[net.sf.saxon.tree.tiny.CompressedWhitespace:compress]]></trailstep>
    <trailstep suggestedvisittimestamp="2017-02-09T17:34:00.785Z"><![CDATA[net.sf.saxon.tree.util.FastStringBuffer:ensureCapacity]]></trailstep>
```

Figure 7. Example ViSiTR code trail XML format.

(example shown in Figure 6). The JSON was parsed, converted to FQNs, and placed in the H2 MethodRank table.

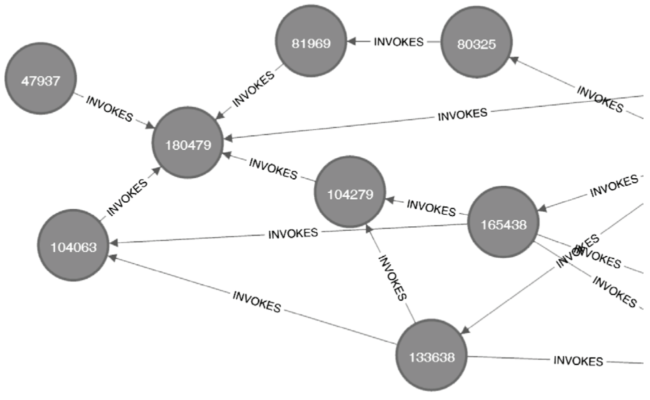
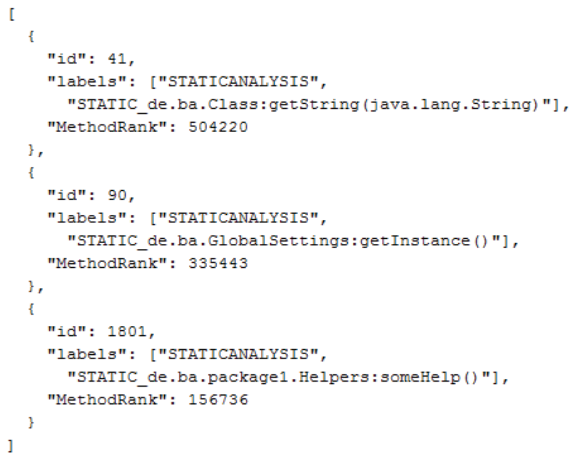


Figure 5. Example partial MethodRank graph in Neo4j.



```
[
  {
    "id": 41,
    "labels": ["STATICANALYSIS",
      "STATIC_de.ba.Class:getString(java.lang.String)"],
    "MethodRank": 504220
  },
  {
    "id": 90,
    "labels": ["STATICANALYSIS",
      "STATIC_de.ba.GlobalSettings:getInstance()"],
    "MethodRank": 335443
  },
  {
    "id": 1801,
    "labels": ["STATICANALYSIS",
      "STATIC_de.ba.package1.Helpers:someHelp()"],
    "MethodRank": 156736
  }
]
```

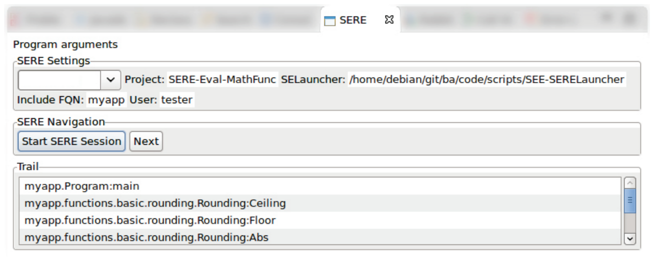
Figure 6. Example NodeRank request result in JSON.

Users are differentiated by a user ID. The visitation time is adjusted by a factor (default = 0.5) to halve the estimated time if it is a senior engineer, and a factor (0.5) also if the user is already familiar with the code. All user sessions are time-boxed (default setting is termination at midnight, but any end time can be set). Once the prioritized POI list is calculated, POIs are selected in priority order to be included in the trail until the accumulated expected visitation times exceed remaining session time. The Hamiltonian path calculation is then applied on this subset.

To order the POI trail according to POI locality, the Trail Planner component integrated OptaPlanner, specifically optimizing the trail with regard to the TSP. For sufficient IDE interaction responsiveness during trail generation, the OptaPlanner solving time was explicitly limited to a maximum of 5 seconds to likely provide sufficient time for at least a solution to be found (depending on the project size, session time, and computation hardware) but not necessarily an optimum (absolute shortest path).

Figure 7 shows a sample of the XML-based code trail that is sent to the ViSiTR client, with the tags explained as follows: `sessionguid` is a unique id for the session. `user` can be a unique username for tracking. `timeboxfinishuntil` is an absolute time for the expected end time for the session. `filterregexinclude` is a regular expression for the packages to be included, if nothing is provided then all are assumed. `executablepath` is the path to the project executable. `sourcerootpath` is the path to the root of the program source files. `topicsfilepath` is a path to the file that contains topics of interest, if it is empty then all topics are assumed. `trailsoutputpath` is the location of the trails file. `topic` provides a list of the topics if given, if empty then all topics are assumed. `prioritizationmode` provides the type of trail prioritization desired. `userprofile` indicates if the user is a junior or senior engineer (relates to how fast they may comprehend code). `knowledgelevel` relates to whether the programmer is a stranger or familiar with this code project. `history` tracks the `trailsteps` visited with the actual methods visited including the `actualvisittimestamp`. `trail` provides a list of the suggested `trailsteps` in the suggested order and with the `suggestedvisittimestamp` as an absolute time.

To demonstrate the REST-based integration capability of ViSiTR recommender service within common IDE tools, an Eclipse IDE client was developed, shown in Figure 8. The upper part shows the current project, the middle part is used for starting and navigating a session, and the bottom displays the upcoming trail locations (methods). Double-clicking causes the method to be shown in the Eclipse source view.



Project: SERE-Eval-MathFunc SELauncher: /home/debian/git/ba/code/scripts/SEE-SERELauncher
Include FQN: myapp User: tester

