Utilising a Geographic Space Metaphor in a Software Development Environment

Henrik Bærbak Christensen

*Centre for Experimental System Development, Department of Computer Science,
University of Aarhus, DK-8000 Århus C, Denmark, hbc@daimi.au.dk*

Abstract: Current software systems are growing ever larger and more complex. This increase in size and complexity is of course reflected in the data produced in the development process and overviewing and navigating the structure are becoming daunting tasks. This paper presents a model for visualising many important aspects of the complex, multi-dimensional, data in a software development project. The model is based on a *geographic space metaphor*: Entities in the software architecture are organised geographically in what becomes a design 'landscape'. Thereby overview and navigation is supported by tapping into humans fine spatial and visual perception. The landscape is shared in the project team, and mediates daily development activities thereby providing a common reference frame, and a framework for visualising aspects of the software. A prototype implementation of a software development environment, Ragnarok, based on this model is presented and some preliminary experiences outlined.

Keywords: Information Visualisation and Navigation, User Interface Design, Spatial Metaphors, Computer Supported Cooperative Work, Software Development Environments.

1. INTRODUCTION

One problem facing managers and developers alike in large software development projects is the complexity and sheer amount of data produced during the development process. As systems grow ever larger it becomes increasingly difficult to maintain overview and navigate in the many aspects and data of the system.

One of the possible causes of this difficulty was pointed out by Brooks in his famous article 'No Silver Bullet—Essence and Accidents of Software Engineering' where Brooks states *invisibility* as an inherent property of software (Frederick P. Brooks, Jr., 1987): The multi-dimensional nature of software does not easily lend itself to a

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35349-4_22](https://doi.org/10.1007/978-0-387-35349-4_22)

S. Chatty et al. (eds.), *Engineering for Human-Computer Interaction*

© IFIP International Federation for Information Processing 1999

single 2D or 3D diagrammatic form and thereby deprives us one of our most powerful conceptual tools: Our visual and spatial perception.

In this paper we try to show, however, that many interesting dimensions of software and data from the development process can still be visualised beneficially using a single metaphor: A geographic space metaphor (Kuhn and Blumenthal, 1996). Our proposal is to use the logical design structure (Lamb, 1996; Sommerville, 1992) of a software system as basis and organise it geographically in a two-dimensional space: The abstractions and hierarchy of a software design are represented by *landmarks* having stable positions, sizes, and appearances in a plane thereby creating a manifest ‘design landscape’. These landmarks are directly manipulable: Daily development tasks are mediated through landmarks. Different dimensions (aspects) of the design entities like e.g. the source code files implementing it, documentation, staffing- and budget information, profiling information, version- and configuration control, etc., are visualised by processing the associated data appropriately and control the visual appearance of landmarks based on this processing; just like ordinary maps may show different aspects like vegetation, roads, elevation, or population density, of the same region of a country.

The approach has some inherent limitations. The data and aspects handled must have a one-to-one mapping to design abstractions/landmarks and a 2D visualisation must be natural. For instance, the planning phase of a project where tasks are best organised on a time line, cannot be handled well by this approach. Still enough interesting aspects exist to make the presented approach worth while.

The visual model is implemented in a prototype software development environment, Ragnarok, which currently supports basic handling and visualisation of a handful of aspects. Some preliminary experiences with Ragnarok will be reported.

The motivation for Ragnarok’s visual model is described in section 2. In section 3, the Ragnarok model of a software architecture is briefly outlined followed by a description of the visual model in section 4. The Ragnarok prototype implementation is described in section 5. Section 6 discusses the benefits of using a geographic space metaphor in software development. Finally related work, future work, and conclusion are described in sections 7–9.

2. MOTIVATION

The problems that motivated the Ragnarok visual model are:

- *Overview and navigation:* Overviewing large software systems and finding the correct piece of code in the thousands of files and libraries, is becoming a daunting task even in systems with a sound logical design. Explaining the design to newcomers is also problematic (Bürkle et al., 1995). We believe the problem is not that the software structure is not sufficiently *logical* but that it is not sufficiently *physical*. The goal in Ragnarok is to make software tangible and manifest by associating software entities with stable positions.
- *Mediating, up-to-date, software design:* The majority of software systems are still implemented in terms of source code files—and files does not convey much information about the design of a software system. Though strong graphical design notations exists, like OMT and UML (Rumbaugh et al., 1991; Rational Rose, 1997), these are

mainly used in the analysis, design, and initial implementation phase with a tendency that the diagrams becomes outdated in later implementation and maintenance phases. The goal is to use graphical notation to make a software design that is used *actively* to perform daily development tasks.

