Rapid Visual Design with Semantics Encoding through 3d CRC Cards

Anthony Savidis\textsuperscript{1,2}, Panayiotis Papadakos\textsuperscript{2}, George Zargianakis\textsuperscript{2}

\textsuperscript{1}Institute of Computer Science, FORTH, Greece
\textsuperscript{2}Department of Computer Science, University of Crete, Greece
\{as, papadako zargian\}@ics.forth.gr

Abstract
Physical CRC cards (Classes – Responsibilities - Collaborators) is a well-known method for rapid software-design prototyping. It is commonly applied with numeric heuristics to encode design semantics and denote architectural relationships through card coloring, size variations and spatial grouping. Existing CRC design tools are weak in terms of interactivity and visualization, focusing mainly on form-based specification and documentation. We present Flying Circus, a tool for rapid and exploratory software-design prototyping via CRC cards, offering facilities such as: interactive 3d visualizations supporting encoding of semantic aspects through custom visual patterns and metaphors. Currently, there are numerous tools capable to automatically visualize design-related aspects from the source code structure, or from information ranging from modeling / design data (e.g. UML) or other forms of program meta-information. For instance, 3d Java code visualization in \[Fronk and Bruckhoff 2006\] reflects the hierarchical implementation structure, while interactive 3d views in \[Marcus et al. 2003\] allow visually query quantitative aspects of the source code. Additionally, semantically-related groups of components may be identified (a short of visual query) from program meta-information as highlighted areas of interest \[Byelas and Telea 2006\]. Sometimes visualizations are targeted in displaying aspects enabling programmers detect ‘code bad smells’ as in \[Parnin and Goerg 2006\]. Apart from static properties, behavior visualizations enable review dynamic characteristics, as in \[Greevy et al. 2006\] where traces of component instantiations and method invocations (messages) are rendered.

Motivation CRC cards [Beck and Cunningham 1989] have been extensively deployed as a software-design prototyping instrument apart of teaching and process description. CRC cards emphasize the exploratory and visual nature of software-design prototyping, allowing heuristic visual encodings and symbols, like color or size variations (for classes and links) and post-it annotations carrying meta-information (e.g. brief documentation, implementation notes, etc.). Such heuristics, though not part of the original method as such, are crucial as they allow embody important semantic information not otherwise expressible. While visualization tools exist to enable developers better analyze the structure and behavior of existing systems, little is done in supporting computer-assisted design as a visualization-centric activity supporting design exploration and semantics encoding.

Contribution We present a tool for exploratory software-design prototyping, to be actually utilized in together with general-purpose more comprehensive design methods like UML and other sorts of program visualizers, primarily supporting: (a) rapid visual software-design prototyping, with emphasis on effective interactive supervision and inspection; and (b) a corpus of tested visual encoding policies for software-design semantics to be applied during interactive visual design.

1. Introduction
In software development, visualizations support various related activities, like supervising design structures, extrapolating implementation details, or observing system’s runtime behavior. Our work mainly concerns computer-assisted software-design with highly interactive 3d visualizations supporting encoding of semantic aspects through custom visual patterns and metaphors. Currently, there are numerous tools capable to automatically visualize design-related aspects from the source code structure, or from information ranging from modeling / design data (e.g. UML) or other forms of program meta-information. For instance, 3d Java code visualization in \[Fronk and Bruckhoff 2006\] reflects the hierarchical implementation structure, while interactive 3d views in \[Marcus et al. 2003\] allow visually query quantitative aspects of the source code. Additionally, semantically-related groups of components may be identified (a short of visual query) from program meta-information as highlighted areas of interest \[Byelas and Telea 2006\]. Sometimes visualizations are targeted in displaying aspects enabling programmers detect ‘code bad smells’ as in \[Parnin and Goerg 2006\]. Apart from static properties, behavior visualizations enable review dynamic characteristics, as in \[Greevy et al. 2006\] where traces of component instantiations and method invocations (messages) are rendered.

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.

2. Related Work
Quick CRC [Quic CRC 2001] is a window-based 2d tool resembling in style and process the construction of interactive UML class diagrams. It emphasizes textual specification and documentation, rather than rapid conduct, visual design and exploratory process. EasyCRC \[Raman and Tyszberiwicz 2007\] is a 2d tool offering very limited interactive facilities, with an extension regarding CRC cards to model scenarios using UML sequence diagrams. CRC Design Assistant \[Roach and Vasquez 2004\] is yet another graphical 2d tool, intended to support students in designing real-life applications. It allows editing class name, description, super-class, subclass, and responsibilities information. All such CRC-card tools provide functionality mimicking the practicing of physical CRC cards in a 2d space, however, with no extra interactive flexibility. Practically, they are simpler forms of general-purpose more comprehensive design methods that one may adopt primarily for small-scale projects, if for some reasons UML is not adopted.

