Immersive 3d Visualizations for Software-Design Prototyping and Inspection

Anthony Savidis[1,2], Panagiotis Papadakos[1], and George Zargianakis[2]

[1] Institute of Computer Science, Foundation for Research and Technology – Hellas
[2] Department of Computer Science, University of Crete
{as,papadako,zargian}@ics.forth.gr

Abstract. In software design, physical CRC cards (Classes – Responsibilities - Collaborators) is a well-known method for rapid software-design prototyping, heavily relying on visualization and metaphors. The method is commonly applied with heuristics for encoding design semantics or denoting architectural relationships, such as card coloring, size variations and spatial grouping. Existing software-design tools are very weak in terms of interactivity, immersion and visualization, focusing primarily on detailed specification and documentation. We present a tool for visual prototyping of software designs based on CRC cards offering: 3d visualizations with zooming and panning, rotational inspection and 3d manipulators, with optional immersive navigation through stereoscopic views. The tool is accompanied with key encoding strategies to represent design semantics, exploiting spatial memory and visual pattern matching, emphasizing highly interactive software visualizations.

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, enabling immersive navigation, while supporting encoding of design semantics 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 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 visual software-design prototyping instrument apart of teaching and

G. Bebis et al. (Eds.): ISVC 2008, Part I, LNCS 5358, pp. 879–890, 2008.

process description. CRC cards emphasize the exploratory and visual nature of software-design prototyping, allowing heuristic visual encodings and symbolisms, 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, very little is done in supporting computer-assisted design as a visualization-centric activity offering advanced 3d features for design exploration and semantics encoding.

Contribution. We present a tool for exploratory software-design prototyping, aimed to be utilized 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.

2 Related Work

Quick CRC [Quikc 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 3d Design Encoding and Exploration

Typical interactive configuration features concern the control of visualization parameters like camera, planes, axes, and special feedback effects (see Fig. 1). The dialogue box for editing card attributes (bottom-right of Fig. 1) is a transparent non-modal billboard surface, allowing design actions like navigation and focus change to be freely applied, applying attribute updates to the current focus card (class). Card visual attributes, like rotation angle, dimensions, color and positioning, are to be exploited by designers as heuristic symbolic vocabularies for encoding design semantics.

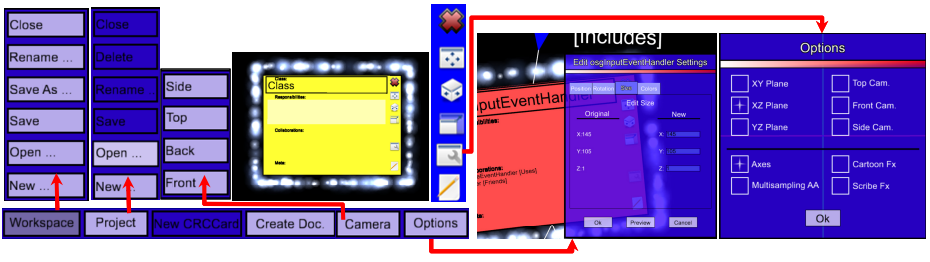

Fig. 1. From left-to-right: workspace / project menus (arrows denote activation), camera control, camera view options, in card manipulators, and properties dialogue boxes

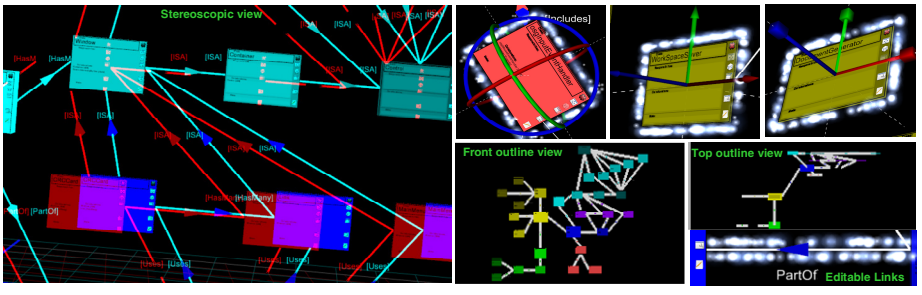

Fig. 2. Left: stereoscopic anaglyphic view; Right top: spatial rotation / repositioning / resizing via 3d manipulators; Bottom: outline views and links