- *Collaboration*: Software development is a team effort today. Therefore a common understanding and reference frame is essential to allow the software design to be discussed, documented, and reused. The goal is to provide a shared landscape that provides a common reference frame and secondly let the landscape visualise development progress.
- *Visualisation framework*: Data in a software project is inherently multi-dimensional: One dimension is the source files; another the budgets, task-lists, and staffing information; yet another version control, etc. Traditionally these different dimensions are handled by different tools and organisational procedures and summary information presented in different formats: Lists, tables, graphs, etc. The goal is to provide a stable, unifying visualisation framework that allows diverse information to be visualised overlaid over the design landscape.

The means to achieve these goals is to organise the software design entities geographically using a geographic space metaphor. One distinguishes between *desktop- (small scale)* and *geographic (large scale) spaces* in spatial metaphors (Kuhn and Blumenthal, 1996; Egenhofer and Mark, 1995). The distinction comes from everyday experience: Objects in a desktop space have sizes comparable to the human body and can readily be moved and turned, whereas objects in geographic space are beyond the human body and have fixed positions over a long time-scale, like for instance buildings, trees, streets, and so forth.

Humans are apt at navigating in a well known physical environment: As Kuhn and Blumenthal notes: ‘Perception, manipulation, and motion in space are largely subconscious activities that impose little cognitive load while offering powerful functionality’ (Kuhn and Blumenthal, 1996). By providing a geographical layout of design abstractions, we believe that humans fine sense of locality can be utilised.

3. SOFTWARE ARCHITECTURE MODEL

Abstraction and hierarchy are key concepts in designing, building, testing, and managing large software systems. Large software systems are decomposed into a hierarchy of abstractions, where each entity in this hierarchy is an abstraction of the underlying parts. The resulting structure is often termed the software architecture (Lamb, 1996) or the logical design structure (Sommerville, 1992). Traditionally such abstractions are termed systems, subsystems, libraries, modules, classes, or class categories, depending on the type and granularity of the abstraction and its position in the hierarchy. In Figure 1, such a hierarchical decomposition is exemplified.

In the Ragnarok software architecture model, an abstraction in the software design is represented by an object denoted a **software component**. A software component contains **substance** which is a physical manifestation of the abstraction; currently implemented as a set of files. A typical example is the abstraction of a class which is represented by a software component whose substance is an interface- and implementation file for the class.

Figure 1: Example of a logical design structure. A box represents an abstraction/component and the lines between boxes are composition (part/whole) relationships.

An abstraction is often composed of other abstractions in the software architecture, for instance a library is a composition of individual modules. This is modelled in the software component by **composition relations** (part/whole) to other software components, like library B2 in Figure 1 that is composed of classes U, V, and T.

Finally a software component has a set of **annotations**. Each annotation contains structured data for a specific dimension/aspect of the component. Examples include: Managerial annotations (staffing: Who is responsible for implementing this component; budget: How many staff hours are budgeted for implementation, how many have been spent so far; estimated-time-to-complete etc.), quality assurance annotations (checklists to be gone through in release situation, regression test suits), progress logs (what bug-fixes/enhancements have been carried out, by whom and when), etc.

4. VISUAL MODEL

In this section the basic concepts in the Ragnarok visual model will be presented.

The **landscape** is an infinite two dimensional plane. The landscape serves as a space for geographically organising **landmarks** and **decorations**.

A landmark occupies a well defined region of the landscape and represents a software component (and thus an abstraction in the software design). The composition (part/whole) structure is visualised by *spatial containment* i.e. the landmarks associated with components that are part of a component *A* are positioned inside the landmark of *A*. In Figure 2 a legal layout for the design structure in Figure 1 is shown.

A decoration is simple graphics, like text, lines, polygons, images, etc., at a specific position in the landscape. Specifically the Ragnarok prototype provides basic support for UML notation, the unified modelling language (Rational Rose, 1997). Thereby the software design can be further documented by stating associations, multiplicity, roles, etc., as seen on Figure 3 in section 5.

