IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 16, Issue 3, Ver. VI (May-Jun. 2014), PP 72-77
www.iosrjournals.org

A Study on User-Interface Architectures

K.Nirmala[1], A.V.Sriharsha[2]
[1] *Assistant.Professor, Dept. of IT, Sree Vidyanikethan Engineering College, Tirupati, India*
[2]*Associate.Professor, Dept. of CSE, Sree Vidyanikethan Engineering College, Tirupati, India*

Abstract: *This paper is a outcome of a detailed examination of tools, systems and notation used in software architectures. Modeling software architecture for a system is not as easy as one can understand tools, applying the appropriate tool essentially varies its efficiency and demands. In the background of Software Architecture analysis, design there is a profound influence of design and specification tools that tend to develop models for even challenging systems is a more systematic and predictable manner. This survey reveals various directions of research in software architectures.*

Keywords: *Architecture description languages, Architecture knowledge, User interface architectures.*

I. Introduction

Software engineering is a faculty that evolved out of serious research and brainstorming about program development, program methodology, programming languages leading to the design of large software that meets overall goal of a business turning into a business application. Ever since the programming languages, programming methodology, program platform, problem specific choice of programming solutions are evolved the field of software engineering had faced challenges in the suitable selection according to requirement specifications. Software Engineering as a field of study equips with various Software process models, software engineering methods, and software tools have been adopted successfully across a broad spectrum of industry applications.

Graphical user interfaces have become a familiar part of our software landscape, both as users and as developers. Looking at it from a design perspective they represent a particular set of problems in system design - problems that have led to a number of different but similar solutions. User Interface Engineering is another fold of study that collects all the user interface design methodologies scientifically and itself puts forth as a responsive, integrated, aesthetic technology.

Modeling and representation of the elements of the systems that exhibits the robustness, integrity and reliability is design. To achieve this diversification and convergence must be put into practice. *Diversification* is the acquisition of collection of alternatives, the raw material of design: components, component solutions, and knowledge, all contained in catalogs, textbooks and some mental models. The designer must choose the elements from the collection that meet the requirements defined by requirements engineering and the analysis model. *Convergence* of these qualities is based on the set of principles or heuristics that seem to be feasible to the designers' understanding of the target model.

Design engineering for a computer software changes continually as new methods, better analysis mechanisms and broader understanding of the principles are applied. Design is an iterative process through which requirements are translated into a "template" for constructing the software. The template or blueprint depicts a holistic view of software. In the due process, the subsequent refinements lead to several design representations at much lower levels of abstraction that are traced to fundamental requirements.

The basic elements of the analysis model of the software provide four design models. The data design, architectural design, component design and interface design. The ER diagrams may be used to illustrate the data relationship with entities in the domain. The architectural design model is a relationship between major structural elements i.e., design patterns and architectural styles. The interface design model describes the communication that the user can do with the components of the system exploring their functional and behavioral aspects.

The roots of evolution of architectures have been developed from transaction and transformation analysis of system design. From the erstwhile analysis methods, useful techniques for developing, assessing and evaluating the overall complexities of architectures is proposed. Using the software requirements there could be possible directions of mappings that build an architecture view of the system. Certain dependencies are identified that help assessing the complexities and the dependencies between the components within the architecture. These dependencies are driven by information/control flow within the system. Zhao.

Structured programming can be seen as a subset or sub discipline of imperative programming, one of the major programming paradigms. Several different structuring techniques or methodologies have been developed for writing structured programs. Edsger Dijkstra's structured programming, the structure composes the logic of a program as that of similar sub-structures in different ways; this improves easy comprehension of

logic and the program. Data Structured Programming or Jackson Structured Programming, which is based on aligning data structures with program structures. Fundamental structures proposed by Dijkstra are also applied as constructs that represent the high-level structure of a program that can be modeled on the underlying data structures. There are at least three major approaches to data structured program design proposed by Jean-Dominique Warnier, Michael A. Jackson, and Ken Orr.

