CREATIVE DESIGN IN OBJECT-ORIENTED ENVIRONMENTS

AUTHORS

Zenon Rychter
Faculty of Architecture, Technical University of Bialystok,
Krakowska 9, 15-875 Bialystok, Poland
rychter@cksr.ac.bialystok.pl

ABSTRACT

Object-oriented approach to software development is discussed as a conceptual framework and working computational model for creative architectural design. Two modes of object orientation in design are elaborated. The more conservative mode is static, based on class-type/object-instance hierarchies. The other mode is dynamic, based on a modern view of computation as multi-threaded evolution of interacting objects.

CREATIVE DESIGN IN OBJECT-ORIENTED ENVIRONMENTS[1]

The Internet and Web as computing environment

Today, an architectural design process involves users, designers and computers. While users and designers, being humans, are rather conservative and set in their ways, computers are undergoing explosive growth and revolutionary change. Computing power has become ubiquitous. The information highway commonly called the Internet reaches everywhere. The World Wide Web (Klander 1997) is still in its infancy being only 7 years old but it is here to stay as technology grows, with over 80 million sites on the Internet now and limitless boundaries as to what you can do. The most important ingredient to the success of the Internet has been its neutrality. No matter what country you live in, what language you speak or even what operating system you are running, you can access the nearly unlimited information published on the Internet. We owe this to WWW standards. These standards encompass everything from programming languages used on the Internet to connection protocols like TCP/IP or modem command strings. The core development has been language standardization. Back in 1986, with literally thousands of word processing programs out on the market, each of which encoded text in a different way when saved to disk, it was virtually impossible to create a single text file that could be viewed by every operating system in existence. And if something as simple as text could not be portable, then one could forget about graphic files or program files. At the time, the Internet as we know it was not possible, simply because the operating systems and programs of the day had major difficulties translating files written in a foreign program. This is where Standard Generalized Markup Language (SGML) came in, a meta-language, a guideline for how other languages should be developed. The foundation for the web sites we see today comes from SGML, without which there would have probably developed an Internet for Microsoft OS systems, another for systems running Mac OS, and yet another for UNIX. Since its inception, several sub-languages have developed out of SGML. The most notable of these is Hyper Text Markup Language (HTML), however additional languages such as Virtual Reality Markup Language (VRML) and Extensible Markup Language (XML) were also developed out of the mold of SGML. HTML was originally intended only to handle the formatting of text. Only now is HTML broadening its horizons to encompass more than text. HTML 2.0 was the first revision of HTML to actually be solidified into a standard, which was released in September 1995. HTML 3.2 was the next revision of the language, published in May of 1996. HTML 4.0 became a standard in 1997. Previous versions of HTML were developed to match the capabilities of the browsers currently deployed on the market. This has changed in HTML 4.0, which is the first attempt to really expand the functionality of the language by adding support for object tags, as well as support for Cascading Style Sheets (CSSs). The object tag is notable because it enables scripts, code from scripting languages, to be embedded directly in the HTML source code for a site. The development of the object tag has a direct relation to the Object-Oriented Model for programming languages, found in application programming languages for years now. Between the object tag

[1] This research was sponsored under grant S/WA/5/97 from Technical University of Bialystok

and Cascading Style Sheets, the possibilities for the coding of a web site will be limited only to the imagination of the site developer. Dynamic HTML is Microsoft's latest advance in HTML, intended to make the language more functional by adding abilities normally found in application programming languages like C++ (Stroustrup 1993) or Java (Jamsa 1996). For the first time, a site developer can code an HTML document that uses animated text, can be set for timed events, scrolling text as well as the ability to access databases all without the need for additional components. Extensible Markup Language (XML) was announced as a proposed standard in December 1997. Like SGML, XML is not really a programming language, but a meta-language. However, whereas SGML was designed as a universal code for the formatting of text, XML is designed as a universal code for the formatting of data, a universal code for how Internet applications share information between one another. As the abilities of online technologies like HTML develop, the differences between web sites and applications are beginning to fade. However, a major hindrance to programmers who wish to develop online applications is the way that current programming languages encode their information. Because of this, it is exceedingly difficult for developers to create web applications intended to share information like databases, spreadsheets or even games simply because the developer has no control over whether or not the software installed on the user's system will be capable of interpreting and using the information delivered to it. This is where XML comes in. XML is designed as an open classification, whereby packets of data transmitted from one computer to another will contain all the information needed to be able to decode itself using any application that can handle XML. Just as SGML enabled electronic text to be useable by any computer with an SGML capable browser installed, XML provides this same function for data itself.