The tools for 3d design are shown under Fig. 2: 3d rotation, repositioning and resizing are possible via special-purpose 3d manipulators appearing automatically when the respective in-card controls (see Fig. 1 middle) are clicked. Next we elaborate on the way interactive configuration of visual parameters for 3d cards allows designers introduce informal visual vocabularies to express semantic properties of the software design. We recall that the emphasis is shifted from an agreed unified vocabulary of standardized semantics for detailed or exhaustive design as in UML, to open add-hoc policies for visual representation of semantic information with emphasis on visual, rapid, exploratory prototyping. We present a set of encoding strategies that we have applied and assessed in the course of real practice. The set is not closed; we aim to demonstrate the expressive power of visual tools in handling software design artifacts.

3.1 Encoding through Varying Dimensions

With variations of dimension we may encode quantitative design attributes, such as individual source size, overall 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 well be annotated with information that is known or predicted prior to implementation, or being consolidated after the implementation phase is initiated and probably entirely completed. Proposed encoding policies we have 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.

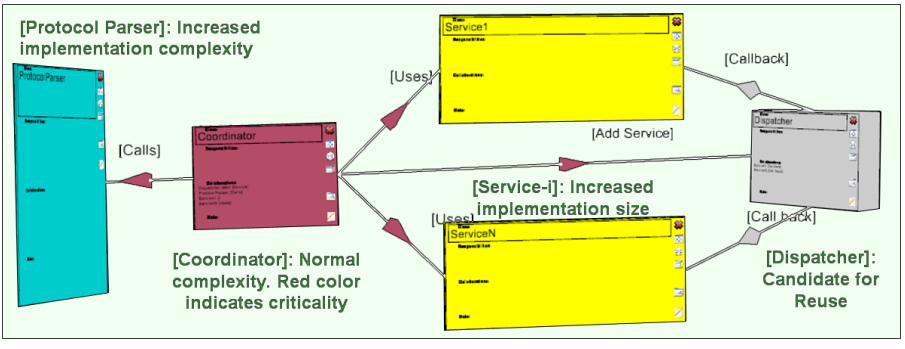

Fig. 3. Encoding components with varying dimensions in the software design of a typical server architecture

An example is provided Fig. 3 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 known to be 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.
- Highlight *inheritance properties*, like abstract classes, interfaces, generic derived classes (i.e., mix-in inheritance).

Practically, color encoding alone is not sufficient for effective recognition of distinct artifacts and group relationships in severely crowded software-design spaces, unless appropriately backed-up with high-quality inspection / exploration facilities. For example, assume a designer working locally in a particular design context needing to temporarily suspend current activities to review quickly another part of the design

space using color encoding knowledge, and then return back to the previous context resuming design work. In general, such a sequence of suspend, switch context, and resume design steps is common in design processes, reflecting the fact that designers frequently assess or recall interrelationships with other parts of a software system.

In our tool, such switching is both fast and usable due to the immersive animated navigation facilities. In particular, the designer may quickly zoom-out, spot the target context via color encoding, zoom-in to focus and inspect it temporarily, and then apply zoom-out and zoom-in to return at the previous context. Alternatively, the designer may bookmark the current position so as to return automatically after such design reviewing activities. Multiple spatial bookmarks are supported, with a typical circular iteration style as in most document editors; we plan to include named bookmarks that include a displayed editable text-field.