Object Oriented programming is paradigm that uses fields and behaviors of the entity encapsulated into one unit. The programming techniques of OO may include features such as data abstraction, encapsulation, messaging, modularity, polymorphism, and inheritance.

II. Architecture

Architecture (from Latin *architectura*, from the Greek *arkhitekton*,) is both the process and product of planning, designing and construction. Architecture can be considered as a design activity, from the macro-level (urban design, landscape architecture) to the micro-level (construction details and furniture). The term "architecture" has been adopted to describe the activity of designing any kind of system, and is commonly used in describing information technology. Many vernaculars, concepts and treatises exist for architectures in general, but for software there are canonical. Software architecture is commonly defined in terms of structural elements and relationships [Bredemeyer]. Structural elements are identified and assigned responsibilities that client elements interact with through "contracted" interfaces.

The history of UI architectures traces back to 1990's, with the advent of visual programming tools. Visual programming is a platform where the programming aids are visually available and also helpful in developing applications with good user interfaces. These user interfaces are said to have designed by a group of controls. Each control is a widget of the application that represents an input and/or output operation of the application as specified in the code behind them. The visual programming tools revolutionized new methodologies in programming, RAD and JAD Rapid Application development and Joint Application development, which yield faster application development time and visually effective. The exploration of user interfaces and architectures is familiar to that of visiting a new place and watching various buildings and monuments. The first way is to know about Containers and Controls. These are termed as various names in various development tools. The better way of understanding the importance of user interfaces with respect to applications is the client/server architecture. It's a familiar architecture because it was the one encouraged by client-server development environments in the 90's - tools like Visual Basic, Delphi, and PowerBuilder. Layouts and the bindings of these containers and controls are cultivated among the user-interface designer. A user-interface designer is a programmer too, who also builds the code to regulate the behavior of the controls. This bears the design and development of interactive systems. Every interactive software system has a domain which it addresses and that its contents or functions are about. The interactive systems are with human tractable interfaces, thus leading to human interface to the device and human computer interface for devices that operate electronically like computers. The task of HCI design is very similar to this: User interface designers shape a virtual environment in which the user works, or, in the case of 3-D virtual environments, may even live. As we know, users generally identify a software system with what its user interface shows them, without caring about its inner works. The artefact that the user interface designer creates is something that its users directly interact with or even live in.

Design patterns that support user interface, are available as user interface patterns, HCI-patterns, interactive patterns. Tool support exists that uses different schools of design patterns that are adopted to the user interface. Pattern language tools exists to represent the patterns useful for user-interface design, such as CoPE, PLML, XML etc, There are several ADLs that can be used to represent the user interface as an architecture. User Interface design as a User Interface Architecture is not a misnomer as its methodology and application is relevantly huge. The user interface architecture is developed as an art and science for the user interface designer that covers adopting of user interface design principles with architecture design.

Architectural decisions are those that must be made from an overall system perspective. Essentially, these decisions identify the system's key structural elements, the externally visible properties of these elements, and their relationships (Len Bass, Paul Clements, and Rick Kazman, Software Architecture in Practice, Addison-Wesley, 1997), and they define how to achieve the architecturally significant requirements. If you can achieve the requirement by deferring the decision to a lower level, it is not architecturally significant, and the decision is not an architectural one (at the given level of scope).

III. Architectural Description Languages

Architecture description languages (ADLs) are formal languages that can be used to represent the architecture of a software-intensive system. As architecture becomes a dominating theme in large system development and acquisition, methods for unambiguously specifying an architecture will become indispensable.

A standard architecture description language can represent underlying architecture of a system and promotes communication among system components. ADL is provisional embodiment of early design decisions, and the creation of a transferable abstraction of a system. Architectures in the past were largely represented by box-and-line drawings annotated with such things as the nature of the component, properties, semantics of connections, and overall system behavior. As seen from the properties of ADLs, graphical syntax with often a textual form and a formally defined syntax and semantics extend easy representation of GUIs of Systems. ADLs permit analysis and assessment of UI architectures, for completeness, consistency, ambiguity, and performance.