```
myapp.Program:main
myapp.functions.basic.rounding.Rounding:Ceiling
myapp.functions.basic.rounding.Rounding:Floor
myapp.functions.basic.rounding.Rounding:Abs
```

Figure 8. An Eclipse IDE ViSiTR client plugin showing a code trail retrieved from the ViSiTR service.

Code trail integration was accomplished using REST and JSON via a Unirest client invoked in a Groovy Script. The returned XML-based code trail was then parsed in the client.

B. *Visualization Client Implementation*

Existing software structures are imported and converted into a common XML-centric model. XML was selected as the primary data format to support the greatest amount of interoperability with various existing software development tools. BaseX [17] is used as an XML repository and XQuery used for queries. srcML [18] v.0.9.5 was selected to convert source code (such as Java) into XML documents and provides various code metrics. Campwood SourceMonitor v.3.5 is used because it creates code metrics across multiple programming languages. To determine dependencies such as coupling and inheritance, DependencyFinder was selected, which also provides data on code structure, dependencies, and metrics from binary Java. Furthermore, Groovy scripts were used for the integration of the various tools.

The project structure consists of the following files:

- *metrics_{date}.xml*: contains metrics obtained from tools such as SourceMonitor and DependencyFinder, which are grouped by project, packages, and classes.
- *source_{date}.xml*: holds all source code in the srcML XML format
- *structure_{date}.xml*: contains the project structure and dependencies, obtained from tools such as DependencyFinder.
- *swexplorer-annotations.xml*: contains user-based annotations with color, flag, and text including manual tags placed by a user and automatic tag patterns placed automatically where matches occur.
- *swexplorer-metrics-config.xml*: contains thresholds for metrics that support visual differentiation.
- *swexplorer-records.xml*: contains a record of each import of the same project done at different times with a reference to the various XML files such as source and structure for that import. This permits changing the model to different timepoints as a project evolves.

Additional HUD screens include: search, filtering (e.g., inclusion/exclusion of packages and classes), tagging, and a minimap (right corner) for orientation.

Minimum PC specifications are a CPU supporting Streaming SIMD Extensions 2 and DX9 GPU with Shader Model 2.0. Recommended is a DX11 GPU and 1GB video RAM. Java 7 and .NET Framework 3.5 or higher are required.

VI. CASE STUDY

In prior work [10], validation of the various learning models was performed: *M:Top-Down* and *M:Bottom-Up* utilizing the package hierarchy, *M:Topics/Goal* which utilizes filtering of packages and classes by names, *M:DynamicPath* which prioritizes methods that appear in various runtime traces by both how often (frequency) within a trace and that they occur within different trace files, and *weighted mode* that uses configurable parameter weighting inputs. Furthermore, the empirical study utilized program code obfuscation to limit any intuitive mental model creation or semantic ordering, assessing its effectiveness and efficiency for program comprehension knowledge navigation within unfamiliar program code (i.e., unfamiliar presented knowledge), while retaining the equivalent program structure.

This case study thus focuses on validating the viability of the 3D visualization solution and its scalability. For this study, two Java projects were used: the Saxon XSLT 2.0 and XQuery processor consisting of 331K lines of code (LOC), 17K methods, and 1655 classes in 38 packages with 53K inter-class dependencies. The ViSiTR client ran on a Fujitsu Lifebook AH531with Windows 10 Pro (x64) 2.4GHz i5-2430M 8GB RAM and SSD disk.

A. *ViSiTR client visualization*

In the universe metaphor, Figure 9 shows the loaded Saxon project consisting of 53 solar systems (without applying any filters to hide any packages or classes), and showing all dependencies. This can be navigated via 3D fly-thru and visual responsiveness for 3D fly-thru navigation showed no issues. In Figure 10, dependencies were deselected and solar systems become recognizable and distinguishable based on package names. In Figure 11, a single package, the net.sf.saxon package is shown as a solar system including internal package dependencies. Figure 12 shows the source code view in the HUD for the net.sf.saxon.Platform class (class name is labeled on a square in the middle of the planet, these are also used for tagging by stacking the squares in customizable colors) with the selected object in white. Planet orbits are in turquoise and dependencies as purple light beams. Planet size can vary based on some metric like number of methods.