3. Visual Design Encoding and Exploration
In our tool, visual card attributes like rotation angle, dimensions, color and positioning, are to be exploited by designers as heuristic symbolic vocabularies for encoding software design semantics. The basic facilities for 3d manipulation of cards are shown in Figure 1. Rotation (with six degrees of freedom), repositioning, symbolic vocabularies for encoding software design semantics. The basic facilities for 3d manipulation of cards are shown in Figure 1. Rotation (with six degrees of freedom), repositioning, and resizing of cards are possible through effective 3d manipulators (see Figure 1), displaying also card axis (dashed lines) for alignment purposes. The manipulators appear or disappear automatically when the respective in-card controls are activated (click using the pointing device) by the user.
Continuing, we elaborate on the specific design encodings strategies that have been deployed and tested in the course of real practice. Once such techniques are instantiated in a 3d visualization context, they enable designers quickly recognize and recall design information within large software design structures.

### 3.1 Encoding through Varying Dimensions

With variations of dimension we may encode quantitative design attributes, such as source size and project size (in terms of files), or properties implying key programming benefits, like reusability and genericity. An important remark is that a design structure may be also annotated with information that is known or presumed either prior to implementation, or is consolidated after the implementation phase is initiated or entirely completed. Proposed encoding policies we have also adopted are the following:

- **Illustrate implementation size** (proportional to width).
- **Indicate implementation complexity** (by height).
- **Denote reusability potential** (by thickness) - polymorphic algorithms, templates, generic classes.
- **Signify comparative importance** (larger size) – critical components other may have larger dimensions.
- **Emphasize common dependencies** (larger size) - when many components depend on the same single component, although evident from incoming links, we may also draw the target component with increased size.

An example is provided Figure 2 for the design of a server, showing the protocol parser (left part: height denoting implementation complexity), distinct requested services (middle part: width denoting implementation size) and the service dispatcher (right part: thickness underlining potential reusability).

### 3.2 Encoding through Colors

Color encoding is amongst the most widely deployed methods to imply semantic information, capable of denoting grouping and classification. Coloring was also proposed as an extension to the visual vocabulary of UML. Suggested color encodings are:

- **Denote architectural grouping** for classes.
- **Indicate mission criticality** with high-intensity colors.
- **Signify common categories**, like storage, UI, etc., with distinct reserved colors.

Practically, color encoding alone is insufficient to support effective recognition of distinct artifacts or group relationships in severely crowded software-design spaces, unless appropriately backed-up with high-quality effective design inspection facilities.

#### 3.3 Encoding through Rotations

Spatial rotation of objects (six degrees of freedom) may imply emphasis, distinction, and semantic separation, and can be used to directly attract visual attention (we tested that planar rotations over 30° are well identified). Strong variations on angles or identical / similar spatial rotation on a group of items are easily and quickly recognized by human vision. Some of the semantic aspects we encoded via rotations are provided below:

- **Indicate inheritance properties**.
- **Highlight specific functional roles** or overall mission, e.g. communication, user interface, etc.
- **Illustrate particular algorithmic category**, such as numeric computations, search algorithms, or pattern matching.
- **Emphasize reusability properties** (e.g. template classes).
- **Denote design volatility** (e.g. in progress, incomplete design, under refactoring, under argumentation).

A few scenarios are provided under Figure 3, showing how spatial rotations can be deployed. In some cases, an encoding policy allows to convey the design semantics through an appropriate metaphor. For instance, the choice of a rotated placement for proxy classes in Figure 3 - top right, depicts the social metaphor of proxies as intermediaries laying in-between other modules while physically facing both of them. However, in most cases the choice of rotation encoding is done by convention rather than metaphorically, like the encoding policy for super classes shown in Figure 3 - bottom left.

The usefulness of rotated views for classes concerns primarily large designs where speed of artifact inspection, detection and recall is very crucial. In particular, they enable designers capture key design aspects directly from the design overviews (viewing with a far camera in our system) without requiring to focus closer where extra information, possibly unnecessary for the task underhand, clutters the display. The same benefit cannot be always gained by color encoding, since colors cannot be freely used to represent all sorts of design aspects: overuse of color may
result in less usable and understandable design images. In practice, blending color encoding with alternative visual encoding methods works better than color encoding alone, enabling achieve higher usability.