The landscape is not directly accessible but viewed and manipulated through **maps**. A map visualises a region of the landscape on the computer display. The region displayed is determined by the map's *view-parameters*: (*O*, *w*, *h*, *s*, *A*). *O* is the position

Figure 2: Example of legal landscape of landmarks for the components in Figure 1.

of the map’s top left pixel projected onto the landscape, w and h the map’s physical (pixel) width and height, s the map’s scale, and A the map’s aspect. Hence a map w pixels wide and h pixels tall will display region $(O.x, O.y, O.x + ws, O.y + hs)$ of the landscape. This is denoted the *displayed region*. The landmarks appearing in a map are denoted *visual landmarks* when we explicitly want to distinguish them from the landmarks in the landscape: There is only one landmark in the landscape for each component but there may be several maps displaying different visual landmarks for the same underlying landmark.

The **aspect** A determines the appearance of visual landmarks in the displayed region by (processing) a subset of the data in the associated software component. For instance, in a management aspect map the management annotation data of a component may be processed to yield a colour code for estimated-time-to-complete which is then used as the background colour for the visual landmark.

4.1 Global context

Any number of maps can be created showing regions of the landscape in various aspects. However, a large number of maps does not in itself provide overview nor global context: How are the displayed regions positioned relative to each other? Therefore the Ragnarok visual model introduces a **map outline**. A map outline is a projection of the displayed region of one map (**detail** map) onto another map dedicated for showing global context (**world** map). To distinguish outlines in the world map, a given outline and the frame of the corresponding detail map have identical colours. The outline and the displayed region in the corresponding map are synchronised so any change in either of them is immediately reflected in the other.

4.2 Interaction

The interaction model employed is *direct manipulation* (Shneiderman, 1983; Hutchins et al., 1986).

Landmarks can be moved and resized directly using the mouse. A move may reflect an architectural/design change; for instance if the landmark for class T in Figure 2 is moved from library B2 to B1.

Landmarks are created using the mouse; thus landscape (and thereby software design) creation and (re)design is an interactive process.

Landmarks also mediate actions from the user to the associated components. The set of actions available on any given component depends on the aspect of the map it is located in; e.g. in a version control map, landmarks mediate actions like check-in and check-out to the underlying components, in a management aspect map the user can edit task lists, log spent staff hours, etc.

The direct interaction also extends to outlines: Moving/resizing an outline will change the displayed region in the corresponding detail map. This is an intuitive way of moving in 2D space compared to the often seen use of a vertical and horizontal scrollbar.

4.3 Shared landscape

The landscape with landmarks and decorations as well as the data of the underlying components are shared among developers in the software project. Therefore modifications made by one developer to a landmark and/or to data in the underlying component are immediately reflected in all running Ragnarok instances.

5. RAGNAROK PROTOTYPE

The Ragnarok visual model exists in a prototype implementation. Though the basic concepts of the model are implemented, the current prototype only provides a handful of different aspects. A subset of these aspects is described below.

A snapshot of the prototype, loaded with the Ragnarok project itself, is depicted in Figure 3. The Ragnarok window is divided into four parts: In the upper left corner is the *world map* (1) with outlines (9) of open detail maps. The world map has a fixed position in the Ragnarok window for easy reference and overview. The lower left part contains the *log window* (2) which is essentially a running log of important operations, here a version control check-out operation. The bottom right corner contains the *status bar* (3), which displays warnings and status information. On the right is a large area (4) in which may reside multiple *detail maps* (5–7). Each detail map has its own frame colour identical to the colour of the corresponding outline in the world map. Visual landmarks are generally displayed as simple, coloured, rectangles containing component name and possible additional information. Clicking any visual landmark brings up a context-sensitive menu which lists available actions on the underlying component (8).

Figure 3: Overview over the Ragnarok prototype window. The numbered parts are explained in the text.

5.1 Prototype usage

Initially the prototype only displays the world map. To create new detail maps, the user defines the region to view by drag-selecting a rectangle ('rubber-banding') on the world map using the mouse. This results in the creation of a new detail map and the corresponding outline in the world map. Outlines can be dragged around the world map using the small circle in the upper left corner; when released the associated detail map is updated to display the new region.

Detail maps can be moved and resized. Two types of resize are possible: Normal (making the map bigger reveals a larger portion of the underlying landscape) or rubber-sheet (making the map bigger displays the same region but zoomed in, like if the landmarks were on a rubber sheet). To zoom in on a certain region one simply defines the region by drag-selecting it, using the mouse, directly in the detail map; the map then zooms in on the region while possibly resizing itself to keep the aspect ratio of the selected region. Aspects can be chosen using the buttons in the top right part or using pop-up menus on the map frame.

Figure 4: A map showing the version control aspect.

