Digitally Annexing Desk Space for Software Development  
(NIER Track)

John Hardy, Christopher Bull, Gerald Kotonya and Jon Whittle  
School of Computing and Communications  
Lancaster University  
Lancaster, UK  
{ john, chris } @highwire-dtc.com and { gerald, whittle } @comp.lancs.ac.uk

ABSTRACT
Software engineering is a team activity yet the programmer's key tool, the IDE, is still largely that of a soloist. This paper describes the vision, implementation and initial evaluation of CoffeeTable – a fully featured research prototype resulting from our reflections on the software design process. CoffeeTable exchanges the traditional IDE for one built around a shared interactive desk. The proposed solution encourages smooth transitions between agile and traditional modes of working whilst helping to create a shared vision and common reference frame – key to sustaining a good design. This paper also presents early results from the evaluation of CoffeeTable and offers some insights from the lessons learned. In particular, it highlights the role of developer tools and the software constructions that are shaped by them.

Categories and Subject Descriptors  
D.2.6 [Software Engineering]: Programming Environments

General Terms  
Design, Human Factors

Keywords  
IDE, CSCW, software visualisation, tabletop user interfaces, collocation, collaboration and collaborative construction.

1. INTRODUCTION
Modern integrated development environments embody many accidental complexities that have arisen from years of evolution in hardware, software and development practice. Such complexities pose problems and bottlenecks such as a dependence on file based editing [1], large code and resource navigation times [2] and restrictive ‘letterbox’ views into complex systems, which are far from advantageous to comprehension [3].

Paraphrasing words from a recent paper [4] succinctly identifies a dilemma in IDE design: Software engineering is a team activity yet the programmer’s key tool, the IDE, is that of a soloist. To address this, we developed CoffeeTable (Figure 1) as a way of exploring a new collaborative approach. CoffeeTable is distinguished by a large interactive desk that allows small groups of developers to synchronously work on a single, live revision of a software project. The desk itself functions as a place to put laptops, but also displays an interactive visual representation of the software architecture and workflow. Elements of the visualisation can be virtually dragged onto laptops for developers to work in a personal space.

Collaboration requires coordination, thus CoffeeTable inevitably increases the amount of information exchanged during development. However, rather than delaying decisions or relying on the assumptions of individuals, CoffeeTable's design encourages developers to informally negotiate decisions which would previously be handled offline due to lack of collocation. In addition, the shared physical space also allows people to see and talk to each other whilst referencing a common frame of understanding [5] [6]. Disharmony between developers can lead to redundant code, misplaced functionality and greater potential for defects, production bottlenecks and frustration.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE’11, May 21-28, 2011, Waikiki, Honolulu, HI, USA.  
Copyright 2011 ACM 978-1-4503-0445-0/11/05... $10.00.

Figure 1. Showing two developers at the CoffeeTable using laptops and interacting with visuals via an infrared stylus.

CoffeeTable aims to:

Minimise production bottlenecks through features that encourage the integration of agile and traditional practice. Agile methods [7] recognise the inevitability of change, emphasise active stakeholder involvement and use short iterations as a basis for rapid system delivery. Traditional models emphasise predictability, accountability and control. They rely on detailed upfront planning to minimise unexpected changes and use systematic processes to drive the development. However, the success of a software project is influenced by many factors including: the size and complexity of the systems being built, the consequences of failure, the volatility of the application domain, the competence of development teams and compatibility with the organisation culture. These factors mean that no single development model can be effective in all situations. A better solution lies in a working environment that allows developers to migrate between the two models based on the nature of the task and the risks posed by factors that influence the success of the project. Rather than thinking of these as separate or competing practices, we hypothesise that when using CoffeeTable developers are encouraged to naturally transition between these...
methodologies – choosing the best mode of working to suit the task at hand.

Reduce the cognitive overhead introduced by tools through visual and semantic organisation mechanisms. Several studies have shown that programmers spend a significant amount of their time navigating code and other development resources [2][8][9]. The use of files within the OOP paradigm is a good example of an accidental byproduct of our technological morphology becoming an unnecessary cognitive overhead. These and other types of cognitive overhead can be replaced with interesting forms of visual semantic organisations that capture different aspects of development, including: development activities, system aspects, collaborations and responsibilities. For instance, position, size and orientation can imply territories and thus responsibility [10] and clusters or arrangements can represent logical relationships. Such organisation is already emerging in IDEs and even languages – e.g. BREGION tags, partial classes and mixins.

Support awareness and assist navigation by capturing and sharing work as it happens. It is well known that programmers do not add code to a project in a linear manner; rather they add pieces to the system by realising an internalised strategy which satisfies particular sub-goals [11]. Therefore, each development task forms a trail that cuts across the information space of the software system by creating links between relevant code and other development resources [12]. CoffeeTable captures these trails as they are created, but unlike works where similar ‘working sets’ are central to the interface structure [8], they are included as a secondary navigation structure embedded within a larger visualisation. This allows developers to move back and forth along the trails they and others have made. While not a core focus of our work, navigating the development in this way may be of interest to those investigating the morphology of a development. This simple but powerful feature is rarely present in modern IDEs.

