The South African Institute for Computer Scientists and
Information Technologists

ANNUAL RESEARCH AND DEVELOPMENT
SYMPOSIUM

23-24 NOVEMBER 1998
CAPE TOWN
Van Riebeeck Hotel in Gordons Bay

Hosted by the University of Cape Town in association with the CSSA,
Forchheistim University for CHE and
The University of Natal

PROCEEDINGS

EDITED BY
D. PETKOV AND L. VENTER

SPONSORED BY

ABSA Group
The South African Institute for Computer Scientists and Information Technologists

ANNUAL RESEARCH AND DEVELOPMENT SYMPOSIUM

23-24 NOVEMBER 1998
CAPE TOWN
Van Riebeeck hotel in Gordons Bay

Hosted by the University of Cape Town in association with the CSSA, Potchefstroom University for CHE and The University of Natal

GENERAL CHAIR: PROF G. HATTINGH, PU CHE

PROGRAMME CO-CHAIRS:
PROF. L VENTER, PU CHE (Vaal Triangle), PROF. D. PETKOV, UN-PMB

LOCAL ORGANISING CHAIR: PROF. P. LICKER, UCT - IS

PROCEEDINGS

EDITED BY
D. PETKOV AND L. VENTER

SYMPOSIUM THEME:
Development of a quality academic CS/IS infrastructure in South Africa

SPONSORED BY
Copyrights reside with the original authors who may be contacted directly.

Edited by Prof. D. Petkov and Prof. L. Venter
Van Reebek Hotel, Gordons Bay, 23-24 November 1998


Keywords: Computer Science, Information Systems, Software Engineering.

The views expressed in this book are those of the individual authors and not of the South African Institute for Computer Scientists and Information Technologists.

Office of SAICSIT: Prof. J.M.Hatting, Department of Computer Science and information Systems, Potchefstroom University for CHE, Private Bag X6001, Potchefstroom, 2520, RSA.

Produced by the Library Copy Centre, University of Natal, Pietermaritzburg.
FOREWORD

The South African Institute for Computer Scientists and Information Technologists (SAICSIT) promotes the cooperation of academics and industry in the area of research and development in Computer Science, Information Systems and Technology and Software Engineering. The culmination of its activities throughout the year is the annual research symposium. This book is a collection of papers presented at the 1998 such event taking place on the 23rd and 24th of November in Gordons Bay, Cape Town. The Conference is hosted by the Department of Information Systems, University of Cape Town in cooperation with the Department of Computer Science, Potchefstroom University for CHE and and Department of Computer Science and Information Systems of the University of Natal, Pietermaritzburg.

There are a total of 46 papers. The speakers represent practitioners and academics from all the major Universities and Technikons in the country. The number of industry based authors has increased compared to previous years.

We would like to express our gratitude to the referees and the paper contributors for their hard work on the papers included in this volume. The Organising and Programme Committees would like to thank the keynote speaker, Prof M.C Jackson, Dean, University of Lincolnshire and Humberside, United Kingdom, President of the International Federation for Systems Research as well as the Computer Society of South Africa and The University of Cape Town for the cooperation as well as the management and staff of the Potchefstroom University for CHE and the University of Natal for their support and for making this event a success.

Giel Hattingh, Paul Licker, Lucas Venter and Don Petkov
Table of Contents

Lynette Drevin: Activities of IFIP wg 11.8 (computer security education) & IT related ethics education in Southern Africa  
Page 1

Reinhardt A. Botha and Jan H.P. ElofT: exA Security Interpretation of the Workflow Reference Model  
Page 3

Willem Krige and Rossouw von Solms: Effective information security monitoring using data logs  
Page 9

Page 12

Carl Papenfus and Reinhardt A. Botha: A shell-based approach to information security  
Page 15

Walter Smuts: A 6-Dimensional Security Classification for Information  
Page 20

Philip Machanick and Pierre Salverda: Implications of emerging DRAM technologies for the RAM page Memory hierarchy  
Page 27

Susan Brown: Practical Experience in Running a Virtual Class to Facilitate On-Campus Under Graduate Teaching  
Page 41

H.D. Masethe, T.A Dandadzi: Quality Academic Development of CS/IS Infrastructure in South Africa  
Page 49

Philip Machanick: The Skills Hierarchy and Curriculum  
Page 54

Theda Thomas: Handling diversity in Information Systems and Computer Science Students: A social Constructivist Perspective  
Page 63