A similar sequence as above would be also performed using UML visual design tools. However, the offered zoom / focus control navigation facilities are not made to be that fast and immersive, resulting in valuable time being lost to handle design context switching. Moreover, when it comes to comprehensive and crowded design spaces, all UML design elements should be arranged in a single plane, meaning the inspection area (view frustum) becomes far larger in comparison to our system.

3.3 Encoding through Rotations

Spatial rotation of objects may imply emphasis, distinction, and semantic separation, and can be used to directly attract visual attention (we tested that planar rotations over 30° are very clear). 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 Fig. 4, 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 Fig. 4 - 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 by convention rather than metaphorically, like the encoding policy for super classes shown in Fig. 4 - bottom left. The usefulness of rotated views for classes concerns primarily large designs where speed of artifact inspection, detection and recall is very crucial.

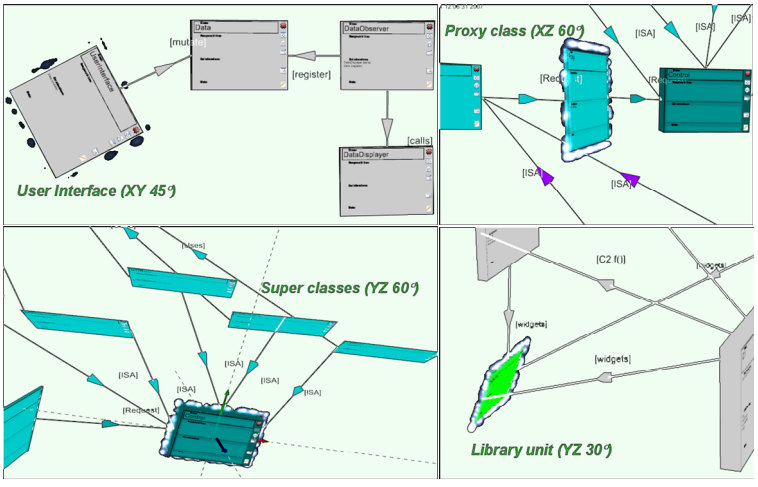

Fig. 4. Sample encodings via rotations: UI (top left), super classes (bottom left), proxies (top right) and library modules (bottom right)

In particular, rotations enable designers capture key design aspects directly from the design overviews (viewing with a far camera in our system) without requiring 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 compared to the use of color encoding alone.

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 visual grouping methods, distinct subgroups in a master group may become easily perceived. 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.
- Outline *architectural metaphors* via planar topologies (e.g., layered, star, etc).
- Illustrate *architectural decomposition* assigning distinct planes to architectural components.
- Emphasize *segregation or exclusion* (e.g. classes under consideration for inclusion in the final design).

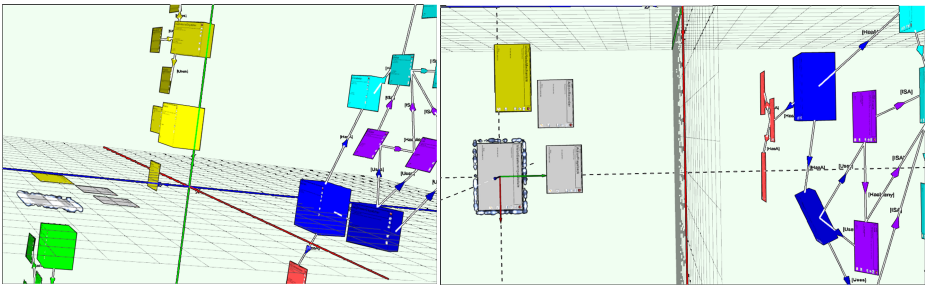

Fig. 5. The ‘extension wall’ concept (left-bottom, right-left) realized as a distinct plane of the main design (partially shown at to the right of the extension wall)