Examples of ADLs include Adage, Aesop, C2, Darwin, Rapide, SADL, UniCon, Meta-H, and Wright. While all of these languages are concerned with architectural design, each provides certain distinctive capabilities: Adage supports the description of architectural frameworks for avionics navigation and guidance; Aesop supports the use of architectural styles; C2 supports the description of user interface systems using an event-based style; Darwin supports the analysis of distributed message-passing systems; Meta-H provides guidance for designers of real-time avionics control software; Rapide allows architectural designs to be simulated, and has tools for analyzing the results of those simulations; SADL provides a formal basis for architectural refinement; UniCon has a high-level compiler for architectural designs that supports a mixture of heterogeneous component and connector types; Wright supports the formal specification and analysis of interactions between architectural components.

Some of the important ADLs that support UI architecture definitions are: LePUS, Unicon, Rapide etc. Since these are formal and graphical to represent components of the systems individually and the system as a whole, these are ideal ADLs suitable for UI architectures.

IV. Architectural Knowledge

Knowledge is now recognized as the key differentiating resource among firms. The ability to integrate widely held knowledge to derive new products and services constitutes a meta-capability from which streams new and of innovative capabilities continually emerge.

In software engineering, reusing life cycle experience, processes and products for software development is often referred to as having an "*Experience Factory*". In this framework, experience is collected from software development projects, and is packaged and stored in an experience base. By packing, we mean generalizing, tailoring, and formalizing experience so that it is easy to reuse.

In the process of architecting, a quiet a lot amount of knowledge is used and also produced. A very important part in architectural knowledge concerns about the solution chosen, in terms of components and connectors, and as documented in views. From the experience of software developers, it is clear that the decisions that lead to a solution are an important ingredient of architectural knowledge as well, thus the following definition may be endorsed:

Architectural Knowledge=Architectural Design + Architectural Design Decisions

Formal graph-based architecture representations are especially suitable for automated reasoning.

Much of the difficulty in architectural design is in integrating and making explicit the knowledge of the many converging disciplines (engineering, sociology, ergonomics and psychology, to name a few), the building requirements from many viewpoints, and to model the complex system interactions.

4.1 Architectural Knowledge for User Interface Design

The bridging process of architectural knowledge for user interface design can be conceptualized as a series of transformations that begins with the gathering of user requirements and ends with the creation of a design. While all of the existing methods discussed construes it somewhat differently, as would they be expected. Each method has its relative merits and appropriate conditions of application. In most instances the designer describes that transformation in the context of a methodology used with a specific design project. While the projects, the processes, and the methods vary considerably, the common theme is the building of that bridge between User Requirements and User Interface Design.

4.2 Existing Architectural Knowledge: MVC, MVP

Over the years, design patterns are being best practiced in design and development of applications in IT and SE. Most of the times, DPs are not followed because of lack of experience or knowledge. In case of MCV/MVP, lack of knowledge of difference between them is one of the fear factors for not choosing the right pattern.

A number of frameworks are provided to implement these patterns. For example: JAVA Struts, ROR (Ruby on Rails), Microsoft Smart Client Software Factory (CAB), Microsoft Web Client Software Factory, ASP.NET MVC Framework. DPs are the blueprints but not the solution itself. They should be used as a guideline for the implementation within the problem domain/space.