There is no limit in the Web specifications to the graphical formats that can be used on the Web. Vector Graphics (SVG) is a recent language for describing two-dimensional graphics in XML. SVG allows for three types of graphic objects: vector graphic shapes, images and text. Graphical objects can be grouped, styled, transformed and composited into previously rendered objects. The feature set includes nested transformations, clipping paths, alpha masks, filter effects, template objects and extensibility. SVG drawings can be dynamic and interactive. SVG allows for straightforward and efficient vector graphics animation via scripting. A rich set of event handlers such as onmouseover and onclick can be assigned to any SVG graphical object. Because of its compatibility and leveraging of other Web standards, features like scripting can be done on HTML and SVG elements simultaneously within the same Web page.

Mathematical Markup Language (MathML) is a recent low-level specification for describing mathematics as a basis for machine to machine communication. It provides a much needed foundation for the inclusion of mathematical expressions in Web pages. MathML is intended to facilitate the use and re-use of mathematical and scientific content on the Web, and for other applications such as computer algebra systems, print typesetting, and voice synthesis. MathML can be used to encode both the presentation of mathematical notation for high-quality visual display, and mathematical content, for applications where the semantics plays more of a key role such as scientific software or voice synthesis. MathML is cast as an application of XML. As such, with adequate style sheet support, it will ultimately be possible for browsers to natively render mathematical expressions. MathML consists of a number of XML tags which can be used to mark up an equation in terms of its presentation and also its semantics. MathML attempts to capture something of the meaning behind equations rather than concentrating entirely on how they are going to be formatted out on the screen. This is on the basis that mathematical equations are meaningful to many applications without regard as to how they are rendered aurally or visually.