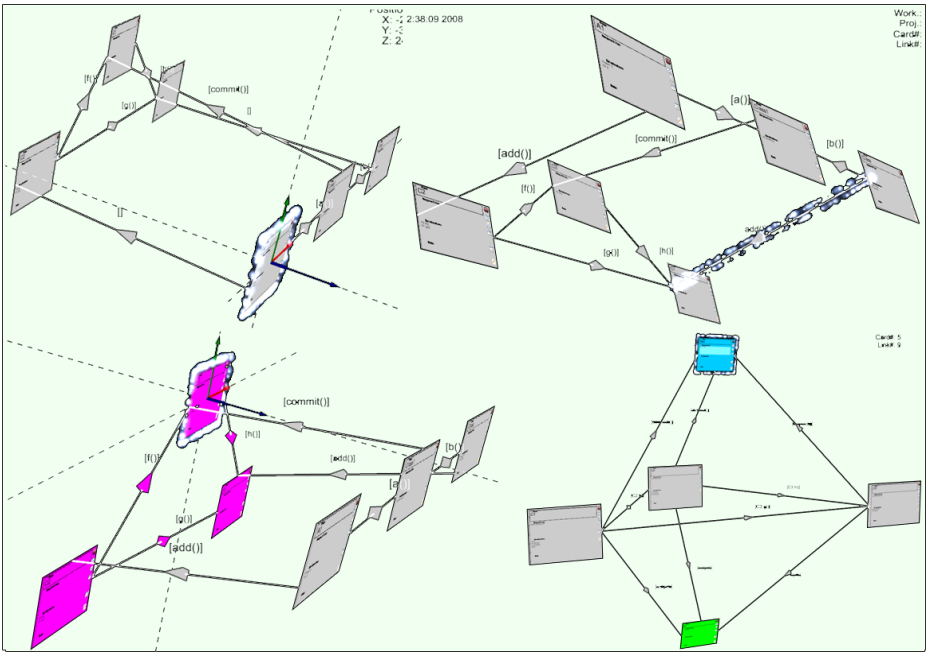

Fig. 6. (a) Views of a design organized in two planar groups for architectural components (top left, top right, bottom left); (b) component grouping with a pyramid metaphor (bottom right)

The ability to populate the design space with information not yet being part of the design itself is a very important feature. In particular, it enables designers introduce classes or packages that are still under consideration without shifting focus of attention from the main design space, like the ‘extension wall’ shown at Fig. 5.

The main design of Fig. 5, partially shown at right part of sub-images, is the tool design itself (outlined in Fig. 8). The last encoding shows that 2d architectural metaphors may be deployed in a 3d space through extra grouping by distinct planes. For example, in Fig. 6 we use two parallel planes to illustrate architectural decomposition: (i) components (planes); and (ii) sub-components (classes in a plane). Additionally, a

more comprehensive spatial organization metaphor like a pyramid is chosen for Fig. 6 – bottom right. Such a spatial organization supporting real-time inspection from any point of view allows severely reduce the visual complexity of the design image when compared to traditional planar topologies. Again, the organization of the design into distinct planes may be also combined with other types of encodings like planar / spatial rotations, varying dimensions, color differentiation (Fig. 6, top right), etc.

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 latter (i.e. 'todo' stuff) behind others.