Within the terrestrial metaphor, Figure 13 shows the loaded Saxon project with 53 packages represented by glass bubble cities viewed here from above (dependencies are hidden). Figure 14 shows the source code view in the HUD for the net.sf.saxon.Platform class (class name is labeled on a stackable square of tags on the top of any building), with classes in a package grouped within a glass city bubble and dependencies shown as purple pipes. If desired, building size can be set to vary based on some metric such as number of methods. Both metaphors were found to be easily navigable via 3D fly-thru, and scalability, performance, and responsiveness for the client showed no issues once the project was loaded.

Code trail navigation is shown in Figure 15 for the universe metaphor, with the current location shown on the trail strip above the cockpit menu. On the right of this strip, the current and upcoming two POIs are shown, and on the left, a play/pause button and a rewind and forward button are available for trail navigation. The slider on the left side center adjusts the speed with which one is transported between POIs by the automated trail guidance. Figure 16 shows the HUD in source code view with the terrestrial metaphor. Figure 17 shows the HUD in the UML view with a dynamically generated class diagram showing the dependencies between classes in 2D and those in 3D can be seen in the background.

B. *3D Code Trail Evaluation*

The performance and scalability of the ViSiTR prototype was measured. The ViSiTR service was run with VirtualBox version 5.1.14 in a virtual machine image of Debian 8 x86, single CPU, and 1.7GB RAM hosted on a Fujitsu Lifebook AH531with Windows 10 Pro (x64) 2.4GHz i5-2430M 8GB RAM and SSD disk. The host was also used to run the ViSiTR client.

To provide a contrast to the relatively large Saxon project, which includes two dynamic traces - one with 100K lines and the other with 17,497 lines, we also measured a custom small project called MathFunc consisting of 5 packages, 6 classes, 22 methods, and 37 dependencies without any trace input.

Table I compares the measured performance (wall clock time in seconds (s) or hours (h)) for various activities with the MathFunc and the Saxon project. Client-side project preparation, involving external tools and including code parsing, dependencies, and metrics was 15 secs for MathFunc and 300 secs for Saxon. Server-side project preparation was 400 secs for MathFunc and 6 hrs for Saxon. This project preparation time, which among other things involves source code parsing, dynamic trace analysis, and static graph call invocation analysis, is usually incurred once for stable projects. Client project loading delays on the Unity game engine were 5 secs for MathFunc and 220 secs for Saxon. All objects are created on initial loading before providing navigational capability. Trail creation, which involves TSP-based POI prioritization, was 15 secs for MathFunc and 110 secs for Saxon. Trail optimization, which sends a user event and requests a code trail optimization (adaptation) based on the data, was 12 secs for MathFunc and 65 secs for Saxon.

TABLE I. ACTIVITY PERFORMANCE

Activity	*MathFunc*	*Saxon*
Client-side project preparation	15s	300s
Server-side project preparation	400s	6h
Client project loading latency	5s	220s
Trail creation	15s	110s
Trail optimization	12s	65s

In our previous empirical study in C-TRAIL [10] using obfuscated code, we had focused on small projects to support reconstruction of the code structure to avoid straining cognitive abilities. For larger projects using code trails, ViSiTR performance showed the code trail service to be the primary bottleneck both in preparation and at runtime. While this service was run locally on the notebook, the service could instead be placed in the cloud to utilize more powerful hardware and reduce the 6h preparation time.

While we were able to successfully prototype the code trail recommendation with 3D visualization, larger projects created noticeable performance issues, although not in the visualization but rather in the recommendation service. In future work, we plan to address the initial prototype's performance issues via profiling, platform tuning, algorithm optimizations, dedicated server hardware for the service, and enabling background loading of visual objects on the client for very large projects. These are needed to enable a comprehensive empirical study to determine acceptance and improved comprehension by programmers.

Understanding the project requires a programmer to visualize the overall structure in abstractions in their mind; UML also requires some cognitive processing within its metaphor of boxes and lines. Even if visual metaphors provide an additional cognitive burden requiring further processing, in our opinion based on results from our previous study they also provide an additional motivation incentive that can offset this burden for various user groups (such as students) and keep them interested in the project code longer while still viewing real source code.

VII. CONCLUSION AND FUTURE WORK