With the launch of Java language and Jini technology, Sun kicks off a new era of distributed computing. Businesses are leaving behind a motley system of fax, phone, and proprietary software by shifting to Java technology. In Boston Computer Museum, thanks to the virtual fish tank exhibit, visitors can interact with playful, brightly colored fish propelled by Java technology. Embedded Java will support real-time capabilities, providing consumer and industrial devices, such as mobile phones, pagers, and medical devices, with a precise, predictable response time and the ability to coordinate functions without any lapse between operations. With Jini technology we can imagine a huge global network with a host of myriad devices and it all just works. Java applets, those nimble servants of the network, are popping up in browsers everywhere. Servlets support dynamic web site applications in ways never dreamed of. The Java platform is making major inroads into every area of computing, offering portable, highly scalable, multiplatform, and simple to use, "Write Once, Run Anywhere" application development. Jini Technology makes computers and devices able to quickly form impromptu systems unified by a network. Such a system is a federation of independent, flexible, smart devices, including computers, that are simply connected. Within a federation, devices are instant on, no one needs to install them. The network is resilient, you simply disconnect devices when you don't need them. This creates a work environment where the tools are ready to use and largely invisible. Jini technology provides simple mechanisms which enable devices to plug together to form an impromptu community, a community put together without any planning, installation, or human intervention. Each device provides services that other devices in the community may use. These devices provide their own interfaces, which ensures reliability and compatibility. Devices permeate our lives: TVs, VCRs, DVDs, cameras, phones, PDAs, radios, furnaces, disk drives, printers, air conditioners, CD players, pagers, and the list goes on. Today devices are unaware of their surroundings, they are rigid and cannot adapt. When you buy a disk drive, you expend a lot of effort to install it or you need an expert to do it for you. But now devices of even the smallest size and most modest capabilities can affordably contain processors powerful enough for them to self-organize into communities that provide the benefits of multi-way interactions. A device can be flexible and negotiate the details of its interaction. We no longer need a computer to act as an intermediary between a cell phone and a printer. These devices can take care of themselves, they are flexible, they adapt. A device can take charge of its own interactions, can self-configure, self-diagnose, and self-install. When computers were the size of large rooms, it made sense to have a staff of people to take care of them. But now technology creates the possibility of impromptu, self-managing device communities popping up in all kinds of places far from any system administrator. The final stretch of the computational power dimension is that now processors are powerful enough to support a high-level, object-oriented programming language in such a way to support moving objects between them. And such a processor is small enough and cheap enough to sit in the simplest devices. Once there is sufficient computational power, the ability to connect and communicate is the dominant factor. Today for most people, a computer runs only a few applications and mainly facilitates communication: email, the Web. Internet's popularity soared first with email and more recently once the Web and browsers became prevalent. When the Internet was developing, there were two essential activities: defining and perfecting the underlying protocols and infrastructure, and creating applications and services on top of that infrastructure: email composers and readers, file fetching programs, Web browsers, and the Web itself. No single company or organization did all the work, and none could, if the venture was to be successful, because underlying it all is a standard protocol, and a protocol can be successful only if it is widely adopted. The Java programming language is the key to making Jini technology work. Java was originally developed for programming intelligent home appliances. Java leverages software reuse across projects, tools, and architectural layers. Astronomers and engineers use Java to monitor and control second-by-second movements of the Hubble Space Telescope. A sophisticated Java applet controls a rover on the surface of Mars. Java gadgets in the works include a finger ring that opens locked doors, a home telephone that offers movie reviews and ticket ordering, a wireless pen-based network computer that manages on-the-floor inventories. Applets run from simple to sophisticated, and uses range from management of business transactions, to sharing and analyzing scientific data, to gaming and consumer-oriented purposes, generally to breathing intelligence into the devices you'll use every day.

The Internet(technology)/WWW(language) story demonstrates that technology cannot be exploited to the full without a proper standard, commonly understood language. We need the language first to conceptualize the system we are interested in and, second, to interact with the system, control it, or be a part of it. Object-Oriented Languages (OOLs) provide both the conceptual framework (Coad, Yourdon 1991) and computational efficiency to model the most complex systems, the heterogeneous Internet and architectural design included. OOLs, notably C++, have been the standard for system programming for some time now. But it is only recently that an OOL, Java, is becoming a standard on the Internet. The structure of the Internet is similar to the brain (Klander 1997). Thanks to Java we can understand the workings of this brain and can plug-in our own brains. The Internet plus Java form an incredibly rich computing environment, on Object-Oriented-Environment (OOE), ready to support all computing needs and styles.

OOEs are relevant to creative, conceptual architectural design on two levels, static and dynamic, which correspond to two modes or models of architectural design, static-sequential-manual-controlled and dynamic-multithreaded-asynchronous-automatic.