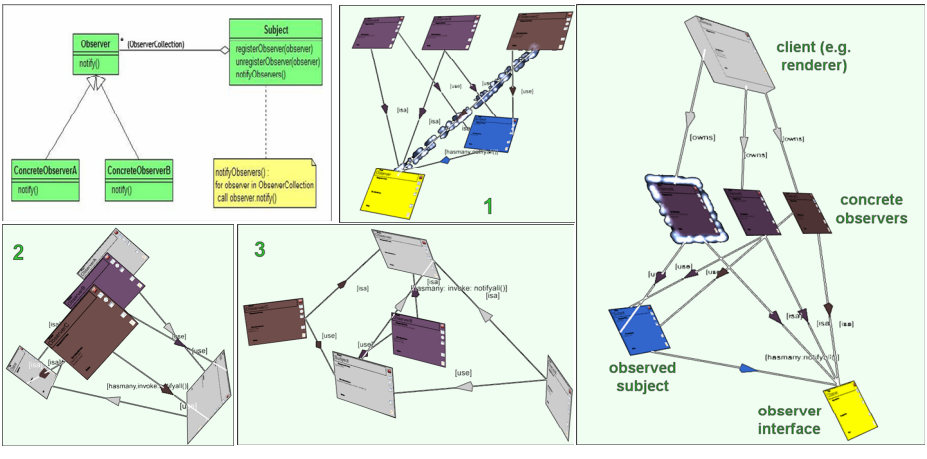

Fig. 7. The incremental spatial reform of the originally planar Observer design pattern

As an example illustrating the benefits of spatial topologies we reform a view of the Observer pattern - originally in UML (see Fig. 7 - top left). In this view, the fact that concrete observers access the subject is not shown: if we introduce the necessary links for the latter, its UML image gets a little cluttered. One possible transformation to 3d CRC design is provided in Fig. 7 - label 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 different associations become more evident.

The deployment of the pattern in an application is provided in Fig. 7 - right part, showing the introduction of a client class (top) that encompasses concrete observers for subject rendering purposes. Alternative topologies with extra encodings for the Observer pattern are also included in Fig. 7. For instance, we may organize concrete observers as parallel cards (label 2), or place concrete observers around the subject (placed in the middle) to give a social-metaphor connotation (label 3).

3.6 Animated Immersive Inspection

While the support for navigation is not related to visual encoding facilities, it is very important since it is essential in enabling the effective and efficient conduct of the design activities. We put emphasis on activities such as: design exploration, reviewing of relationships and dependencies, shifting focus of attention, viewing design context of spatially proximate classes, and effective visual control over large design structures. Automatic animated reviews are initiated through specific mouse gesture; the animation speed is proportional to the speed of the mouse gesture; e.g., if the designer is making a gesture very fast, the animation is also fast. We support reviews for model rotation, zooming, and panning. An example is provided under Fig. 8, demonstrating the way complexity is reduced with spatial organization and visual encoding, and how spatial navigation is an effective global supervision and exploration activity.

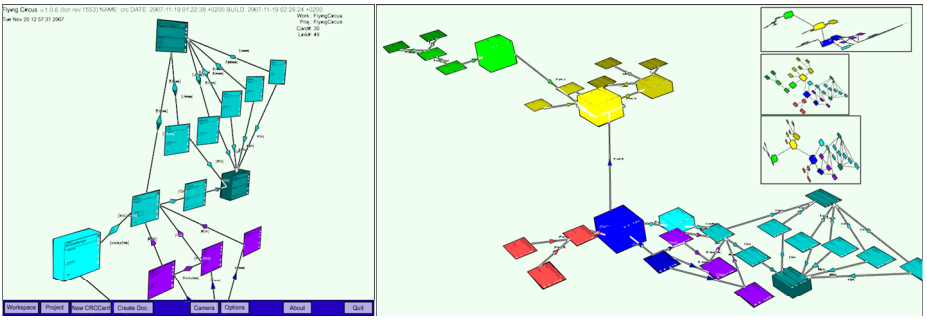

Fig. 8. Views of the Flying Circus design in itself – miniature resident views are also supported, shown at top right

Mentally linking a design structure with a visual pattern is very crucial for design memorization. In this context, deriving an appropriate visual pattern (like the one shown in Fig. 8) is far easier to accomplish with a spatial topology in comparison to planar structures, meaning designers have far more chances to structure suitable design representations. Additionally, inspection and identification of component dependencies is an activity that is emphasized in our tool. For instance, bilateral method invocations may introduce undesirable coupling that should be rectified. Apparently, visual recognition through link arrows is far more efficient compared to extrapolating dependencies from the source code. The advantage over planar associations is that a dependency is easier assimilated when seen from varying perspectives (see Fig. 9).