5.2 Version control

Software components in Ragnarok are under tight version- and configuration control (Christensen, 1998a; Christensen, 1997). An important collaboration aspect in source code version- and configuration management is to enable the individual developer to overview how his/her private copy of source code relates to the overall project code. The typical question is: ‘Do I have the newest version of the project libraries?’

The version control aspect visualises this in a compact form. Colour coding of landmarks is used to show the state of the developers local copy. Referring to Figure 4 light red (medium gray on the figure) indicates components where newer versions exist. Gray (hatched) indicate that the local source code match the newest. The colours light green (dotted) and bright red (dark gray) are used to convey information about currently ongoing work: Light green indicates that the developer himself is currently editing source code in the component, bright red (warning) that some other, named, developer is working on it, and thus warns about potential conflicts if the developer decides to edit this component as well. Light yellow (light gray) indicate indirect changes because components depended upon have changed.

The context sensitive menu, half visible, allows version control commands, check-in and -out, display version graph, source code access, etc., to be issued to the individual components.

Progress is instantly reflected in all running Ragnarok instances and thus the evolution of the software system is visible on-line: For instance, if developer ‘johan’ checks in component ‘VersionedFile’ in Figure 4, the component will immediately turn light red in version control maps of all other developers, to indicate that a new version is available.

Figure 5: A map in visual scripting aspect, showing the result of a grep for the string 'GetLockOwner'.

5.3 Topography

This aspect minimises the amount of information presented to provide a compact overview, refer to the world map (1) in Figure 3. Decorations are not shown, and landmarks are grayed according to their nesting levels and without text. The context-sensitive menu lists (source) files in the component and choosing one loads the file into an editor.

This aspect is the standard one for the world map providing convenient overview of the project landscape and a neutral background for outlines. An unanticipated but strong feature of the topography aspect world map is that in essence it is a compact and fast file browser: Any of the 160 source files in the Ragnarok project can be loaded into our emacs editor using *one* mouse-click in the world map (clicking the landmark brings up a pop-up menu listing all files, releasing the mouse button over the wanted file tells our editor to load it).

5.4 Visual script

This aspect allows users to run scripts, written in the interpreted language TCL (Ousterhout, 1994) on (parts of) the project and interpret the result spatially and visually. User actions like mouse clicks on landmarks or positions also result in user defined TCL functions being called. As an illustration the prototype has a visual grep facility, please refer to Figure 5. Here the user has requested for a grep in the source files with target string 'GetLockOwner' to be run in a part of a project (same part as in Figure 4).

In this aspect the interior of landmarks is filled with black bars. Each bar represents a single file in the component, the bar height is a relative measure of the file size measured in lines. Each red line (white on the figure) shows that the search string

occurs in the file at this relative position. Clicking and holding down the left mouse button near a red line pops up a text viewer displaying 20 lines around the position where the search string occurs in the file—releasing the mouse button again makes the text viewer disappear again. This way one can quickly browse the occurrences and their immediate context without polluting the screen with numerous new windows. Double clicking a red line automatically loads the file into the editor centred on the matching line.

This visualisation of a recursive grep is compact and provides better overview than traditional textual recursive greps. Furthermore the clustering, density, and distribution of red lines in itself give important information. For instance, grepping for a function or class name may show misuses ('Now why is there a call in the GUI library?') or high coupling ('Hey, look, this class pops up in every component in the system!') that are easily missed in a 300 line textual output.

By basing this aspects on user written, interpreted, scripts, Ragnarok provides a degree of tailorability to the context of a given project: Developers can write scripts that provide custom visualisations. The TCL language has strong support for file handling and invoking external programs and it is therefore relatively simple to parse files (as done in the grep case above), or invoke profilers, run regression tests, extract relevant data from a project database, etc., and visualise the results of such external processing. The ability to associate user written TCL scripts to mouse clicks makes these custom visualisation direct manipulable: Clicking a landmark that highlights an unsuccessful regression test run can load the test into an editor; clicking a landmark with project data can instruct the database to load the proper table/view, etc.

5.5 Preliminary experiences

The Ragnarok prototype has been used in the continued development of Ragnarok itself, as well as on a minor part of the Mjølnir BETA System (Andersen et al., 93). It is currently being introduced in a medium-sized project (code size: 225KLOC), ConSys (ISA, 1996), where three developers are developing a control system for large scale equipment in experimental physics.