Static object-oriented approach: design from components

In the static-sequential mode, developing an architectural design is likened to developing a windowed application with a graphical user interface (GUI) or an interactive Web site. The basic concept is the metaphor of objects (Stroustrup 1993). This is a natural way we interpret and interact with the world around us. Objects, whether real-world or computer representations, are described in terms of what they are and how they behave. Objects have certain characteristics or attributes, called properties, that define their appearance or state—for example, color, size, and modification date. Properties are not limited to the external or visible traits of an object. They may reflect the internal or operational state of an object, such as an option. Things that can be done with or to an object are considered its operations. Moving or copying an object are examples of operations. You can expose operations in the interface through a variety of mechanisms, including commands and direct manipulation. Objects always exist within the context of other objects. The context, or relationships, that an object may have often affects the way the object appears or behaves. Common kinds of relationships include collections, constraints, and composites. As in the natural world, the metaphor of objects implies a constructed environment. Objects are compositions of other objects. You can define most tasks supported by applications as a specialized combination or set of relationships between objects. A text document is a composition of text, paragraphs, footnotes, or other items. A table is a combination of cells; a chart is a particular organization of graphics. When you define user interaction with objects to be as consistent as possible at any level, you can produce complex constructions while maintaining a small, basic set of conventions. In addition, using composition to model tasks encourages modular, component-oriented design. This allows objects to be adapted or recombined for other uses. Applying object-based concepts offers greater potential for a well-designed interface. As with any good user interface design, a good user-centered design process ensures the success and quality of the interface. The first step to object-based design should begin with a thorough understanding of what users' objectives and tasks are. When doing the task analysis, you should identify the basic components or objects used in those tasks and the behavior and the characteristics that differentiate each kind of object, including the relationships of the objects to each other and to the user. Also identify the actions that are performed, the objects to which they apply, and the state information or attributes that each object in the task must preserve, display, and allow to be edited. An effective user-centered design process involves a number of important phases: designing, prototyping, testing, and iterating. The initial work on a software's design can be the most critical because, during this phase, you decide the general shape of your product. If the foundation work is flawed, it is difficult to correct afterwards. This part of the process involves not only defining the objectives and features for your product, but understanding who your users are and their tasks, intentions, and goals. This includes understanding factors such as their background — age, gender, expertise, experience level, physical limitations, and special needs; their work environment — equipment, social and cultural influences, and physical surroundings. Ideally, you want to create a design model that fits the user's conceptual view of the tasks to be performed. You should consider the basic organization and different types of metaphors that can be employed. Observing users at their current tasks can provide ideas on effective metaphors to use. After you have defined a design model, prototype some of the basic aspects of the design. A prototype is a valuable asset in many ways. First, it provides an effective tool for communicating the design. Second, it can help you define task flow and better visualize the design. Finally, it provides a low-cost vehicle for getting user input on a design. This is particularly useful early in the design process. User-centered design involves the user in the design process. Usability testing a design, or a particular aspect of a design, provides valuable information and is a key part of a product's success. There can be different reasons for testing. You can use testing to look for potential problems in a proposed design. You can also focus on comparative studies of two or more designs to determine which is better. Usability testing provides you not only with task efficiency and success-or-failure data, it also can provide you with information about the user's perceptions, satisfaction, questions, and problems. When testing, it is important to use participants who fit the profile of your target audience. Because testing often uncovers design weaknesses, or at least provides additional information you will want to use, you should repeat the entire process, taking what you have learned and reworking your design or moving onto reprototyping and retesting. Continue this refining cycle through the development process until you are satisfied with the results.

Working in OOEs is further simplified by using classes. For example, the classes in the Microsoft Foundation Class Library (MFC) make up an "application framework" — the framework on which you build an application for Windows. At a very general level, the framework defines the skeleton of an application and supplies standard user-interface implementations that can be placed onto the skeleton. Your job as programmer is to fill in the rest of the skeleton — those things that are specific to your application. You can get a head start by using the application wizard to create the files for a very thorough starter application. You use resource editors to design your user-interface elements visually, class wizard to connect those elements to code, and the class library to implement your application-specific logic. Your role in configuring an application with the framework is to supply the application-specific source code and to connect the components by defining what messages and commands they respond to. You use standard OOL techniques to derive your own application-specific classes from those supplied by the class library and to override and augment the base class's behavior. It is crucial to understand the relationship between your source code and the code in the framework. When your application runs, most of the flow of control resides in the framework's code. The framework manages the message loop that gets messages from Windows as the user chooses commands and edits data in a view. Events that the framework can handle by itself don't rely on your code at all. For example, the framework knows how to close windows and how to exit the application in response to user commands. As it handles these tasks, the framework gives you opportunities to respond to these events as well. But your code is not in the driver's seat, the framework is. You can use the framework to write sophisticated programs, even if you are not an experienced programmer. The reason is the component library: a collection of routines that make it easy to work with the Windows environment. You don't need to understand the low-level workings of Windows. You don't have to be able to define a class. You just have to learn a few simple techniques for using the library components. At the same time, experienced programmers can use the full facilities of OOL whenever necessary. OOE provides the best of both worlds: a simple way to produce application programs using a comprehensive component library and the full power of a standardized programming language for those with more far-reaching requirements.