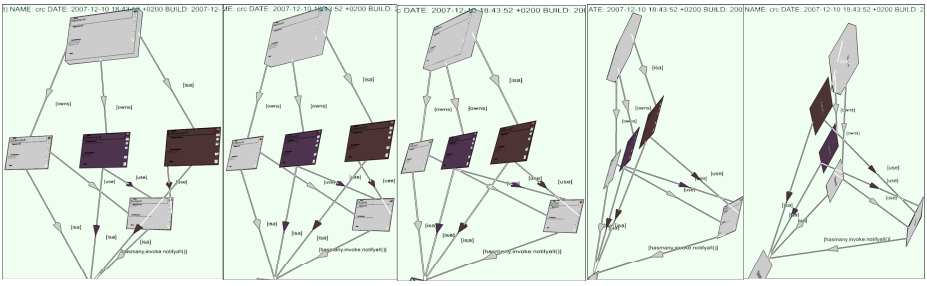

Fig. 9. Inspecting call-dependencies and cross-invocations from different perspectives via auto-motion flying cameras

3.7 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.

Editing the visual design and topologically reorganizing classes and links, to make the design more readable, apparently presents far more possibilities in the 3d space compared to the 2d world, since overlapping of links can be eliminated with appropriate card placements. Although the 3d structure is still projected on the screen as a planar artifact, the facilities for navigation allow designers inspect their designs from varying angles and zooming factors so that a chosen links does not overlap with others on the screen.

An example on the easiness of manipulating spatial links is provided in Fig. 10, showing how the left part is transformed to the middle part by repositioning classes with the aim to increase visual comprehension, by making links cleaner and revealing a hierarchical association discipline. From the interaction point of view, such a transformation takes only a couple of minutes to accomplish. Combined with semantic grouping the links may convey different types of dependencies at the architectural

level: (i) planar links inside the same groups represent intra-component associations; and (ii) spatial links among distinct groups denote inter-component dependencies. The encoding method to denote grouping could be coloring or planar placement.

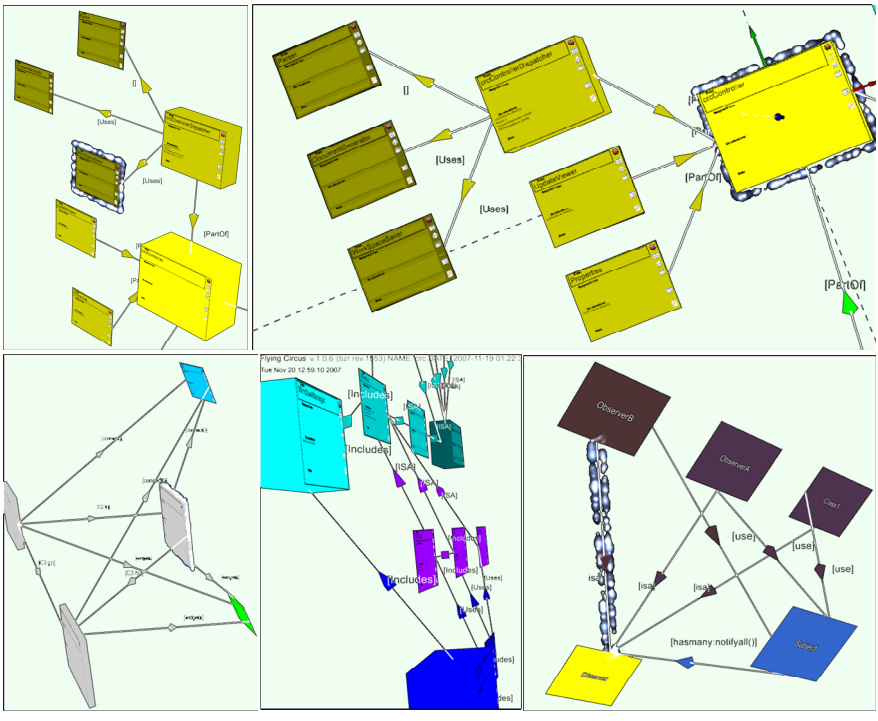

Fig. 10. Rearranging links to make dependencies more clear (top), adoption of spatial links for invocations (bottom left), and various spatial linking topologies in different designs (bottom)

