Enfoque UTE, V.10-N.1, Mar.2019, pp. 218-233
http://ingenieria.ute.edu.ec/enfoqueute/
e-ISSN: 1390-6542 / p-ISSN: 1390-9363

Recibido (Received): 2018/11/16
Aceptado (Accepted): 2019/03/06

A visual analytics architecture for the analysis and understanding of software systems

(Una arquitectura de analítica visual para el análisis y la comprensión de los sistemas de software)

Antonio González-Torres,[1,2] *José Navas-Sú,*[1] *Marco Hernández-Vásquez,*[1] *Franklin Hernández-Castro*[1] *y Jennier Solano-Cordero*[1]

Abstract

Visual analytics facilitates the creation of knowledge to interpret trends and relationships for better decision making. However, it has not being widely used for the understanding of software systems and the changing process that takes place during their development and maintenance. This occurs despite the need of project managers and developers to analyze their systems to calculate the complexity, cohesion, direct, indirect and logical coupling, detecting of clones, defects and bad odors, and the comparison of individual revisions. This research considers the design of an extensible and scalable architecture to incorporate new and existing methods to retrieve source code from different versioning systems, to carry out the analysis of programs in different languages, to perform the calculation of software metrics and to present the results using visual representations, incorporated as Eclipse and Visual Studio extensions. Consequently, the aim of this work is to design a visual analytics architecture for the analysis and understanding of systems in different languages and its main contributions are the specification of the design and requirements of such architecture, taking as base the lessons learned in Maleku (A. González-Torres et al., 2016).

Keywords

Code analysis, repository mining, software visualization, metrics.

Resumen

La analítica visual facilita la creación de conocimiento para interpretar tendencias y relaciones que permitan una mejor toma de decisiones. Sin embargo, no se ha utilizado para la comprensión de los sistemas de *software* y el proceso de cambio durante su desarrollo y mantenimiento. Esto ocurre a pesar de la necesidad de los administradores y desarrolladores de analizar sus proyectos, calcular la complejidad, la cohesión, el acoplamiento directo, indirecto y lógico, detectar clones, defectos y malos olores, y la comparación de revisiones individuales. Esta investigación considera la necesidad de una arquitectura extensible y escalable para incorporar métodos nuevos y existentes para recuperar el código fuente de diferentes sistemas de versiones, con el fin de hacer el análisis de programas escritos en diferentes lenguajes. La presentación de los resultados se realiza mediante representaciones visuales, incorporadas como extensiones de Eclipse y Visual Studio. En consecuencia, el objetivo de este trabajo es diseñar una arquitectura de analítica visual para el análisis y la comprensión de sistemas escritos en diferentes lenguajes y sus principales contribuciones son la especificación del diseño y los requisitos de dicha arquitectura, tomando como base las lecciones aprendidas en Maleku (González-Torres, García-Peñalvo, Therón-Sánchez y Colomo-Palacios, 2016).

Palabras clave

Análisis de Código, minería de repositorios, visualización de software, métricas.

1. Introducción

Visual analytics is the science of analytical reasoning, which uses advanced data analysis, interactive visualizations, human-computer interaction and the visual and cognitive abilities of human beings, to make sense of abstract information. This process allows the creation of useful

1 Costa Rica Institute of Technology, San José and Cartago, Costa Rica ({antonio.gonzalez, jnavas, marco.hernandez, franhernandez, jensolano} @tec.ac.cr).
2 ULACIT Costa Rica, San José and Cartago, Costa Rica (agonzalez@ulacit.ac.cr).

knowledge to interpret trends and relationships, which are difficult to appreciate at a glance, aiming to improve the process of decision making (Thomas & Cook, 2006). In line with this, visual analytics systems allow better decision making, which is why organizations are motivated to use it. However, visual analytics has been little used to facilitate the understanding of the dynamics of software systems and the change process that takes place during their development and maintenance.