All said above about developing GUI applications in OOEs is valid for developing architectural designs: user-centered design, the design cycle, the skeleton-framework-class-object hierarchical approach, the available framework tools. It is hard to define what's creativity about, but ignoring what OOEs already has in store is anything but creative. It's reinventing the wheel, ignoring what is readily available. Within an OOE each designer can draw upon and add to what others have done in the form of architectural class hierarchies. If you come across an idea of general utility, make it a class or component and publish it on the Web, for all to use and re-use. If you can be more specific than others, derive your class from someone else's more abstract class. If a composition of features is needed, derive your class from several parents. An architectural style can become a class, or a hierarchy of classes. An individual architect or a group can develop their own style, based on someone else's effort. All sorts of legal codes and technical standards, once put into classes, will be automatically fulfilled in designs that use objects-instantiations of such classes, with obvious increase of design quality. Clearly developing useful classes is creative. And the effort put into transforming the subjective and vague ideas in the designer's brain into the objective structure and intelligence of a class is illuminating. It also enables sharing with others, humans or machines. But it is equally creative finding the best of available classes for a design job, customizing the objects, and turning them into a harmonious whole. And because the Web is becoming an OOE, orchestrating distributed design by a community of networked users, designers, consultants may soon be more creative and productive than the current, disintegrated approach, where each designer and each design is an isolated island.

Dynamic object-oriented approach: design as a networked game
The second, radical view of architectural design is based on a whole new way of thinking about computation, where computations are about concurrent interactions with users, networks, and environments. Almost any software program of significance today is composed of concurrent interactions among communities of entities. Today's programmer can never be ignorant of the context in which a program will run. There are always many things going on, inside your program and around it. That means you need to know how to think concurrently. The questions become: What are the services my system provides? Who are the entities that make up the community that is my system? How do these entities interact to provide those services? In the old system, computation is a process of sequencing steps, there is a single thread of control and you own it, and all you have to do as a programmer is tell the computer what to do next. That's not reality. A computer is a system of continuously, simultaneously interacting parts, you can't really isolate the pieces. Today, user actions often affect the computation's execution. And increasingly, computations have pieces that run simultaneously. The Java language presumes these types of computations at the most fundamental level. In designing the language, the Java developers were aware that computation is about constituting communities of interacting entities. The fact that threads are a part of Java is evidence of that. The fact that GUIs were built into Java is evidence. The fact that networking was built in is a natural