Support rich content documentation. In-source documentation is often about expressing complex ideas and linking developers to important or clarifying resources. In most IDEs this feature is limited to textual references. CoffeeTable lets developers attach drawings and web-media to elements of the software architecture that can be viewed inline. These help to express complex ideas and relationships that are either inherently abstract, philosophical or too complex to be clearly explained with a UML diagram.

2. DESIGN AND IMPLEMENTATION

2.1 Overview

The CoffeeTable environment (Figure 2) is composed of 3 components: Firstly, the physical hardware including a 1.8x1.4m table, a short throw projector, 2 Nintendo Wii Remotes and 2 infrared pens. Secondly, client software that runs on laptops that developers use to write code. Lastly, server software that manages the code being developed whilst also rendering the interactive visualisation (Figure 3).

The workspace itself is constructed through the synthesis of the three main types of information exchanged:

Significant Actions – These are actions that should be observed by all the team members operating in the software space. Examples include; creating or deleting a class or moving functionality.

Architecture Summary – These are spatial representations of the software elements comprising the architecture (Section 2.2). Comprehending code is a key part of the software development process [12][13]. Combining recent attempts at maximising spatial memory [2][3] with source code summaries [14] and simple visualisation creates an environment where developers grow the workspace into an appropriate reflection of what they are working on. An added advantage is that different system stakeholders can use the simplified representations as a focus for discussion.

Workflow Indicators – These are visual indicators that help others in the group to understand, at a glance, the different states, foci and responsibilities within a workflow. For instance, focus can be conveyed through size, position and orientation relative to the people around the table. States of development can be conveyed through visual cues that indicate whether methods are stubs or have documentation. Responsibility or transient forms of ownership are conveyed with physical actions supported with further visual cues. For instance, if a method is physically dragged onto a laptop, this triggers a pulsing colour within its visual and the drawing of a line that represents the information being exchanged between the visual on the desk and developer.

2.2 Architecture and Workflow Visualisation

Interface Artefacts

(A) Window – Windows are divided into two sections: a content panel and a semantic link panel. The content panel can display various media such as a web browser, drawing canvas or code editor. The adjacent semantic link panel lists links to more content that is of relevance to what is displayed in the content panel. Essentially, an interface to a bi-directional weakly connected graph that is transparently generated as the programmer works.

(B) User Areas – User areas are ‘drop zones’ that serve as a mechanism for exchanging information between the visualisation and specific clients. A developer uses their stylus to draw a shape that they can then ‘acquire’ using their client software.

(C) Linker Band – These are important interface elements that visualise the process of information exchange within the system as a direct result of stylus interaction. Uses include linking together items such as methods, objects, websites, code, documentation and sketches. Linker band enabled items will, when pressed, draw a line from the pressed control to the user’s pen. When released, the linker band will generate an event which invokes subscribing functionality.

(D) Highlight Lines – A simple transient glowing line used to communicate the state and location of information. Red lines show the location of ‘locked’ items. Green lines show the movement of an item to and from a user area. Yellow lines highlight requests for input and blue lines indicate compile requests.

(E) Widgets – These are analogous to icons that use linker bands to invoke significant actions within the visualisation. For example one may have a ‘create object’ widget which developers can use
to create new objects. Notable widgets are: drawing canvas, web browser, create, close, delete and compile.

**Architecture Representation**

**(F) Object** – This visual represents a software object within a project. Protruding arms indicate an internal member such as field, method or whitespace. Objects can be moved by dragging the centre and resized/oriented by dragging the border. Centrally located accelerators are used to filter members, create accessor methods and perform other common functions.

**(G) Internal Members** – Internal members protrude as arms of objects; each represents a child member of the parent object. A member can be a field, a method or even an inner class. Members can be edited (locked) in a private space by being dragged into a user area. Similarly they can be viewed in public space within a window by being dragged onto empty desk space.

![Figure 3. A selection of visuals relative to those in Section 2.2.](image)

### 3. INITIAL EVALUATION

#### 3.1 Experimental Design

Our evaluation used a mixture of quantitative observations and qualitative feedback from interviews. To provide an early indication of the efficacy of **CoffeeTable** with respect to other modes of working we devised an experiment that compared 3 collaborative coding environments, namely: pair-programming (Eclipse), traditional (Eclipse + SVN) and **CoffeeTable**.