The study of software systems looks to improve software development and maintenance through the analysis of continuous change, complexity, growth and quality control (Lehman, Ramil, Wernick, Perry, & Turski, 1997). Therefore, programming teams require tools to recover and analyze software projects to discover patterns and relationships, calculate software quality metrics (e.g., complexity, cohesion and direct, indirect and logical coupling), detect clones, defects and bad odors, and extract facts from the comparison of individual revisions (D'Ambros, Gall, Lanza, & Pinzger, 2008).

Software evolution is a result of the change record of a software system. It is a cyclic process that is based on the understanding of the current state of systems and the accumulation of previous changes (Mens & Demeyer, 2008). A change usually involves a group of source code items that are frequently together and are associated by some kind of relationship. The changes performed on any of these items can be carried out by one or several programmers, simultaneously, and are propagated automatically to other elements, coupled directly or indirectly. The comprehension of changes implies not only to appreciate the modifications to software elements, but its effects on the system structure and the relationship between the elements that compose it.

Furthermore, the evolution of systems usually expands through several years and generates thousands and even millions of lines of code (Kagdi, Collard, & Maletic, 2007), hundreds of software components and thousands of revisions (D'Ambros et al., 2008). In addition, source code is composed of variables, constants, programming structures, methods and relationships among those elements (Cárdenas & Aponte, 2017). Besides logs, the analysis of software systems also requires the retrieval of data from communication systems and the metadata records from bug tracking and SCM tools which keep records with dates, comments, changes made to the systems and details of the associated programmers (Hassan, 2005). Hence, the tools aimed to analyze software systems and their evolution should facilitate the understanding of system changes and depend upon techniques and methods, such as advanced source code analysis, software visualization and interaction techniques (Antonio González-Torres et al., 2013).

Consequently, this research is aimed to contribute with the design of a visual analytics architecture for the analysis and understanding of software systems written in different languages, using as basis the theory and the requirements of the industry and software practitioners. Therefore, the main contributions of this research are the specification of the design and requirements of such architecture, taking as base the lessons learned in Maleku (A. González-Torres et al., 2016).

This paper is an extension of the paper accepted in INCISCOS 2018 (A Gonzalez-Torres et al., 2018) and discusses the previous work carried in the proposal and implementation of similar architectures and frameworks (see section 2), explaining a general overview of the proposed architecture (see section 3), its main requirements (see section 4) and discussing the main conclusions (see section 5).

2. Previous work

SonarQube, TRICORDER and Understand are three popular systems which offer support for the analysis of source code in multiple languages, permiting the extraction of a predefined set of metrics and the possibility to define custom metrics. SonarQube and TRICORDER use a plugin model, so programmers can contribute with new or improved functionality.

SonarQube is a platform that works with Sonarlint (SonarSource, 2018) and it's available in commercial and community edition. Its architecture allows programmers to contribute with new plugins to the platform and run local real time analysis as the programmers write the source code in the supported IDEs (i.e., Eclipse, IntelliJ and Visual Studio). This system supports more than 7 versioning systems, 20 languages and the extraction of several metric types. It has a predefined set of rules that can be used to define custom metrics to detect bugs, security vulnerabilities and code smells, and measure the reliability, maintainability and security of systems.

The plugins of SonarQube can connect to SCM repositories and perform analysis tasks, whereas the server permits to configure the parameters of the analysis, process the reports and store the results into a database. The execution of these tools is triggered by a continuous integration server which call a source code analysis scanner associated to a chosen language. The results are processed, stored and sent to managers to be reviewed, and displayed to programmers.

Similarly, TRICORDER is a robust and scalable platform that is based on microservices to accomplish the static analysis of programs at Google. This tool reads a program snapshot when it becomes available, calls a language specific driver to calculate the dependencies, builds the files included in the change list to obtain the inputs required by a compiler, generates an Abstract Syntax Tree (AST) and use it to perform the analysis, which later is displayed to users (Sadowski, van Gogh, Jaspan, Söderberg, & Winter, 2015).

The main difference between Understand and the tools mentioned above is that it has a focus on providing details concerned with the architecture and structure of the systems. Thus, it carries out the analysis of dependencies for an architecture or part of it, examines the control flow of algorithms, studies the hierarchy, and checks the compliance with coding standards. Furthermore, it uses a set of visualizations for presenting the results obtained and the calculated metric values (SciTools, 2018).