Preliminary results are encouraging. Some observations follow:

- Typically landmarks are created with sizes that allows about three to four levels of the design structure to be visible on a map.
- When designing a landscape, it is important to vary landmark sizes and arrange them in somewhat irregular patterns; otherwise the landscape becomes too uniform and identifying the right component based on its position and appearance becomes difficult.
- In software development there are often a number of core components that are constantly worked upon and a larger number of more stable components that are inspected less often. Not surprisingly developers can accurately locate the landmarks of core components without hesitation. Less often visited landmarks are located in an iterative manner, where a larger region around the expected location is examined and the sought landmark identified from its name.

6. DISCUSSION

Below we argue for the benefits of using a geographic space metaphor in a software development environment:

- *Overview and navigation:* In traditional systems the method for data and file access is name-based: You must remember the sequence of folder/directory names in order to find a file. The Ragnarok approach change the focus from name-based to location-based search: Developers know *where* the landmark containing the file is located. Because navigational and overview knowledge acquired performing one task can be carried on and remains valid performing the next, the design landscape (or the relevant part of it) quickly becomes well known. A company can define *positional semantics* by for instance require that GUI components are placed in the upper parts of the landscape, etc. Thereby the rough outlines of a project landscape will be known even for a new developer on a project.
- *Mediating, up-to-date, software design:* The software landscape becomes the focal point of daily activities: Many daily development activities are performed through interaction with the landscape. We believe this is a direct and intuitive way and that the pressure to ensure a correct, up-to-date, design landscape is strengthened. Newcomers on a project can be introduced to the software design directly in the daily software development environment.
- *Collaborative issues:* Members of the development team get well acquainted with the software landscape creating a common understanding and allows everyday language, like ‘to the left’, ‘in the northern part’, to be used. The same, spatially stable, layout can be used to visualise different aspects meaning that persons with different responsibilities, like managers, designers, programmers, testers, maintainers, etc., refer to the same layout. Secondly the landscape always provides a snapshot of the state of the project: One example was given in section 5.2, another is that when any developer log spent staff hours on a component this is immediately reflected visually on a management aspect map showing estimated-time-to-complete and/or budgeted versus actual spent time.
- *Visualisation framework:* Different aspects/data of the same component are visualised in the same geographical region of the landscape. The primary effect is stability of the visual appearance which is important for navigation. A second effect is that different aspects gets geographically correlated, which is a natural and strong way of comparing data: If a bug-report aspect map (displaying number of detected bugs) highlights certain landmarks, and the same landmarks are highlighted in a staffing aspect (displaying how components are staffed), one is instantly warned.

The aspects described in section 5 of course only serve as a sample of uses. Many other aspects can be envisioned: Defect reporting, possibly determined by automatically running regression test suits on modified components; profiling information to identify bottlenecks in a system; release control, based on passed tests on checklists for each component; different management aspects focusing on estimated-time-to-complete, actual- versus budget-cost estimates, etc.

The map aspect property is important because of the inherently multi-dimensional nature of software. Each dimension can be superimposed on the same landscape—just as different types of ordinary maps visualise different properties of the same region of a country: Vegetation, town names, elevation, or population density, etc.

Note that Ragnarok is not purely location-based: The files defining the substance of a component/landmark are not assigned a position (however, this can be simulated by having one component per file), therefore files are accessed from a traditional textual list of filenames in a context-sensitive menu. This design is deliberate because the focus is on design abstractions like classes, libraries, etc., and not on files; and usually such abstractions require several files (for example a .h and a .cpp file in C++ to implement a class).

Navigation using the presented visual model relies on a stable landscape that is well known to the developers: If the landscape is constantly modified, spatial knowledge is virtually non existing—compare the effort of driving home from the office every day with the effort of driving in an unknown city. Consequently, navigation will become easier as the design stabilises. During the initial analysis and design phases, ideas and abstractions are fostered and discarded rapidly meaning many changes in the landscape; in these phases, a more traditional, search-based, navigation mechanism is a beneficial supplement.

In ‘different-place’ software development teams, the design landscape can more actively mediate collaborative awareness: When a developer is working on a certain component, she is metaphorically speaking working in a certain place in the landscape. This can be shown in all running Ragnarok instance by for instance a special image/cursor identifying the developer. This indicator could be a link to the person for instance by opening a audio- or video link when clicked.

An important issue is scalability. As the Ragnarok software architecture- and visual models are inherently hierarchical Ragnarok should scale without problems. In the discussion so far the world map has been described as displaying the overall project but in a large project the typical situation would be that it displays only the part relevant for the team or individual—in essence defining their ‘world’.