Udo Averweg and G J Erwin: Critical success factors for implementation of Decision support systems  
Page 70

Magda Huisman: A conceptual model for the adoption and use of case technology  
Page 78

Paul S. Licker: A Framework for Information Systems and National Development Research  
Page 79

K. Niki Kunene and Don Petkov: On problem structuring in an Electronic Brainstorming (EBS) environment  
Page 89
Derek Smith: Characteristics of high-performing Information Systems Project Managers and Project Teams

Lucas Venêr: INST AP: Experiences in building a multimedia application

Scott Hazelhurst, Anton Fatti, and Andrew Henwood: Binary Decision Diagram Representations of Firewall and Router Access Lists

Andre Joubert and Annelie Jordaan: Hardware System interfacing with Delphi 3 to achieve quality academic integration between the fields of Computer Systems and Software Engineering

Borislav Roussev: Experience with Java in an Advanced Operating Systems Module

Conrad Mueller: A Static Programming Paradigm

Sipho Langa: Management Aspects of Client/Server Computing

T Nepal and T Andrew: An Integrated Research Programme in AI applied to Telecommunications at ML Sultan Technikon

Yuri Velinov: Electronic lectures for the mathematical subjects in Computer Science

Philip Machanick: Disk delay lines

D Petkov and O Petkova: One way to make better decisions related to IT Outsourcing

Jay van Zyl: Quality Learning, Learning Quality

Matthew O Adigun: A Case for Reuse Technology as a CS/IS Training Infrastructure

Andy Bytheway and Grant Hearn: Academic CS/IS Infrastructure in South Africa: An exploratory stakeholder perspective

Chantel van Niekerk: The Academic Institution and Software Vendor Partnership

Christopher Chalmers: Quality aspects of the development of a rule-based architecture

Rudi Harmse: Managing large programming classes using computer mediated communication and cognitive modelling techniques
Michael Muller: How to gain Quality when developing a Repository Driven User Interface 184

Elsabe Cloete and Lucas Venter: Reducing Fractal Encoding Complexities 193

Jean Bilbrough and Ian Sanders: Partial Edge Visibility in Linear Time 200

Philip Machanick: Design of a scalable Video on Demand architecture 211

Freddie Janssen: Quality considerations of Real Time access to Multidimensional Matrices 218

Machiel Kruger and Giel Hattingh: A Partitioning Scheme for Solving the Exact k-item 0-1 Knapsack Problem 229

Ian Sanders: Non-orthogonal Ray Guarding 230

Fanie Terblanche and Giel Hattingh: Response surface analysis as a technique for the visualization of linear models and data 236

Olga Petkova and Dewald Roode: A pluralist systemic framework for the evaluation of factors affecting software development productivity 243

Peter Warren and Marcel Viljoen: Design patterns for user interfaces 252

Andre de Waal and Giel Hattingh: Refuting conjectures in first order theories 261

Edna Randiki: Error analysis in Selected Medical Devices and Information Systems 262
HOW TO GAIN QUALITY WHEN DEVELOPING A REPOSITORY DRIVEN USER INTERFACE

Michael C. Muller
Client Products, Engineering,
Rubico, PostNet #22, Private Bag X87, Bryanston 2021, email: michael@rubico.com

1. Abstract

When I use the term Repository Driven User Interface (RDUI), it simply means that besides the normal data that users save and use in the running of a business, other data, concerning the user interface is also stored in the repository. The main challenge with these kinds of systems is that they tend to be slower than their conventional counterparts. Quality characteristics that are specifically applicable to Repository Driven User Interfaces include maintainability, portability, reliability, usability, functionality and efficiency. One gains quality when using OO and SBA to build the Repository Driven User Interface. Object-oriented programming is a faster way of developing and maintaining systems. Service Based Architecture gives greater flexibility and causes maintenance to be less. Objects are easily maintainable, which improves developer productivity. The functionality of the system should also improve, because the objects are smaller and mistakes can be picked up easily. The reusable component or object also makes it easier for developers to write client applications. They can simply pick the service that they want from the available services. Adding new services are just as easy, using Service Based Architecture.

2. Introduction

Repository Driven User Interfaces (RDUI) are not widely used, but it has been gaining popularity in recent years. The main challenge with these kinds of systems is that they tend to be slower than their conventional counterparts.