3.4 Encoding through Distinct Planes
Placement of items on the same plane is a way to emphasize grouping in a 3d world. When combined with other encoding methods also supporting semantic grouping, distinct subgroups in a master group may become more easily perceived than other visual techniques. For instance, two orthogonal planes of cards, initially encoded with the same color, can be very quickly recognized from varying point of view; another possibility is to use parallel planes. The geometrical placement of distinct planes is again a matter of chosen metaphoric representation: cubes, pyramids, plane layers, etc. may be designed. Overall, we have considered the following encodings related to planar placement:

- Signify (sub) grouping for a set of related classes.
- Illustrate architectural decomposition assigning distinct planes to architectural components.
- Emphasize segregation or exclusion (e.g. classes under consideration for inclusion in the final design—see Figure 4).
- Outline architectural metaphors by varying planar topologies (e.g., layered, star, etc).

The ability to populate the design space with information, not yet being part of the actual design itself, is a very important feature. In particular, it enables designers introduce or review classes that are still under consideration without shifting away the focus of attention from the main design (see Figure 4 top – ‘extension wall’ concept). In other words, such extra planes are a short of visual board where initial design may be placed. In Figure 4, the artifacts indicated as ‘main design’, are actually part of the Flying Circus design itself shown at the bottom of Figure 4.

The previous encoding tactic indicates that architectural topologies or metaphors used in the 2d world may be directly deployed in a 3d space, by supporting extra semantic grouping when put at distinct planes. For example, in Figure 5 we show how two parallel planes are used to illustrate two levels of architectural decomposition: (i) components (planes); and (ii) sub-components (classes in a plane).

3.5 Encoding through Spatial Arrangement
Free placement at distinct spatial positions allows realize a desirable topological pattern where the specific placement of design items denotes distinct semantic roles. Such a topological pattern may be known a priori, or may be totally heuristic, derived by designers after experimenting with alternative placements. There is actually no need to encode anything in particular during such a process, since the primary objective is to derive more usable and understandable representations:

- Reflect metaphors of architectural organization, like layered structures, sequential processing, etc.
- Illustrate role categories, like the use of depth-sorted placement: e.g., placing I/O classes close to the camera.
- Emphasize work in progress, such as placing classes to be elaborated later (i.e. ‘todo’ stuff) behind others.

As an example illustrating the benefits of spatial topologies we reform a view of the Observer pattern - originally in UML (see Figure 6 - top left. One possible transformation to 3d CRC design is provided in Figure 6 - option 1. In the 3d view we clearly illustrate the relationship among the Observer superclass and its concrete derivatives in a distinct plane, while putting the observed Subject class at a different spatial position (below). The deployment of the pattern in an application is provided in Figure 6 – bottom right part, showing the introduction of a client class (top thick card) that encompasses concrete observers for subject rendering purposes. Alternative topologies with extra encodings for the Observer pattern are also included in Figure 6 (options 2 and 3); in option 3, Observers are put around the Subject.
3.6 Encoding through Spatial Labeled Links

3d connections amongst spatially distinct collections of classes are directly perceived by designers, something hardly possible in a respective 2d structure, making easier the identification of component or package inter-dependencies. Spatial links, when combined with particular arrangement policies for grouped classes allow depict in an emphatic way component cross-dependencies. In particular, our tool allows inspecting call-dependencies and cross-invocations from different perspectives via auto-motion flying cameras.

Additionally, link labeling can be particularly useful since, not only it allows specialize the expectations raised for the target class (with whom collaborating), but also allows embody a micro-language (scripting) in the labels so as to carry extra information to be interpreted by accompanying tools. Typical labels we have used to convey important design information are:

- **ISA, MIXIN**: link target is a base class (normal inheritance), or the link source is a generic derived class (mixin inheritance), respectively.
- **HAS, HASMANY**: link target is a constituent object (i.e. link source is aggregate).
- **CALLS<what>**: a method of the link target is invoked by the link source; by collecting together all incoming links for a given class we may gain its public exported interface.

An example on the easiness of manipulating spatial links is provided under Figure 7, showing how the top-left top part is transformed to the top-right part by repositioning classes with the aim to increase visual comprehension.

4. Conclusions and Future Work

In software design, the use of visual metaphors and symbolisms is well-known common practice. Conceptually, visualizations are a sort of visual syntax for design semantics, affecting the way we assimilate, memorize, recall, reuse and adapt design structures. Linking to this, our work emphasized the exploitation of spatial memory, visual pattern recognition, and spatial orientation in a software design context. Overall, our primary objective has been to identify, develop and assess methods enabling the rapid visual conduct of software-design prototyping with emphasis on exploration and semantics encoding. Our results do not undermine the usefulness or usability of popular and de facto proven methods like as UML. Instead, we suggest that highly interactive visual-design instruments, like our tool, should be incorporated within existing design environments.

5. References