This paper described the ViSiTR approach to code trail visualization, describing its theoretical background in the holey quilt theory and cognitive learning models. The solution concept was described and implementation details of a prototype provided. A case study evaluated its viability for code trail visualization and its scalability for larger projects.

As an automated tutor and recommender system in the program code comprehension space, ViSiTR applies a conceptual mapping of geographical POIs to code locations, considers the locality or knowledge closeness of such granules, and applies TSP to an unfamiliar knowledge landscape consisting of program code. It incorporates MethodRanking as a variant of PageRanking and granular distance in the form of POI locality. Furthermore, it recommends a knowledge navigation order by generating a code trail as a Hamiltonian cycle. While the ViSiTR prototype showed the feasibility and viability of 3D visual code trails, the evaluation also showed that optimizations of the prototype implementation are needed to improve its scalability and permit empirical studies with larger projects.

Our approach does not consider extraneous artifacts related to program comprehension, such as configuration files and documentation, since these typically must be analyzed and provided by humans. In future work we will consider providing a way to include this information.

Future work includes prototype performance and scalability optimization and testing with workstations, a comprehensive empirical study with different projects and user groups, support for additional programming languages, support for displaying different and directed relationship categories and cardinalities, and additional visualization paradigms. Application of the elaborated ViSiTR solution principles to other domains beyond software engineering could provide beneficial knowledge navigation guidance and recommendations in form of a trail for other unfamiliar knowledge landscapes.

ACKNOWLEDGMENT

The author thanks Claudius Eisele for his assistance with the implementation of the service and Dominik Bergen for the client implementation and evaluation.

REFERENCES

[1] R. Oberhauser, "ReSCU: A Trail Recommender Approach to Support Program Code Understanding," Proceedings of the Eighth International Conference on Information, Process, and Knowledge Management (eKNOW 2016). IARIA XPS Press, 2016, pp. 112-118.

[2] G. Booch, "The complexity of programming models," keynote talk at AOSD 2005, Chicago, IL, March 14-18, 2005.

[3] A. Deshpande and D. Riehle. “The total growth of open source,” In: IFIP International Federation for Information Processing. Vol. 275. 2008, pp. 197–209.

[4] C. Jones, "The economics of software maintenance in the twenty first century," Version 3, 2006. [Online]. Available from: http://www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf 2017.02.23

[5] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer: an investigation of how developers spend their time," Proc. IEEE 23rd International Conference on Program Comprehension, IEEE Press, 2015, pp. 25-35.

[6] L. Kappelman, "Some strategic Y2K blessings," Software, IEEE, 17(2), 2000, pp. 42-46.

[7] PayScale, "Full List of Most and Least Loyal Employees." [Online]. Available from: http://www.payscale.com/data-packages/employee-loyalty/full-list 2017.02.23

[8] M. Robillard, W. Maalej, R. Walker, and T. Zimmermann, Recommendation Systems in Software Engineering. Springer, 2014.

[9] F. P. Brooks, Jr., The Mythical Man-Month. Boston, MA: Addison-Wesley Longman Publ. Co., Inc., 1995.

[10] R. Oberhauser, "C-TRAIL: A Program Comprehension Approach for Leveraging Learning Models in Automated Code Trail Generation," Proceedings of the 11th International Conference on Software Engineering and Applications (ICSOFT-EA 2016), SciTePress, 2016, pp. 177-185.

[11] R. Oberhauser, C. Silfang, and C. Lecon, "Code structure visualization using 3D-flythrough," Proc. of the 11th International Conference on Computer Science & Education (ICCSE), IEEE, 2016, pp. 365-370.

[12] T. Clear, "The hermeneutics of program comprehension: a 'holey quilt' theory," ACM Inroads, 3(2), June 2012, pp.6-7.

[13] A. Bargiela and W. Pedrycz, Granular computing: an introduction. Springer Science & Business Media, vol. 717, 2012.

[14] Y. Qian, J. Liang, C. Dang, F. Wang, and W. Xu, "Knowledge distance in information systems," J. of Systems Science and Systems Engineering, 16(4), 2007, pp. 434-449.

[15] M. Rahman and M. Kaykobad, "On Hamiltonian cycles and Hamiltonian paths," Information Processing Letters, 94(1), 2005, pp. 37-41.

[16] E. Lawler, J. Lenstra, A. Kan, and D. Shmoys. The traveling salesman problem: a guided tour of combinatorial optimization. Wiley, New York, 1985.

[17] J. Singer, R. Elves, and M.-A. Storey, "NavTracks: Supporting Navigation in Software Maintenance," Proc. Int'l Conf. on Software Maintenance, 2005, pp. 325–334.

[18] M. Kersten and G. Murphy, "Mylar: A degree-of-interest model for IDEs," Proc. 4th international conf. on aspect-oriented software development, ACM, 2005, pp. 159-168.