At the moment, software developers are still writing systems in a conventional way. A GUI (Graphical User Interface) tool is used to develop the user interfaces of the system. Code is written in the events of the different GUI objects that are going to be on a screen or window. When an event is triggered, it may cause data to be retrieved from a database or a batch process to be executed.

In a conventional development cycle, one tends to drift off from the patterns and standards that should be used. In some cases no standards or patterns exist. The developers just take a problem and implement a solution in any way they see fit. As long as it works, right?

The Repository Driven User Interface that I discuss in this paper, uses a sound architecture and object oriented concepts. The use of a good architecture and concepts such as Object Orientation, forces the developers to follow a certain pattern when developing systems. They are also forced to follow certain standards, which prevents them from using the system in any other way.

There are not many developers who write Repository Driven User Interfaces. The main reason for this is a lack of knowledge of implementation.
3. The Repository Driven User Interface (RDUI) concept

3.1 What is a repository?
A repository is any place that data or information can be stored and retrieved. This can be a database, text file, excel spreadsheet etc.

3.2 What is a Repository Driven User Interface?
When I use the term Repository Driven User Interface (RDUI), it simply means that besides the normal data that users save and use in the running of a business, other data, concerning the user interface is also stored in the repository.

To some, this may seem like overkill, but just think of how easy it would be to change user interface requirements.

When a user suddenly wants to change the position of a button or any other object, or add an object or two to a form or window, there will be no need to change or recompile the code. This is because all the objects on a window sit in a repository. Nothing that shows up on the screen is hardcoded into the code.

If the objects are all in the repository, then the properties of the objects must also be in the repository. For example: A button normally has an X-value and a Y-value, which point to the Button's position on the window. It also has a certain width and height. Other properties such as the text on the button and it's colour, must also be set at some point in time.

All these properties and their values must be set inside the repository, when the screen is set up. Once all the objects and properties for a screen are set up in the repository, the window can be opened and the objects will appear in their positions, with all the other properties set.

Now all the objects are displayed on the screen, but something is still missing. What will happen when a button is clicked? The answer is NOTHING. This is because the events for objects have not been told what to do, when they are triggered.

The behaviour of objects on the screen, should therefore be built in, or put into the repository. The obvious way to go is to build functionality into the objects. This means that code will have to be changed, when functionality changes.

The correct way is to put the behaviour into the repository. Each of the objects that can be used on user interfaces, will have a piece of code in all their events. This piece of code will never have to be changed. The code simply interacts with the repository, to execute functionality, set up against an event for an object.

Separate objects, called services that implement the functionality of user interface objects, must be built. For example:
- A button is clicked and must populate a Listbox with values.
- The clicked event for the button will simply have code that checks if the clicked event, for that particular button, has any behaviour set up against it in the repository.
- The behaviour set up in the repository may instruct that a Populate function of a certain Listbox must be triggered. The data for the Listbox, can be a set of values, retrieved by an SQL statement.
- A Populate service must therefore exist which will populate a Listbox with values. Since this type of functionality will never change, it can be hardcoded into the code.

Many of these service objects will exist inside the code. Most of them will simply implement the functions or methods of objects like buttons, listboxes and treeviews. For example: A Button and all other object may have a SetX() and SetY() service, which can be used to set or change the position of objects on the screen.
For the whole system to work there has to be some controlling mechanism to get the information from the database and to get the objects on the screen.

These controlling mechanisms are the following:
- A Data Manager to get the object data and any other data from the repository;
- An Object Manager to control the objects on the screen and to receive messages from the objects;
- A Presentation Manager to handle requests from the object manager and the data manager and instruct the object manager to execute services or methods, depending on certain conditions set in the data received, from the data manager.

The responsibility of the data manager is to:
- get user and version information from the repository;
- get the objects for a screen from the repository;
- get the properties for each object from the repository;
- get the events that have been configured for each object from the repository;
- get data for objects that need data from the repository.

The responsibilities of the object manager is to:
- create the objects that need to appear on the screen;
- listen to requests from the objects when an event has been triggered on the objects;
- execute services that manipulate the user interface objects.

The responsibilities of the presentation manager is to:
- Receive requests from the object manager;
- Request information from the data manager when an object requests something;

Figure 1. The Architecture of the system
• Interpret data received from the data manager to execute certain services (methods) requested by the objects.

4. Discussion of quality characteristics.

4.1 Definition of software quality[2]:
• "The totality of features and characteristics of a software product that bear on its ability to satisfy given needs; for example, conform to specifications."
• "The degree to which software possesses a desired combination of attributes."
• "The degree to which a customer or user perceives that software meets his or her composite expectations."
• "The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer."