consequence of that thinking. This isn't surprising because Java technology was originally designed for set-top boxes, which are embedded systems that have to interact simultaneously with users and televisions and networks. They have to be able to handle a lot of things going on, if not concurrently, at least conceptually concurrently. It's not at all surprising that the Java language was the right language for programming the Web. Because in building a programming language that would fit the needs of embedded systems, the people at Sun were building something that anticipated that things would be coupled together concurrently rather than sequentially, and that the world was full of an ever-expanding community of entities, each of which might be communities within themselves. Java is a simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language. But we are still entrenched in an outdated computational model: a single-thread-of-control static problem-solving view of the role of the computer program, and computation as calculation, which does not correspond to our computing environments. Instead, we should conceptualize computation with a model of computer programs as simultaneous ongoing entities embedded in and interacting with a dynamic environment: computation as interaction, computation as it occurs in spreadsheets and video games, web applications and robots. Perhaps the most fundamental idea in modern computer science is that of interactive processes. Computation is embedded in a (physical or virtual) world; its role is to interact with that world to produce desired behavior. While von Neumann serial programming has it that computation-as-calculation uses inputs at the beginning to produce outputs at the end, computation-as-interaction treats inputs as things that are monitored and outputs as actions that are taken over the lifetime of an ongoing process. Current practice changes the fundamental vocabulary of computation: not only are functions replaced by interaction patterns, but these interaction patterns take place concurrently and asynchronously. Programming no longer (necessarily) involves designing the flow of control in a system; instead, it is fundamentally about constituting a community of autonomously interacting entities, deciding what goes inside and what goes between them. Systems today are increasingly difficult to describe in the old paradigm. Networks, distributed computing, and concurrency are integral parts of the computational world. And languages such as Java force us to confront these issues. In the new vision, accounting for the role of the user becomes straightforward: the user is another member of the community of interacting processes that together constitute our computation. A program of this type is judged by the set of services or behaviors it provides. This description characterizes robots and software agents; it is descriptive of operating systems and network services; it fits video games and spreadsheets and modern word processing programs, as well as the controllers for nuclear power plants and automotive cruise controls. In modern computational systems, much of the interesting behavior is generated not by individual components per se, but by the interactions among these components. New approaches to computation are needed at many levels, from theoretical foundations to design methodologies. This is less a shift in the systems that we build and more an upgrading of our understanding of these systems. Every Java program with a graphical user interface is inherently concurrent. We must get rid of our sequentialist illusions and live from the very beginning in a potentially concurrent, distributed, interactive, embedded environment. The atomic unit of the new vocabulary is an infinite loop that senses and reacts, handles service requests in turn, or behaves. This approach to computation has the potential to build bridges with neighboring and less obviously related disciplines. The community model of computation is similar to foundational concepts in organizational science and other social sciences. Reconceptualized, the truth of computational practice is much closer to complex systems engineering.

Taking this model of computation as a model for architectural design is only a matter of time. All operating systems of significance work like this, even those inside modest desktop PCs. The Internet and the Web work this way. We find here a general conceptual framework and a living, all embracing, omnipresent, ready 24 hours a day environment. All arrangements of users (clients), designers, consultants, hardware and software are supported here and all modes of their interaction are enabled. Everything and everybody is just an object in a community of interacting objects. Cybernetics as the theory of/for design (Glanville 1997) can be put here into practice. The fundamental concepts of cybernetics are supported in a natural way. There is feedback between objects-actors enabling them correction of their state and behavior and evolution by trial and error towards a common goal. Control is distributed between the actors and works in two directions between any pair of objects. Humans (designers, users) can work personally, on-line with each other and with software objects, or can be represented in a computation through tireless software agents. Actually, humans are, at one extreme, too slow to control a computation in real-time. At the other extreme, they are to impatient to wait for the result of a lengthy computation, the more so that it may be not at all clear what kind of result to expect. Furthermore, software agents can be anyplace, anytime, and there can be a host of them. They can check each and every aspect of an evolving design, real-time. They can have and use all senses (touch, smell, hear, feel the temperature, not just see), unlike a human designer (detached classical observer) observing a walk- or even fly-through. Thus software agents acting on behalf of human actors seem a viable alternative to personal involvement. The intellectual effort spent on trying to define such an agent (or many of them for different aspects of design) is an invaluable added value, and occasion to learn. The variety of such systems is enormous, compared to what we have today, and so is the potential for emergent, surprising forms.

References

Coad P, and Yourdon E (1991), Object-Oriented Design, Prentice Hall, Englewood Cliffs, NJ.
Glanville R (1997), The value when cybernetics is added to CAAD, AVOCAAD First International Conference (Ed. J. Verbeke, T. Provoost et al), Hogeschool voor Wetenschap en Kunst, Brussels.
Jamsa K (1996), JAVA Now, Jamsa Press.
Klander L (1997), Hacker Proof, Jamsa Press.
Stroustrup B (1993), The C++ Programming Language, Addison-Wesley, Reading, MA.