[19] D. Cubranic, G. Murphy, J. Singer, and K. Booth, "Hipikat: A project memory for software development," Software Eng., IEEE Trans. on, 31(6), 2005, pp. 446-465.

[20] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," Software Eng., IEEE Trans. on, 31(6), 2005, pp. 429-445.

[21] M. Robillard and G. Murphy, "FEAT: A tool for locating, describing, and analyzing concerns in source code," Proc. 25th Int'l Conf. on Software Eng., IEEE, 2003, pp. 822–823.

[22] M. Robillard and G. Murphy, "Automatically Inferring Concern Code from Program Investigation Activities," Proc. 18th Int'l Conf. Autom. SW Eng., IEEE, 2003, pp. 225-234.

[23] M. Robillard, “Topology Analysis of Software Dependencies,” ACM Trans. Software Eng. and Methodology, vol. 17, no. 4, article no. 18, 2008.

[24] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman, "Mining Jungloids: Helping to Navigate the API Jungle," Proceedings of PLDI, Chicago, IL, 2005, pp. 48–61.

[25] R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," IEEE Transactions on Software Engineering 32(12), 2006, pp. 952–970.

[26] M. Bruch, T. Schaefer, and M. Mezini, "Fruit: IDE support for framework understanding," Proc. 2006 OOPSLA workshop on eclipse technology eXchange, eclipse ’06, ACM, 2006, pp. 55–59.

[27] M. Goldman and R. C. Miller, "Codetrail: Connecting source code and web resources," Journal of Visual Languages & Computing, 20(4), 2009, pp.223-235.

[28] M. Yin, B. Li, and C. Tao, "Using cognitive easiness metric for program comprehension," Proc. 2nd Int. Conf. on Softw. Eng. and Data Mining, IEEE, 2010, pp. 134-139.

[29] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," Softw. Eng., IEEE Trans. on, 35(5), 2009, pp.684-702.

[30] A. Teyseyre and M. Campo, "An overview of 3D software visualization," Visualization and Computer Graphics, IEEE Transactions on, vol. 15, no. 1, (2009, pp. 87-105.

[31] A. Kashcha, "Software Galaxies." [Online]. Available: http://github.com/anvaka/pm/ 2017.02.23

[32] R. Wettel and M. Lanza, “Program comprehension through software habitability,” in Proc. 15th IEEE Int'l Conf. on Program Comprehension, IEEE CS, 2007, pp. 231–240.

[33] R. Wettel et al., "Software systems as cities: A controlled experiment," in Proc. of the 33rd Int'l Conf. on Software Engineering, ACM, 2011, pp. 551-560.

[34] J. Rilling and S. P. Mudur, "On the use of metaballs to visually map source code structures and analysis results onto 3d space," in Proc.. 9th Work. Conf. on Reverse Engineering, IEEE, 2002, pp. 299-308.

[35] P. McIntosh, "X3D-UML: user-centred design, implementation and evaluation of 3D UML using X3D," Ph.D. dissertation, RMIT University, 2009.

[36] A. Krolovitsch and L. Nilsson, "3D Visualization for Model Comprehension: A Case Study Conducted at Ericsson AB," University of Gothenburg, Sweden, 2009.

[37] C. Schulte, "Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching," Proc. Fourth International Workshop on Computing Education Research, ACM, 2008, pp. 149-160.

[38] C. Schulte, T. Busjahn, T. Clear, J. Paterson, and A. Taherkhani, "An introduction to program comprehension for computer science educators," Proc. 2010 ITiCSE Working group reports (ITiCSE-WGR ‘10), ACM, 2010, pp. 65-86.

[39] J. Novak. Learning, creating, and using knowledge. Lawrence Erlbaum Assoc., Mahwah, NJ, 1998.

[40] S. Letovsky, "Cognitive processes in program comprehension," Journal of Systems and Software, 7(4), 1987, pp. 325-339.

[41] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," Cognitive psychology, 19(3), 1987, pp.295-341

[42] E. Soloway, B. Adelson, and K. Ehrlich, "Knowledge and processes in the comprehension of computer programs," In: The Nature of Expertise, A. Lawrence Erlbaum Associates, 1988, pp. 129-152

[43] J. Koenemann and S. Robertson, "Expert problem solving strategies for program comprehension," Proc. of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 1991, pp. 125-130.

[44] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," Softw. Eng., IEEE Trans. on, 35(5), 2009, pp.684-702

[45] Y. Guéhéneuc, "TAUPE: towards understanding program comprehension," Proc. 2006 conf. Center for Adv. Studies on Collaborative research (CASCON '06) IBM Corp., 2006.

[46] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: bringing order to the Web," In:World Wide Web Internet And Web Information Systems 54.1999-66, 1998, pp. 1–17.

[47] GraphAware. Neo4j NodeRank. [Online]. Available from: https://github.com/graphaware/neo4j-noderank 2017.02.23

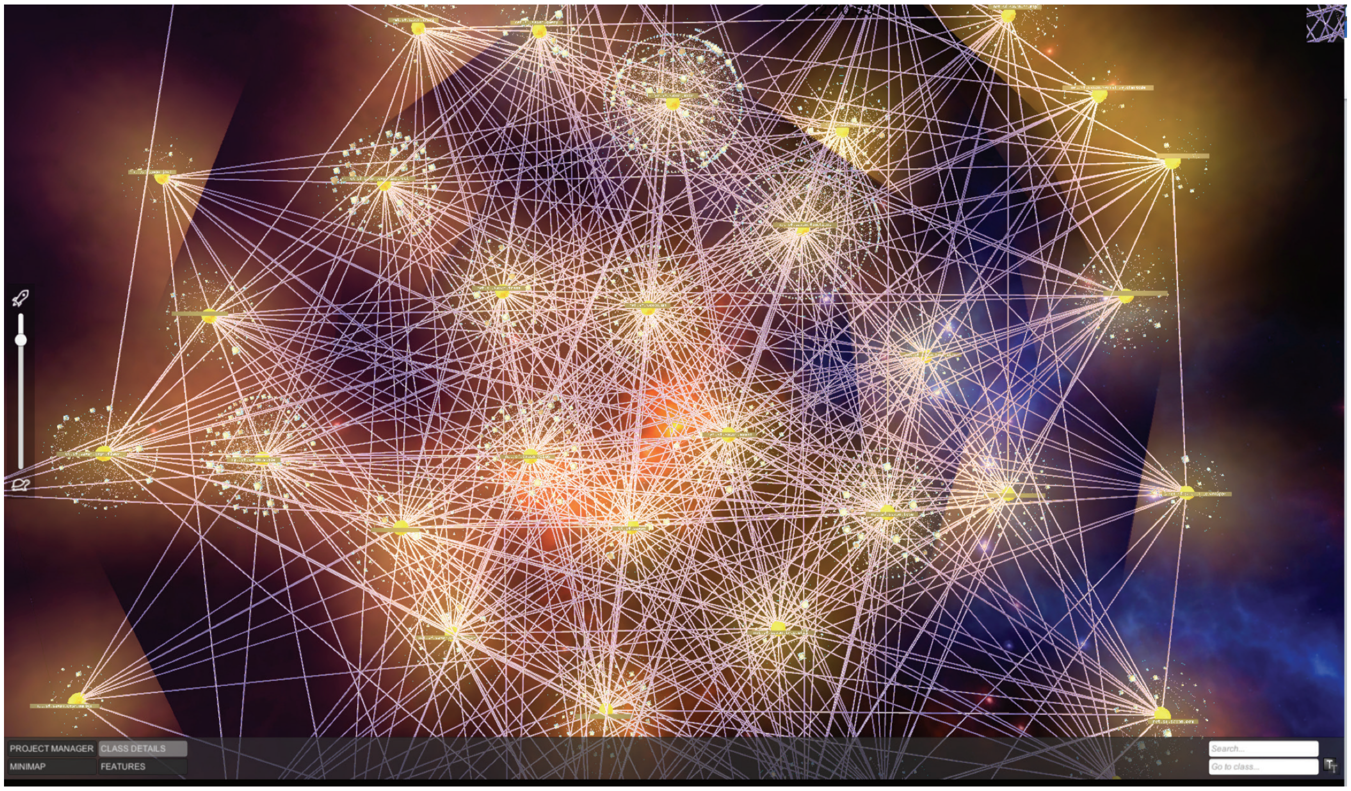


Figure 9. Saxon project with dependencies shown in the ViSiTR universe metaphor.

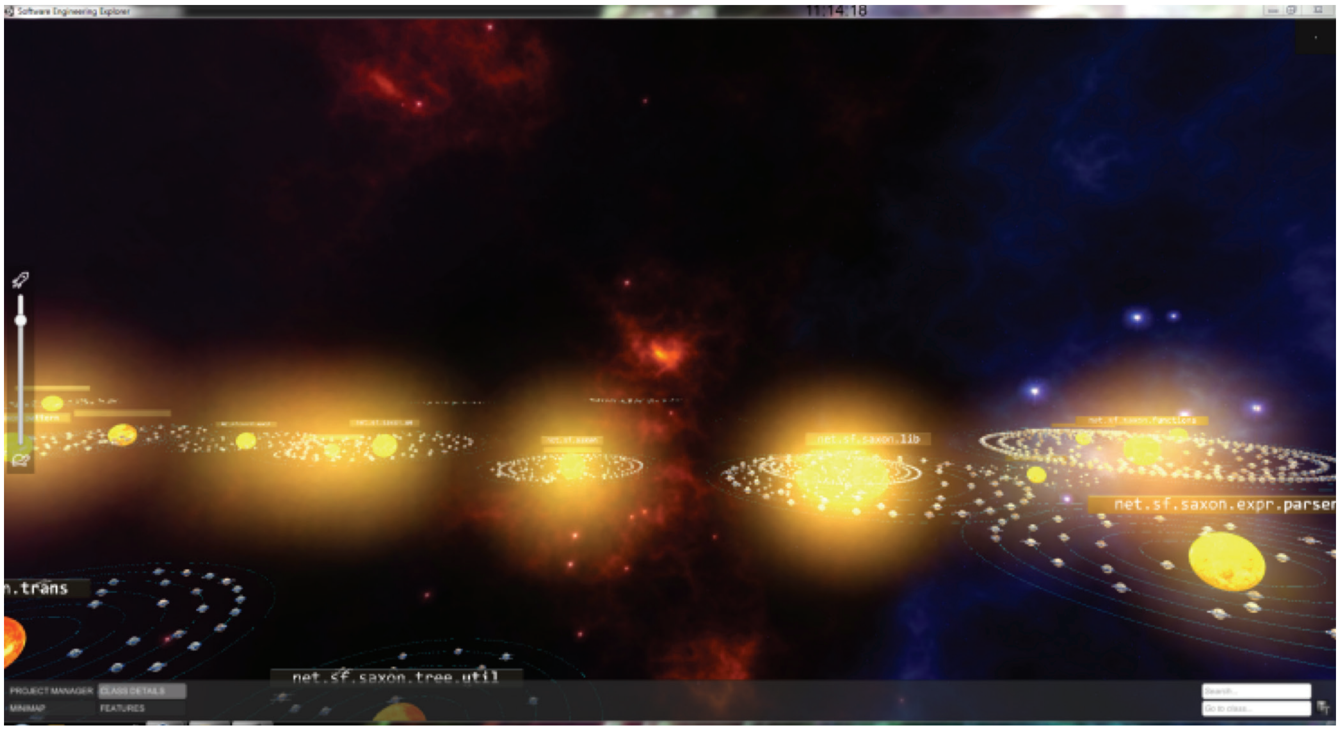


Figure 10. Saxon project without dependencies shown in the ViSiTR universe metaphor.

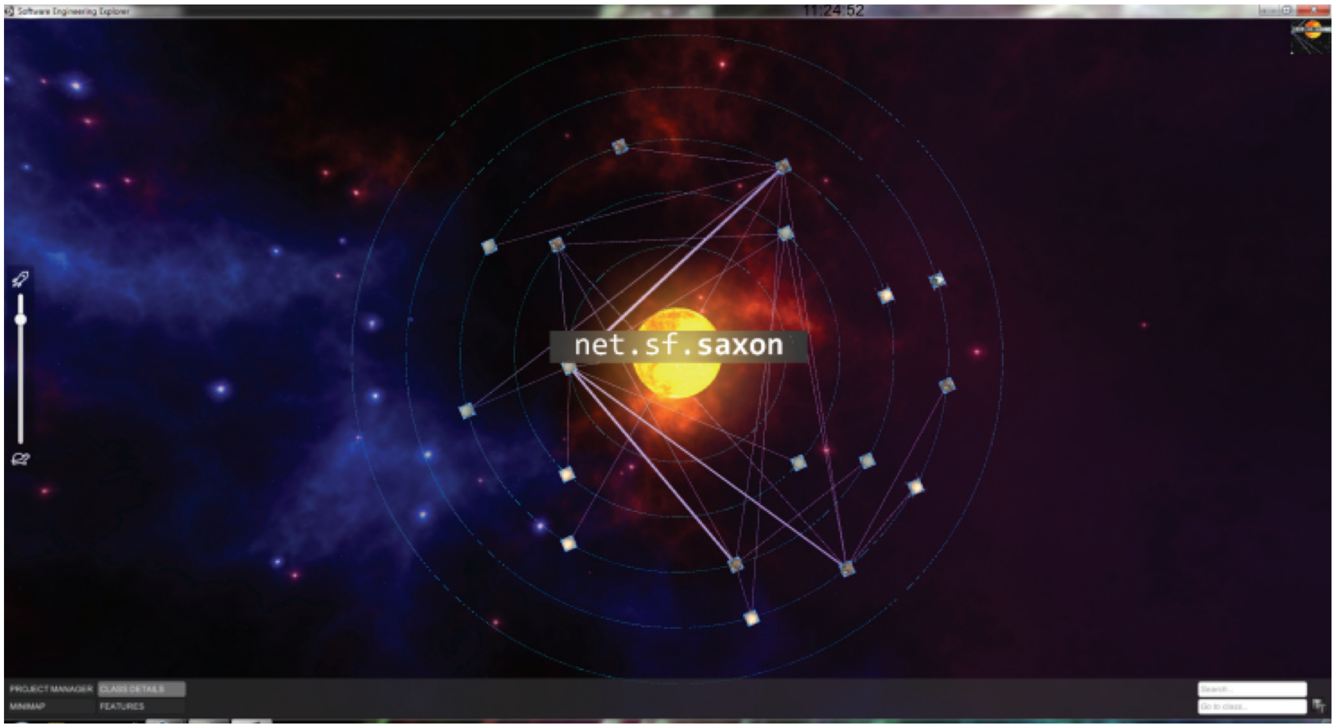
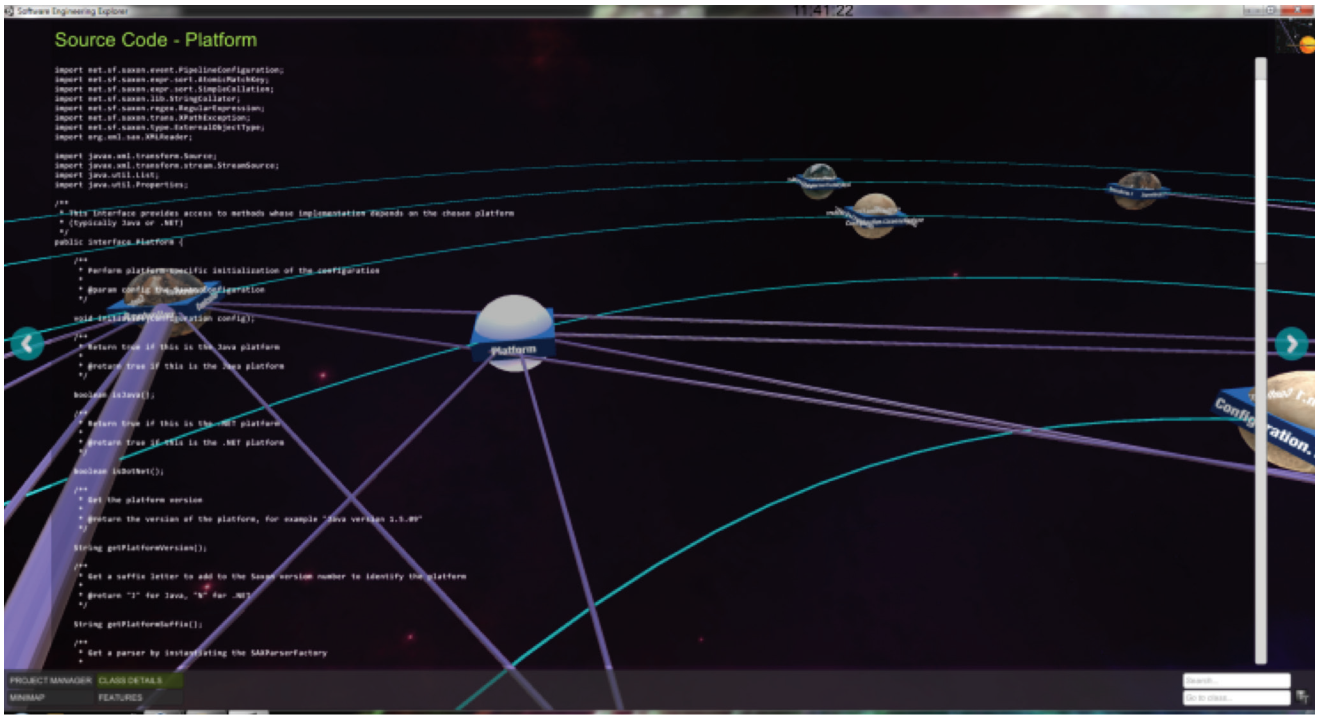


Figure 11. The net.sf.saxon package shown as an isolated solar system with internal dependencies in the ViSiTR universe metaphor.



Source Code - Platform