4.2 Quality Characteristics

Quality characteristics that are specifically applicable to Repository Driven User Interfaces include maintainability, portability, reliability, usability, functionality and efficiency.

4.2.1 Maintenance
Maintaining the system, changing and testing in particular, should be easier when using a Repository Driven User Interface.

In general:
• Testing should start early in the life cycle of a product and must be done as often as possible;
• The test planning must start as soon as the requirements are complete;
• Modules should be tested as you develop them;
• A good test suite tries to prove that errors exist;
• Document your current development process, no matter how chaotic it may be.

4.2.2 Portability
The software must be transportable from one environment to another. Writing such a system, that can adapt and be used in different environments, is difficult.

The system must therefore be able to run on different platforms. It must also be able to run on these platforms as a standalone application, or inside a browser, which is the trend these days.

4.2.3 Functionality
As regards functionality, the system must always give accurate results, be able to perform the same functions under various operating systems and environments and interact with other systems on the same or any other platform.

4.2.4 Reliability
The reliability of the system is important if it is going to be used in potentially high-risk environments. Fault tolerance and level of maturity of the system must be very high.

This is especially important if the system is going to help spaceships launch or help make other life or death decisions. You don't want the system to crash when final countdown on a rocket launch has been reached.

4.2.5 Usability
The usability of the system is one characteristic where a Repository Driven User Interface can have a lot of advantages. The user must be able to understand, learn and operate the system.
The best way to make users comfortable with a new system is to give them something that is familiar to what they are used to or close to it.

This will make the process of learning the system easy and less time consuming.

4.2.6 Efficiency

When looking at efficiency of the product, the main concern is time behaviour. The objective would be to get the response time of the system to an acceptable rate, whether the system is to be used as an Internet browser application or as a standalone application.

4.3 Cost of lack of quality

Poor quality software costs a lot of money, lost opportunities and irritation.

The following has been said about quality or the result of bad quality:

- “Truth comes out of error more easily than out of confusion.” — Francis Bacon[2]
- “I know not a single less relevant reason for an update than bug fixes. The reason for updates is to present new features.” — Bill Gates in Focus, a German magazine[2]

5. Repository Driven User Interface and Quality

5.1 How would such a concept be implemented?

There are many ways to go about and develop a system that uses the repository as the basis for building application user interfaces. The objective would be to find a way that will enforce quality onto the system.

The Repository Driven User Interface described in section 2, uses Object Oriented technology and a Service Based Architecture as basis for its development. One gains quality when using OO and SBA to build the Repository Driven User Interface.

5.1.1 Object Oriented (OO) technology

Object-oriented programming is a faster way of developing and maintaining systems. The initial learning process may be longer than for traditional procedural programming techniques, but the end result is worth it. The system design is simpler and programmers experience an increase in productivity.

The way it all works is that the system will consist of a bunch of objects. These objects send messages to each other and instruct each other what to do.

In the OO world there are classes and objects.
- A class is anything that can live on its own. For example: Book, table, button, and payroll.
- Objects are classes at runtime.
The characteristics of OO used in the development of the RDUI are the following:

- **Inheritance** – This happens when a class is derived from another class. In other words, it will have all the data (attributes) and functionality (methods) of the parent class. It is however possible to override or extend those attributes and methods. See Figure 2.

- **Polymorphism** – When an object of a derived class is treated as an object of the base (parent) class. This means you can write a single piece of code in the parent class, which will be called for any of the derived classes. The process that enables polymorphism to happen is called dynamic binding. In Figure 2 the Stop() method for Car will be called when the Vehicle class object is called and the type of object is Car.

- **Encapsulation** – An object keeps all its attributes and methods that can be called for that object within itself. See Figure 3.

---

**Figure 2. Inheritance of classes**

**Figure 3. Encapsulation**
• Information hiding – The objects that interact with an object, are only allowed to see certain public methods and attributes of that object. This allows for a single entry point into an object. This keeps other objects from accessing private methods and attributes of that object directly. In Figure 4 the Tree() method gives the only access to the private attributes and method of the Tree object.

5.1.2 Service based architecture (SBA)

Service Based Architecture gives greater flexibility and causes maintenance to be less.

This type of methodology places emphasis on processing within a service object According to Green and Brown these objects should be compact, combining functionality which relates specifically to it’s purpose[3].“Services are cohesive collections of related functionality accessed through a consistent interface that encapsulates the implementation”[1].