A final note is on software abstractions that are reused in different contexts. Consider a container class hierarchy defined in a library component that we want to document is used in, say, both a business- and graphics module. The Ragnarok visual model demands that a component is assigned a unique position and thus the same container classes cannot be displayed both in the library, business- and graphics regions of the design landscape. This problem has no easy solution; our current suggestion is to provide ‘shadow’ landmarks which through their appearance (grayed for instance) acts as hyperlinks to the real landmarks—actions are forwarded to the real landmark and a special action should be to initiate a zoom and pan travel to the real landmark.

7. RELATED WORK

CASE tools: Looking at e.g. map (5) in Figure 3 Ragnarok may resemble a traditional CASE tool or UML diagram editor. However, the focus of CASE tools and Ragnarok is different. CASE tools are generally analysis and design tools with code generation features i.e. with strong support and focus on the early phases of a project and the programming task. The emphasis in Ragnarok is foremost on making design abstractions manifest and manipulable by assigning a geographic location and secondly on documenting the design using some notation like UML; once this is done it is used as a visualisation framework for project data which CASE-tools traditionally

do not address.

Pad++: Another interesting comparison is to Pad++ (Bederson et al., 1996; Bederson and Hollan, 1994; Perlin and Fox, 1993). Pad++ is an innovative and powerful 2D visualisation system. In Pad++ the user manipulate objects on an infinitely zoomable 2D surface. The Pad++ system incorporates a very effective engine for panning and zooming. The objects on the surface are text, simple graphics, and images. The underlying visual model employed in Ragnarok is similar to Pad++. The difference is the objects handled. In Pad++ objects reside directly on the Pad surface; in contrast landmarks serve as visual *representations* of the complex, multi-dimensional, data of the underlying component; they *are* not the actual data. Therefore Ragnarok uses the *same* region to visualise *different* data in different aspect maps (e.g. grep matches or version information). Pad++ provides portals and lenses but these provide different visual representations of the *same* data, say a slider or textual representation of an integer object. The aspect property of Ragnarok is essential because of the multi-dimensional nature of software.

SeeSoft: The visual scripting facilities are inspired by the interesting work in the Bell Labs ‘SeeSoft’ systems (Ball and Eick, 1996; Baker and Eick, 1995). SeeSoft is a powerful tool for visualising properties of text files (source code) in a highly compact form. Individual lines in the source code are represented by colour coded text lines, pixel lines or even individual pixels. Compared to SeeSoft the Ragnarok visual layout is less compact due to the ‘unused’ space between the landmarks but on the other hand it carries valuable information in itself compared to SeeSoft which sorts files alphabetically. The stable layout is important in many contexts because it eases comparisons (like profiling information before and after an optimisation phase) and the distribution and density of ‘hot spots’ in itself provides valuable clues to system properties as mentioned in section 5.4.

MacFinder: No comparison is complete without contrasting the Ragnarok visual model to well-known desktop space metaphors like the Macintosh Finder. In Finder objects (files/directories) are represented by icons that when clicked expands into a window showing its contents (part objects). These only shows one level of the part/whole hierarchy and therefore navigation typically spawns many new windows. In contrast most Ragnarok aspects show 3-5 levels of the hierarchy thus showing context and avoiding intermediate maps during navigation. More important, however, is the underlying spatial model: In Ragnarok all landmarks have a specific relative position to all other landmarks—you can always answer the question: ‘Does landmark A lie left of landmark B?’ In contrast MacFinder objects only have a position relative to the window it is part of but not to objects in other windows: You cannot tell the position of MyMac:FolderA:Document1 relative to MyMac:FolderB:Document2 in general, only where they are relative to each other in your current, transient, layout of your desktop. Therefore the Finder spatial reference frame is weaker and more difficult to remember, share, and communicate in a team.

8. FUTURE WORK

Currently there are a number of lines along which Ragnarok evolves. A high priority is to get more experience with actual every-day use of the prototype. Ragnarok is build

upon a software configuration management (SCM) subsystem that is currently used in two, real, medium-sized development projects (Christensen, 1998a). They use the SCM subsystem through a character- and command based interface and experiences from these projects show a substantial navigational overhead. The Ragnarok prototype is currently being introduced in one of these projects and the other will follow soon. We expect this will prove a significant decrease in navigational effort. Therefore a current effort is instrumentation of the prototype which allows the actual usage pattern to be logged for later analysis.

Tailorability of aspects and visualisations is another key issue. No two development projects are alike and visualisation needs will vary. Therefore it should be possible to create run-time specifications of new aspects and visualisations including: A) Define structure of an annotation B) Processing scheme defining the appearance of landmarks based on annotation data processing, and C) A set of context sensitive actions (tailorable pop-up menus and associated actions). The current visual scripting facility currently provides a limited version of B) only.