4 Conclusions and Future Work

In software design, the use of visual metaphors and symbolisms is well-known common practice. We frequently seek for architectural structures that encompass the key semantic aspects, being simultaneously more understandable and easier to memorize. The latter is crucial not only during the early phases where initial designs are shaped out, but also as systems continue to grow and evolve. In this context, visualizations are a sort of visual syntax for design semantics, affecting the way we assimilate, memorize, recall, reuse and adapt design structures. The latter relates to the human mechanism of cognition, since the perceived complexity of a system of objects is affected by the way we represent involved artifacts and relationships. Linking to this, we consider that our capacity for spatial memory, visual pattern recognition, and spatial orientation needs to be further exploited in a software design context. Technically, the focus of our work was far from questioning the usefulness of popular and practically approved methods such as UML. Instead, our primary objective has been to identify, develop

and assess methods enabling the visual conduct of design prototyping with emphasis on exploration and semantics encoding. Our results may be incorporated in the form of specialized tool-boxes within existing design environments.

We were motivated by the fact that software design, a critical activity of the software life cycle, is still carried out with instruments offering traditional graphical means of interaction that haven't essentially progressed within the last decade. In the meantime, for code analysis and behavior monitoring / observation there are good tools offering very advanced visualization features. We consider this to be an imbalance that cannot be merely addressed by having more automatic post-design or post-coding visualization tools. We consider that the design activity should be genuinely supported as an exploratory visualization-centric process. Clearly, in such a context, automatic visualizers always play a crucial role as they reveal alternative system perspectives. [*The tool is available from: http://www.ics.forth.gr/hci/files/plang/FLYINGCIRCUS.ZIP*]

References

QUICK CRC. 2001 (2001), `http://www.excelsoftware.com/crccards.html`

Beck, K., Cunning, W.: A laboratory for teaching object-oriented thinking. In: Proceedings of the ACM OOPSLA 1989 Conference, New Orleans, Louisiana, pp. 1–6. ACM Press, New York (1989)

Byelas, H., Telea, A.: Visualization of areas of interest in software architecture diagrams. In: ACM Symposium on Software Visualization (SoftVis), pp. 105–114. ACM Press, New York (2006)

Fronk, A., Bruckhoff, A.: 3d visualization of code structures in Java software systems. In: ACM Symposium on Software Visualization (SoftVis), pp. 145–146. ACM Press, New York (2006)

Greevy, O., Lanza, M., Wysseier, C.: Visualizing live software systems in 3d. In: ACM Symposium on Software Visualization (SoftVis), pp. 47–56. ACM Press, New York (2006)

Parnin, C., Goerg, C.: Lightweight visualizations for inspecting code smells. In: ACM Symposium on Software Visualization (SoftVis), pp. 171–172. ACM Press, New York (2006)

Raman, A., Tyszberowicz, S.: The easycrc tool. In: Proceedings of the IEEE International Conference on Software Engineering Advances ICSEA 2007, pp. 52–52. IEEE Press, Los Alamitos (2007)

Roach, S., Vasquez, J.: A tool to support the crc design method. In: Proceedings of the International Conference on Engineering Education (2004), `http://www.succeed.ufl.edu/icee/Papers/339_Roach-Vasquez(1).pdf`

Wirfs-Brock, R., Mckean, A.: Object Design-Roles, Responsibilities, and Collaborations. Addison-Wesley, Reading (2003)