```
import net.sf.saxon.event.PipelineConfiguration;
import net.sf.saxon.expr.sort.AtomicMatchKey;
import net.sf.saxon.expr.sort.SimpleCollation;
import net.sf.saxon.lib.StringCollator;
import net.sf.saxon.regex.RegularExpression;
import net.sf.saxon.trans.XPathException;
import net.sf.saxon.type.ExternalObjectType;
import org.xml.sax.XMLReader;

import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import java.util.List;
import java.util.Properties;

/**
 * This interface provides access to methods whose implementation depends on the chosen platform
 * (typically Java or .NET)
 */
public interface Platform {

    /**
     * Perform platform-specific initialization of the configuration
     *
     * @param config the Saxon Configuration
     */

    void initialize(Configuration config);

    /**
     * Return true if this is the Java platform
     *
     * @return true if this is the Java platform
     */

    boolean isJava();

    /**
     * Return true if this is the .NET platform
     *
     * @return true if this is the .NET platform
     */

    boolean isDotNet();

    /**
     * Get the platform version
     *
     * @return the version of the platform, for example "Java version 1.5.09"
     */

    String getPlatformVersion();

    /**
     * Get a suffix letter to add to the Saxon version number to identify the platform
     *
     * @return "J" for Java, "N" for .NET
     */

    String getPlatformSuffix();

    /**
     * Get a parser by instantiating the SAXParserFactory
     *
```

Figure 12. HUD source code view of the saxon Platform class as a planet with dependencies in the ViSiTR universe metaphor.

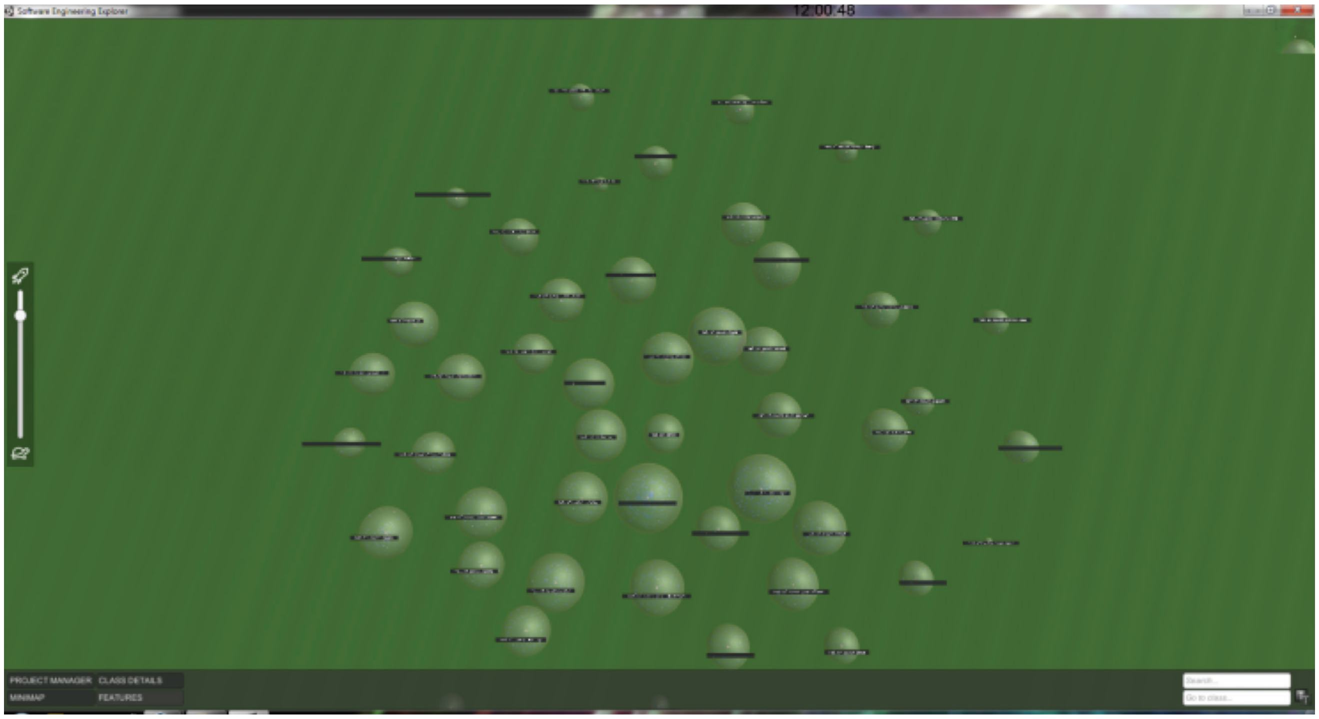
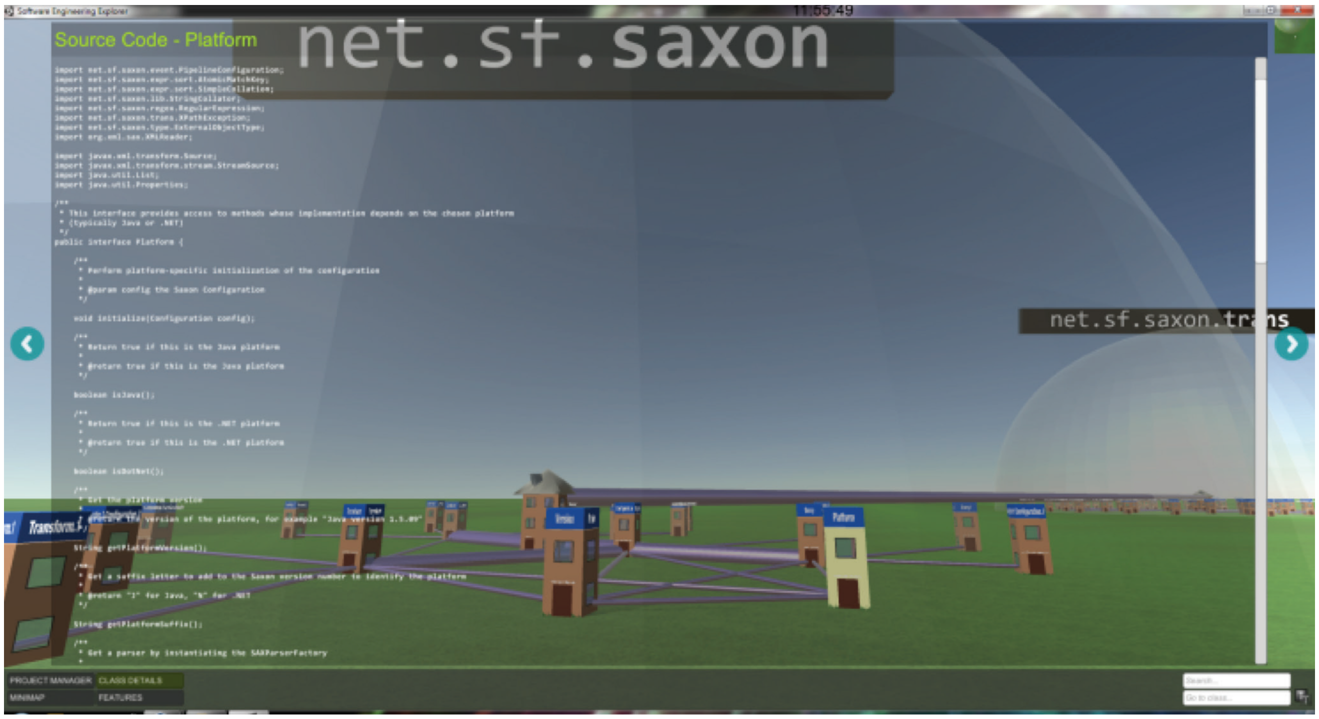


Figure 13. The Saxon project with dependencies shown in the ViSiTR terrestrial metaphor.