4.2.1 Differences

Although the MVP model is based on MVC (UI Presentation Pattern), UI Presentation Pattern focus on separation of view with Model. In MVC model, there is separation of responsibility between three components: View - responsible for rendering UI elements; Controller - responsible for responding to view actions; Model - responsible for business behavior and state management. In MVP model, there is a separation of responsibility between four components: View - responsible for rendering UI elements; View Interface - responsible for loose coupling between view and model; Presenter - responsible for view and model interaction; Model - responsible for business behavior and state management. In MVC, the three components would interact with each other. Controller would sometime also be responsible to update view (like Front Controller Pattern). Whereas in MVP, Presenter can also interact with Controller to access Model. MVC is fairly loose coupling and MVP is comparatively high degree of loose coupling

4.2.2 Benefits

Any pattern has its pros and cons in association with the environment it is being implemented in. It is not always advisable to implement the patterns everywhere in the application. There are benefits of MVC/MVP usage:

1. Code reuse - Separation of concern principle will provide code reusability as the design will have proper domain model and business logic in its logical unit.
2. Adaptable design - A good design is close for changes and open for additions. Isolated code in presenter/controller, domain model, view and data access provide a freedom of choosing a number of views and data sources.
3. Layering - MVC/MVP forces to separate data access login for the other layers and various other patterns would be opted to implement data access layer.
4. Test driven approach - Isolated implementation allows to test each component separately. Especially in MVP pattern that uses interface for a view, it is a true test driven approach.

4.2.3 Drawbacks

1. Higher complexity
2. Extra learning curve
3. Experience and knowledge makes a difference in the right implementation
4. Not suitable for simple and small solutions

A fundamental reality of application development is that the user interface is the system to the users. What users want is for developers to build applications that meet their needs and that are easy to use. User interface (UI) prototyping is an iterative development technique in which users are actively involved in the mocking-up of the UI for a system. UI prototypes have several purposes:

- As an analysis artifact that enables you to explore the problem space with your stakeholders.
- As a design artifact that enables you to explore the solution space of your system.
- A basis from which to explore the usability of your system.
- A vehicle for you to communicate the possible UI design(s) of your system.
- A potential foundation from which to continue developing the system (if you intend to throw the prototype away and start over from scratch then you don't need to invest the time writing quality code for your prototype).

4.3 GUI Tools

As the use of graphical user interfaces (GUIs) has become increasingly widespread, the graphical user interface programming was originally both highly complicated and system dependent, a large collection of widget libraries and GUI design tools have been created to simplify this task. GUI design tools allow the user to visually create the graphical user interface for their programs, and generate code automatically which creates the user interface, which usually targets a particular widget library that provides operations to allow the user to query the status of the widgets.

Unfortunately, many of these tools restrict the user to a particular platform (e.g. Windows). Some GUI design tools implicate libraries that have been implemented across some particular platforms. While these tools allow a programmer to use the same generated source code on many different machines, they still constrain the user to a particular implementation.

4.4 Java: AWT, Swings

Abstract Window Toolkit and Swing are the UI API in Java Platform. Both of these tools are versatile in developing UI applications. Java implicates the UI widgets of native operating system. Apart from the direct

implacability of UI or graphical shell of the operating system, Java Swing API also provides PLAF, a pluggable look and feel interface where the UI of an application can be depicted as if in another operating system or to a particular UI theme.

4.5 Microsoft: MFC, Win32SDK

The Microsoft Windows graphics device interface (GDI) enables applications to use Bitmaps, Brushes, Clipping, Colors, Coordinate Spaces and Transformations, Device Contexts, Filled Shapes, Fonts and Text, Lines and Curves, Metafiles, Multiple Display Monitors, Painting and Drawing, Paths, Pens, Printing and Print Spooler, Rectangles, Regions on both the video display and the printer. Windows-based applications do not access the graphics hardware directly. Instead, GDI interacts with device drivers on behalf of applications. Microsoft Windows promotes a GDI driver, which builds the GUI widgets, canvas and the desktop for UI development. The programming APIs MFC and Win32SDK are rich in accessing the properties, events methods of the widgets at low level. As the API is native to the operating system, there is a provision of accessing and customizing the properties, events and methods of the widgets. Microsoft Foundation Classes an O-O API for developing O-O applications quickly. MFC encapsulates all the events, methods, properties of objects for controlling, containers and other miscellaneous widgets also. Developing a structured object oriented GUI application with MFC is easy in windows platform.