Pad++ introduced the concept of semantic zooming i.e. the appearance of an object varies according to the zoom level it is seen in. Ragnarok presently have limited support for this capability, but it is an beneficial extension: When landmarks appears physically small on the display they should only convey summary information as e.g. a colour coding; when zoomed in the landmark itself could begin to display more detailed information in its interior.

9. CONCLUSION

The main contribution of the present work is the proposal to use a geographic space metaphor to organise the multi-dimensional data produced in the software development process, centred on the logical software design; that the design landscape should mediate daily development activities; and the use of aspect maps to visualise the individual dimensions of the data, possibly in a processed way.

We have argued that this proposal provides a development team with a common reference frame for overiewing, navigating within, and documenting a large software structure by utilising humans fine spatial and visual perception. Furthermore it has the ability to visualise many, although not all, interesting aspects of the data associated with software and software development in a way that makes comparisons of different aspects easy.

The preliminary experiences support the strength of position-based navigation—once the landscape has become well-known the world map is a compact and fast browser of source files in a project: Faster and less tedious than traditional approaches like shell name-completion or other graphical hierarchical tools like MacFinder and Windows95 Explorer. The visual grep in the scripting aspect provides better overview and less confusion than traditional textual recursive greps. The ability to explain the software design to newcomers on a project directly in the daily development environment is valuable.

The Ragnarok prototype is freely available from the author. A overview and reference guide describing the prototype in detail is provided on the World-Wide-Web (Christensen, 1998b).

REFERENCES

- Andersen, P., Bak, L., Brandt, S., Knudsen, J. L., Madsen, O. L., Møller, K. J., Nørgaard, C., and Sandvad, E. (93). The Mjølner BETA System. In *Object-Oriented Environments - The Mjølner Approach*, pages 24–35. Prentice-Hall.
- Baker, M. J. and Eick, S. G. (1995). Space-filling Software Visualisation. *Journal of Visual Languages and Computing*, 6:119–133.
- Ball, T. and Eick, S. G. (1996). Software Visualization in the Large. *IEEE Computer*, 29(4):33–43.
- Bederson, B. B. and Hollan, J. D. (1994). Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics. In *Proceedings of ACM UIST '94*. ACM Press.
- Bederson, B. B., Hollan, J. D., Perlin, K., Meyer, J., Bacon, D., and Furnas, G. (1996). Pad++: A Zoomable Graphical Sketchpad for Exploring Alternate Interface Physics. *Journal of Visual Languages and Computing*, 7:3–31.
- Bürkle, U., Gryczan, G., and Züllighoven, H. (1995). Object-Oriented System Development in a Banking Project: Methodology, Experience, and Conclusions. *Human-Computer Interaction*, 10:293–336.
- Christensen, H. B. (1997). Context-Preserving Software Configuration Management. In Conradi, R., editor, *Supplementary Proceedings: 7th International Workshop, SCM7*, pages 14–24, Boston, USA.
- Christensen, H. B. (1998a). Experiences with Architectural Software Configuration Management in Ragnarok. In Magnusson, B., editor, *System Configuration Management*, Lecture Notes in Computer Science 1439. ECOOP'98 SCM-8 Symposium, Springer Verlag.
- Christensen, H. B. (1998b). *Ragnarok: Overview and Reference Guide*. Department of Computer Science, University of Aarhus. <http://www.daimi.au.dk/~hbc/Ragnarok/ragn-doc.html>.
- Egenhofer, M. J. and Mark, D. M. (1995). Naive Geography. In Frank, A. U. and Kuhn, W., editors, *Spatial Information Theory / A Theoretical Basis for GIS*, pages 1–15. COSIT '95, Lecture Notes in Computer Science 988, Springer-Verlag.
- Frederick P. Brooks, Jr. (1987). No Silver Bullet—Essence and Accidents of Software Engineering. *IEEE Computer*, 20:10–19.
- Hutchins, E. L., Hollan, J. D., and Norman, D. A. (1986). Direct manipulation interfaces. In Norman, D. A. and Draper, S. W., editors, *User Centered System Design*, chapter 5. Lawrence Erlbaum.
- ISA (1996). Consys. <http://isals.dfi.aau.dk>. ISA: Institute for Storage Ring Facilities, University of Aarhus.
- Kuhn, W. and Blumenthal, B. (1996). *Spatialization: Spatial Metaphors for User Interfaces*. Geoinfo-Series, Department of Geoinformation, Technical University, Vienna. Reprinted tutorial notes from CHI'96.
- Lamb, D. A. (1996). Introduction: Studies of Software Design. In Lamb, D. A., editor, *Studies of Software Design*, Lecture Notes in Computer Science 1078. ICSE'93 Workshop, Springer Verlag.
- Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit*. Addison-Wesley Professional Computing Series.