```
import net.sf.saxon.event.PipelineConfiguration;
import net.sf.saxon.expr.sort.AtomicMatchKey;
import net.sf.saxon.expr.sort.SimpleCollation;
import net.sf.saxon.lib.StringCollator;
import net.sf.saxon.regex.RegularExpression;
import net.sf.saxon.trans.XPathException;
import net.sf.saxon.type.ExternalObjectType;
import org.xml.sax.XMLReader;

import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import java.util.List;
import java.util.Properties;

/**
 * This interface provides access to methods whose implementation depends on the chosen platform
 * (typically Java or .NET)
 */
public interface Platform {

    /**
     * Perform platform-specific initialization of the configuration
     *
     * @param config the Saxon Configuration
     */

    void initialize(Configuration config);

    /**
     * Return true if this is the Java platform
     *
     * @return true if this is the Java platform
     */

    boolean isJava();

    /**
     * Return true if this is the .NET platform
     *
     * @return true if this is the .NET platform
     */

    boolean isDotNet();

    /**
     * Get the platform version
     *
     * @return the version of the platform, for example "Java version 1.5.09"
     */

    String getPlatformVersion();

    /**
     * Get a suffix letter to add to the Saxon version number to identify the platform
     *
     * @return "J" for Java, "N" for .NET
     */

    String getPlatformSuffix();

    /**
     * Get a parser by instantiating the SAXParserFactory
     *
```

Figure 14. HUD source code of saxon Platform class as building in glass city bubble with dependencies in ViSiTR terrestrial metaphor.

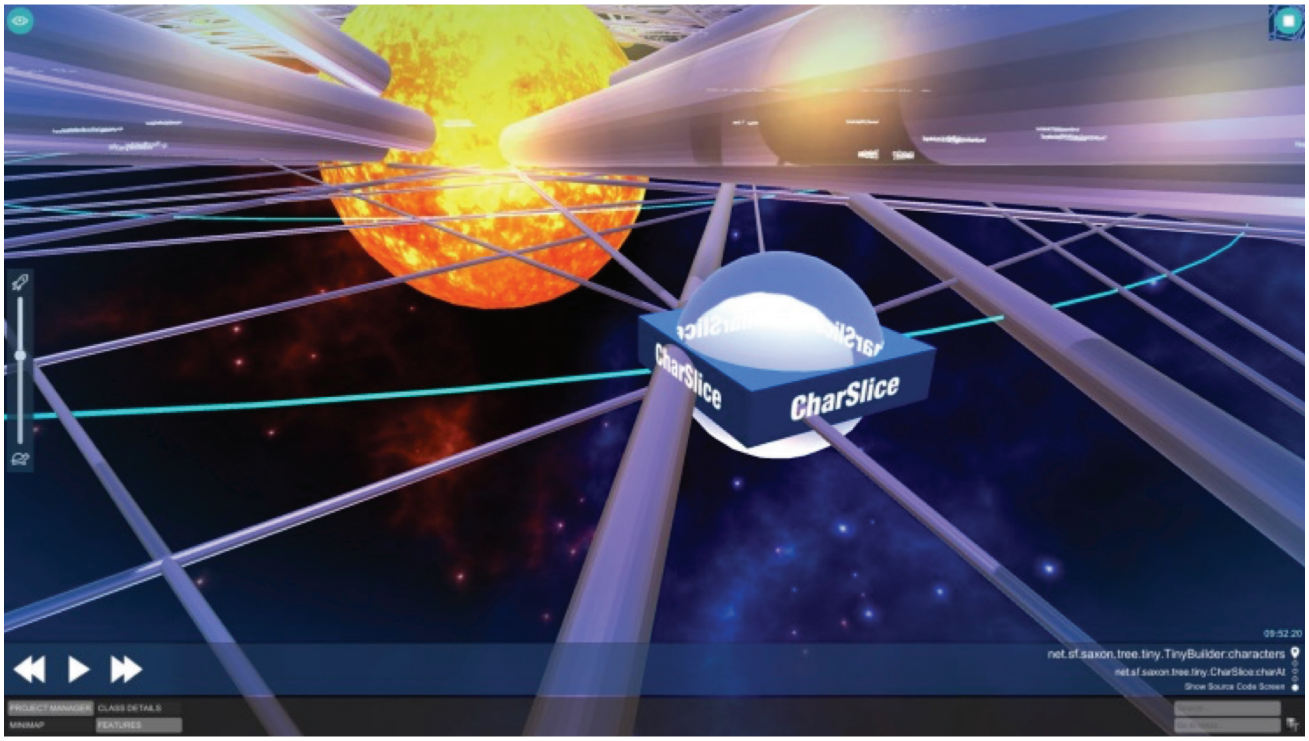
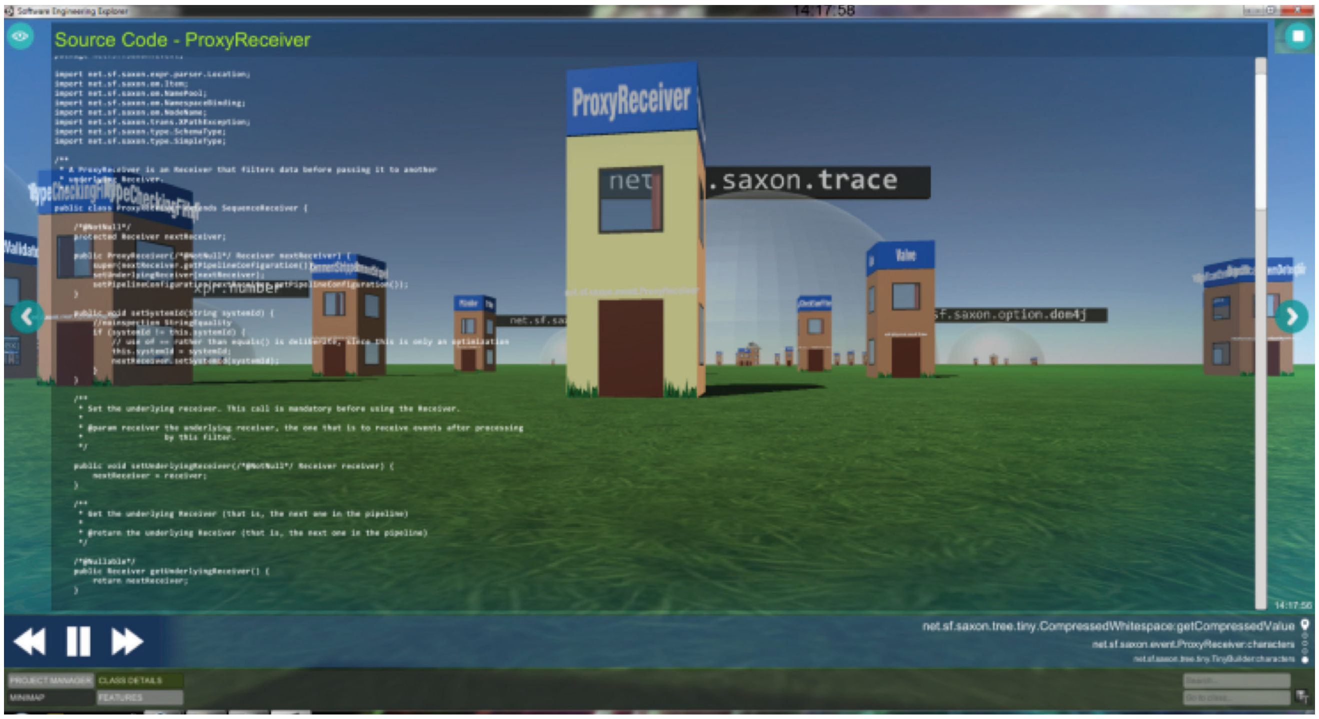


Figure 15. Visitation of CharSlice during automated code trail navigation in the ViSiTR universe metaphor.