4.6 Python: Orange

Open source data visualization and analysis for novice and experts. Data mining through visual programming or Python scripting. Components for machine learning. Extensions for bioinformatics and text mining. Packed with features for data analytics. Orange offers powerful widgets that are specifically used for data mining and exploratory statistical analysis on datasets.

4.7 Eclipse:

Eclipse is a universal tool platform - an open, extensible IDE for anything, but nothing in particular. The real value comes from tool plug-ins that "teach" Eclipse how to work with things - Java™ files, Web content, graphics, video - almost anything you can imagine. Eclipse allows you to independently develop tools that integrate with other people's tools so seamlessly, you won't know where one tool ends and another starts. The very notion of a tool, as we know it, disappears completely.

The platform is very flexible and extensible, but this flexibility has a serious drawback. In particular, there is no way within the program to ensure user interface consistency between the registered components within the platform. This document attempts to reconcile this problem, by defining standard user interface guidelines for the creation of new components. If these guidelines are adopted within your own tools, it will lead to greater consistency with the platform and other tools, and an easier learning curve for your customers.

4.8 Ruby: Rails

Ruby (http://ruby-lang.org) is a dynamically typed programming language created by a Japanese Software Engineer called Yukihiro "Matz" Matsumoto in February 1993. Ruby on Rails (http://rubyonrails.org) is a web application framework written using the Ruby programming language. Ruby on Rails was first released to the public in July 2004. Since then, it has seen an explosive growth in popularity. Everything in ruby is object unlike Java has instances and primitives. Framework of UI development in Ruby is Rails, which is a rich collection of objects that can host a robust application skeleton. Specific to web and web-commerce applications Rails contributes secure interaction, data base operations and persistence. Rails with Ruby install a versatile framework into application development, which implicitly incarnates MVC.

4.9 Mobile: LCDUI has revolutionized several UI API for mobile devices. Symbian is the first to become popular for LCDUI and later Java Micro Edition has developed LCDUI framework for developing MIDP applications for Micro Devices. GNUs recommendations have proceeded to the development of FOSS Linux/Unix flavor of operating systems and further into Androids. Androids are now with several flavors injected into micro devices, operating system that is versatile with UI and System, most akin to a file system, network portability, multi user and multi tasking system.

V. Conclusions and Future Work

Technologies contribute different kinds of interfaces, supported by interface design and development tools that endeavor to fix the difficulty with respect to application design, input and output feasibilities, platform limitations and specializations, user centric and cost effective. Though all the above qualities are considered into the ground of developing interfaces, the design of interface is not voted to the qualitative satisfaction of the application. The life of the application relies much on the interface with the user's perspective, rather how many

crucial functionalities are deployed with in it. With the designers perspective the interface design is stuffed with many technical, quantitative metrics that heads the interface into a sophisticated, reliable and robust. In this paper, some of the tools and platforms that are related to interface development are presented. A comparison of these will be one of the good survey that user interface designer can enlighten about the optimization, robust, user-pleasing designs.

References

[1]. Roger Pressman, "Software Engineering: A Practitioner Approach", (C) Tata Mc Graw Hill, 2010.
[2]. Ian Somerville, "Software Engineering", (C) Pearson Education, 2007
[3]. Mary Shaw, David Garlan, "Software Architecture: Perspectives on Emerging Discipline", (C) Prentice Hall, 1996.
[4]. Lenn Bass, Paul Clements, Rick Kazman, "Software Architecture in Practice", Carnegie Melon Software Engineering Institute, (C) Pearson, 2003.
[5]. Wilbert O Galitz, "The Essential Guide to User Interface Design", (C) John Wiley & Sons, 2007.
[6]. Shneiderman Ben, "Designing the User Interface", Addison-Wesley Longman, Limited, © 1997.