These objects are more encapsulated and easier to maintain. There is a common access point to the object’s functionality.

Figure 5. Service Based Architecture

The benefits of using SBA according to Green and Brown[3]:
• Improved encapsulation and code portioning. The classes are more manageable and defined;
• Make reusable classes easier to use. Services need only to be “turned on”, requiring little or no work from developers. When requirements change, services can be removed or just added;
- Makes code maintainable and enhanceable. The classes are smaller and services are well defined. New services can be added without backward compatibility fears;
- Enhanced flexibility; developers can choose the service they need when developing client objects or applications;
- Improve developer performance. New services can easily be added to existing client applications.

Using this architecture enables the services to be separated from the client application. This means that objects that need more power can run on fast server machines somewhere else.

Figure 5 is a simple example of how the Service Based Architecture works. The functionality of the client application or object is controlled by the services. The developer of the client object can select any or all of the services to be part of the application.

5.1.3 Quality characteristics addressed

One of the biggest advantages of using a RDUI is the ability to modify or change the way screens look or the way that certain functions are performed. In a conventional system, the functionality and user interface is fixed and can only be changed by changing and recompiling the code.

When using a Repository Driven User Interface, the functionality of a system sits in a repository somewhere. All that needs to be done, is to make a change in the repository. The next time the system runs, the change will be done.

The best way to write an RDUI system is to follow an Object Oriented development technique, and a Service-Based Architecture. When these technologies combined, the portability of the system becomes easy.

The design of the repository plays a huge role in the success of a Repository Driven User Interface. In order to gain quality, the best design for this type of system must be used.

Object Orientation is used mainly in the repository. It is however also used in the code. In the repository, the objects are stored in such a way, that they can inherit values from each other. This makes it possible to use the same object on more than one window, with different or the same functionality. It also means that a change in the repository to one object, can impact many other objects in the system. A change will therefore be global. This means the developer can make one change in the repository and that change will automatically reflect in all the client objects it is used. Reuse is high and the ability to maintain the system is faster and simpler.

It enables the separation of the user interface (client objects) from the back-end (data objects) and service objects, which enables the use of the same system as a browser application or as a normal standalone windows application or a distributed application.

No changes in coding are necessary when data is retrieved from the repository; functionality will be the same for an Internet application as it is for a normal windows application. The separation of the user interface, data and services allow for the system to interact with other systems by simply calling services. This is made easy by having a consistent way of passing information between the objects.

Using Object Oriented techniques simplify unit testing, because the code is written in objects that each do specific tasks. There are only so many possibilities to test for when an object oriented technique is used.

This makes the code extremely reliable and thus, the system. Error handling can be done by calling a service object.

A Repository Driven User Interface can be tailored to a user’s liking, by linking the way the screens look, to his/her user id, for example. This makes the system understandable to the user. The user will learn the system a lot faster. Users will be able to operate the system without any problem.
Although all of this sounds good in theory, one of the biggest challenges is to get Repository Driven User Interface systems to conventional system's level of performance in terms of speed.

One way to do this is to take all the data that is used to build the system, and compile it into a simpler form, so that getting the data, becomes easier and faster. This may take away some of the advantages that a RDUI provides, but it doesn't have to.

The same way that conventional programs have to be recompiled, the repository data must be recompiled. The difference here is that only the values that have been changed or added, have to be recompiled and not the whole program – Incremental compilation. Also, the change is made in one place, the repository. Nothing else has to be changed.

6. Discussion and conclusion

The nature of object orientation and service based architecture enables us to write objects that have specific functionality, which makes it easier to test and the reliability of the objects become very high.

Objects are easily maintainable, which improves developer productivity. The functionality of the system should also improve, because the objects are smaller and mistakes can be picked up easily.

The reusable component or object also makes it easier for developers to write client applications. They can simply pick the service that they want from the available services. Adding new services are just as easy, using Service Based Architecture.

The Repository Driven User Interface allows the builders of the system to make the user interface friendly to individual users or groups of user of the system.

The system is also highly portable. Services can be deployed on various machines and called from the client application sitting on another machine.

The system's ability to be compiled incrementally not only gives it the efficiency needed to compare to conventional systems, but gives it the edge.

Databases are becoming faster, which means that soon, there should be no difference in runtime performance between conventional and Repository Driven systems. Repository driven system may even be faster.

7. References