```
import net.sf.saxon.expr.parser.Location;
import net.sf.saxon.om.Item;
import net.sf.saxon.om.NamePool;
import net.sf.saxon.om.NamespaceBinding;
import net.sf.saxon.om.NodeName;
import net.sf.saxon.trans.XPathException;
import net.sf.saxon.type.SchemaType;
import net.sf.saxon.type.SimpleType;

/**
 * A ProxyReceiver is an Receiver that filters data before passing it to another
 * underlying Receiver.
 */

public class ProxyReceiver extends SequenceReceiver {

    /*@NotNull*/
    protected Receiver nextReceiver;

    public ProxyReceiver(/*@NotNull*/ Receiver nextReceiver) {
        super(nextReceiver.getPipelineConfiguration());
        setUnderlyingReceiver(nextReceiver);
        setPipelineConfiguration(nextReceiver.getPipelineConfiguration());
    }

    public void setSystemId(String systemId) {
        //noinspection StringEquality
        if (systemId != this.systemId) {
            // use of == rather than equals() is deliberate, since this is only an optimization
            this.systemId = systemId;
            nextReceiver.setSystemId(systemId);
        }
    }

    /**
     * Set the underlying receiver. This call is mandatory before using the Receiver.
     *
     * @param receiver the underlying receiver, the one that is to receive events after processing
     *                 by this filter.
     */

    public void setUnderlyingReceiver(/*@NotNull*/ Receiver receiver) {
        nextReceiver = receiver;
    }

    /**
     * Get the underlying Receiver (that is, the next one in the pipeline)
     *
     * @return the underlying Receiver (that is, the next one in the pipeline)
     */

    /*@Nullable*/
    public Receiver getUnderlyingReceiver() {
        return nextReceiver;
    }
```

```
net.sf.saxon.tree.tiny.CompressedWhitespace:getCompressedValue
net.sf.saxon.event.ProxyReceiver:characters
net.sf.saxon.tree.tiny.TinyBuilder:characters
```

Figure 16. Visitation of ProxyReceiver showing HUD source code view during automated code trail navigation in the ViSiTR terrestrial metaphor.

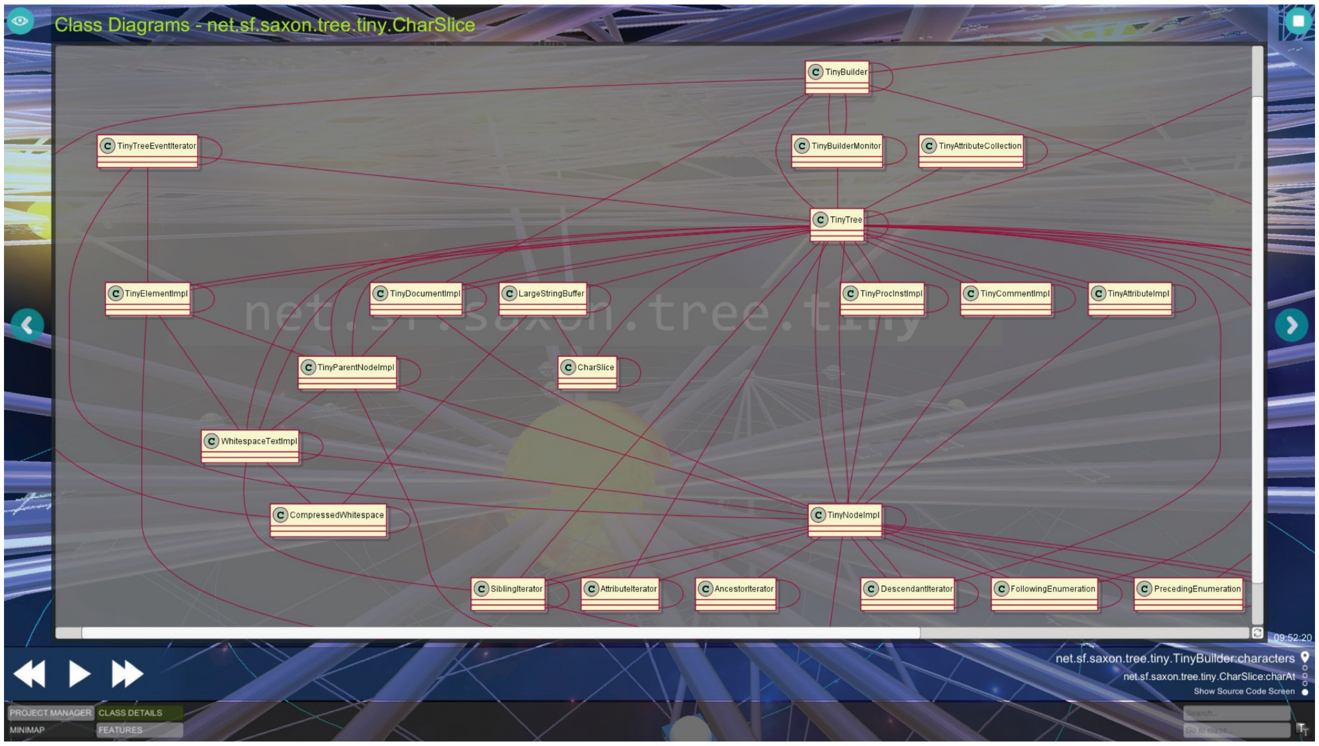


Figure 17. Dynamically generated UML class diagram showing dependencies for CharSlice during code trail visit in the ViSiTR universe metaphor.