- Perlin, K. and Fox, D. (1993). Pad - An Alternative Approach to the Computer Interface. In *Proceedings of ACM SIGGRAPH '93*. ACM Press.
- Rational Rose (1997). Unified Modeling Language, version 1.0. Rational Software Corporation, Santa Clara/CA. URL:<http://www.rational.com>.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice-Hall International Editions.
- Shneiderman, B. (1983). Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer*, 16(8):56–69.
- Sommerville, I. (1992). *Software Engineering*. Addison-Wesley Publishers Ltd., 4 edition.

Discussion

Ken Fishkin: Layout is a severely underconstrained problem, and the algorithm you choose can drastically affect user experience. How do you deal with this?

Henrik Bærbak Christensen: I agree. For that reason, we use no layout algorithm: layout is specified interactively by the user.

Helmut Steigler: How open or closed is your system e.g. with respect to include existing software or products?

Henrik Bærbak Christensen: The system is on top of and independent of the underlying development environment. In our case the interfacing to Visual C++ raised some difficult questions.

Stephane Chatty: You presented graphs more than landscapes actually, and I think that trigger different ways of thinking. So do you think that what is really important is properties like salient positions, or that a real landscape would bring even more?

Henrik Bærbak Christensen: I decided to use maps instead of a more true “real world” metaphor as they are easier to navigate in, and generally easier to use. For instance, travel in real space is more time-consuming than just clicking at some point on a map. Gilbert Cockton added to the answer: “There is no big difference between landscape and maps.”

Nick Graham: Have you considered the social aspects of introducing this kind of tool where you may run into difficulties with stakeholders who are not enthusiastic about giving up their “power” of being the only one to understand the software architecture?

Henrik Bærbak Christensen: No, an interesting issue that I have not thought about.

Morten Harning: In what situations would you suggest the use of a geographic metaphor?

Henrik Bærbak Christensen: If you have a system with objects in a hierarchical structure, that is relatively stable over time, this metaphor has much to offer.

Shijian Lu: One of the important aspects of geographic space is that all locations are predefined therefore the layout of a map is dictated by location while in a software architecture diagram the layout is very much dictated by readability. How do you map a software architecture into a unique layout?

Henrik Bærbak Christensen: Yes and No; Yes, a complete rearranged landscape would make us loose sense of direction; but most rearrangements takes place at the detail level (adding few new classes, removing some), and not in the overall picture. Large architectures are prohibitively costly to do large restructurings on.

Shijian Lu: If there is no unique map from a software architecture into a fixed layout, in the fail of modification, the components’ location will be changed, then the notion of fixed location of the software landscape will be lost. How can that improve easy navigation?

Henrik Bærbak Christensen: Yes and no. Yes, because we need stable structure for the layout, but No, as large architectures are costly to build and therefore in real situations it is unlikely that the whole architecture is rebuild.

Frank Tarpin-Bernard: Even if designers use creative processes, I am not sure that

they design from scratch. Don't you think that reusable design patterns are a good way to improve design?

Henrik Bærbak Christensen: True, patterns and preexisting subarchitectures are very important. I think one should therefore import their landscape just as much as import the underlying architecture.

Prasun Dewan: Many programming environments typically impose a lot of overhead on you. To build a "hello world" Java program I have to create a class, a file, a project and a workspace. Now you want me to also create a landscape. Is this a reasonable overhead?

Henrik Bærbak Christensen: Good comment: A landscape may seem like an overhead but in Ragnarok drawing a landmark potentially constructs much of the framework you speak of: Source code framework, project management framework, etc. The vision is of course that the benefits should outweigh the cost.

Christian Gram: Why are landmarks dull similar rectangles? Why not use triangles, circles, ...

Henrik Bærbak Christensen: We did that because it was easier.

Christian Gram: Why is the system called "Ragnarok" and not Paradise?

Henrik Bærbak Christensen: Ragnarok is the cataclysmic struggle between gods and giants where the old world is destroyed, according to nordic mythology. The Ragnarok name is kind of black humour—we hope to do better, and surely we have not promised too much :)