A total of 6 professional developers, working in pairs, undertook 3 programming tasks. Each task lasted 45 minutes and was presented as a mock requirements specification that described a range of complex and simple features with the intention of emulating a real world scenario. The tasks were tailored to be relevant during pair programming but less frequent and more irrelevant when programming separately. When using **CoffeeTable**, participants tended to communicate in short clarifying bursts with the visualisation acting as a common reference point. To assist these conversations, it became apparent that we needed to support concurrent editing of the same code. This would enable developers to demonstrate ideas and capitalise opportunities to be more productive as a team.

Overheads in tools affect the way source is constructed and modified. Developers appeared to structure contributions based on the overheads involved in navigating source within the tools: “making a new method is a real effort so I’m just going to bung it in here.” The spatial navigation on the table is very different to that in the traditional methods, we note how the relative lack of shared awareness made it easy for one to unintentionally corrupt what the other was working on – directly stalling progress. This was not the case with **CoffeeTable**.

**Does the soloist culture preclude the possibility of constructs that aid collaborative development?**

One of our most interesting observations was that participants frequently overlooked opportunities to be more productive as a team. For instance, they opted not to create separate functions to handle individual features, instead, reverting to working individually within a large monolithic control structure (a main menu). This created a bottleneck that prevented the other

produced fewer features at a higher level of solution quality. **CoffeeTable** provided more fully completed complex features than pair programming at the cost of overall solution quality. The **CoffeeTable** solutions were also generally of a higher quality than those produced using traditional methods. **CoffeeTable** allowed developers to work independently on well-understood tasks and participants were observed switching to a collective effort when the need arose [15].

**Methodological Lessons**

Reflecting on the study, we note that debugging and auto-completion are important features to either remove or provide across all treatments as they are heavily relied upon by developers. In one instance a participant using **CoffeeTable** opened Eclipse for API exploration as it was “faster than accessing the internet.” The lack of auto-completion support for external APIs within the **CoffeeTable** client may also have contributed to the large number of minor corrections required in comparison to the other methods. The most common errors made with **CoffeeTable** were: (1) uninstantiated variables, (2) lack of required user feedback and (3) out of order arguments.

To properly understand coding practices it is important to observe developers for long periods of time. Our experience suggests that the 45 minute sessions were not long enough. In addition, we observed that after 2 or 3 sessions the developers became tired. This alludes to a more sustainable evaluation methodology being found in automated statistics gathering, interviews and ethnographic study within a working environment.
developer from simultaneously contributing their work. Despite this technically being addressable through the addition of concurrent method editing at the user interface level, it highlights that developers are not accustomed to structuring code for concurrent collaboration.

To paraphrase: developers are not taught to think about programming as a group exercise beyond the division of labour. The fine-grain collaboration CoffeeTable strives for plays a key role in art forms such as theatre, music and indeed engineering. In pursuit of effective collaborative techniques, this provides evidence that micro-patterns and structures can be classified - and thus managed - by their ability to boost productivity in collaborative environments.

4. CONCLUSIONS

We recognise that programming is an activity rich in culture and niche and thus to fully evaluate the potential of CoffeeTable we should seek to deploy it long term in different organisations. The early results presented here offer interesting insights about the benefits and limitations of using such a tool to support software development, but are by no means definitive [6].

The main conclusion that we can draw from these early results is that there are constructions within both program code and team dynamics that are shaped by the tools used by a team. Thus, exploring diverse tools can encourage the emergence of more appropriate constructions. Our findings recognise at the micro-level what others like Conway [16] and Parnas [10] have observed at the macro-level. In 2005, de Souza, Froehlich and Dourish used a visualisation tool to reveal the intertwined nature of software artefacts and development processes [17]. Despite taking different paths, we conclude on a similar point: Software artefacts reflect the tools, processes and social structures that were applied to regulate and create them. By exaggerating certain overheads we have demonstrated how this can be both beneficial and also detrimental to the software development process.

During the interviews it was suggested that CoffeeTable be repurposed as a “creative prototyping tool.” We believe this would be a good reaplication of the technology and one worth exploring further. A key feature shown to be missing was the ability for multiple people to simultaneously view and edit the same section of code. Another interesting suggestion was for a concurrent method editing at the user interface level, it highlights that developers are not accustomed to structuring code for concurrent collaboration.

To paraphrase: developers are not taught to think about programming as a group exercise beyond the division of labour. The fine-grain collaboration CoffeeTable strives for plays a key role in art forms such as theatre, music and indeed engineering. In pursuit of effective collaborative techniques, this provides evidence that micro-patterns and structures can be classified - and thus managed - by their ability to boost productivity in collaborative environments.

5. ACKNOWLEDGMENTS

This work was undertaken as part of the post-disciplinary HighWire programme at Lancaster University. HighWire is funded by the Digital Economy Programme: a Research Councils UK cross council initiative led by EPSRC and contributed to by AHRC, ESRC and MRC.

6. REFERENCES