Although these systems have many useful features, they lack solid mechanisms for integrating the analysis results with effective methods to facilitate decision making during coding and management tasks. Therefore, González-Torres described the process of applying visual analytics to software evolution to enhance the understanding of changes with the active participation of users by means of human-computer interaction and implemented Maleku, a proof of concept architecture (A. Gonzalez-Torres, 2015; A. González-Torres et al., 2016). Visual analytics is the combination of interactive visualizations with analysis techniques to facilitate the decision-making processes (Keim, Kohlhammer, Ellis, & Mansmann, 2010).

Maleku was designed to support both programmers and managers when correlating metrics, project structure, inheritance, interface implementation and socio-technical relationships (A. González-Torres et al., 2013; A. González-Torres et al., 2016). Such framework performs ETL operations, the automated analysis of software projects and the visual representation of the analysis results. The ETL component performs the connection and source code, project structure, source code revisions, programmer activities and logs retrieval from software repositories, and then it cleans and merges data and loads it into a data warehouse. Thereafter, the automa-

ted analysis process performs metric calculations (e.g., LOC, NOM and Cyclomatic Complexity), the detection of inheritance (parent-child and child-parent) and interface implementation relationships (implementing and implemented by), the identification of socio-technical relationships, the contributions made by individual programmers and carries out examination of the architecture and structure of the project for each revision under study, using details from the metadata and the parsed source code.

The use of visual analytics to aid the analysis of source code is relatively new, although there exists substantial research on the use of software visualization. However, tasks such as debugging, the navigation of dependencies, the detection of indirect coupling and source code clones, code refactoring, the tracking of changes and contributions and software quality metrics monitoring are carried out in the industry without the support of visualization tools.

The outcomes of the research carried out by González-Torres (Antonio Gonzalez-Torres, 2015) discusses, based on an interview carried out during a usability study and on the results of survey, that some reasons that may have an adverse effect on the adoption of visualization tools are visual stress, inadequate design, the complexity of the visualizations, the time needed to learn how to use them; the requirement of prior knowledge and experience of visual tools, as well as aspects related to the lack of clarity and ambiguity of the designs.

Therefore, the same research points out that a possible reason for this situation is that most programmers are not aware of the availability of visualization tools and the options that these systems have. Furthermore, there is no substantial evidence about the diffusion and transference of the results obtained by the investigations to industry, concerning the application of information visualization to software systems and their evolution. Hence, the use of visual tools for assisting programming and management tasks needs to be sponsored by key players in the software industry (e.g., Microsoft, IBM and Borland), incorporating complete toolsets into their IDEs, SCM and bug tracking tools, and by creating training courses and technical documentation that takes them into account as central elements.

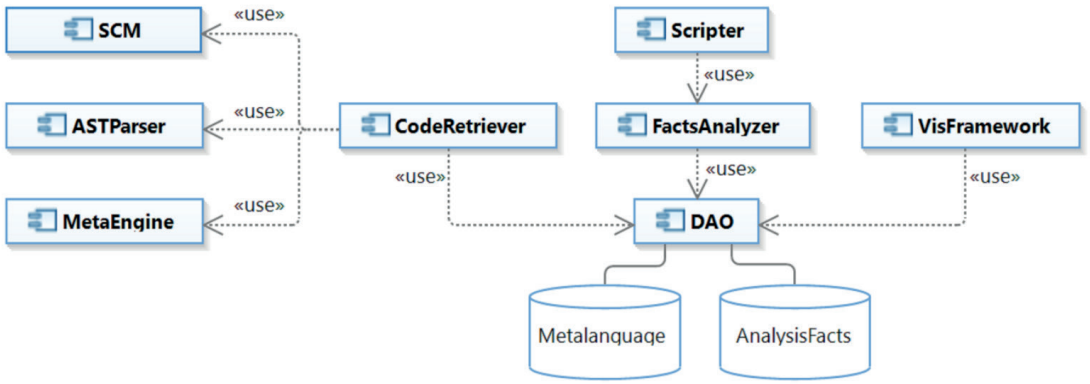
3. Architecture perspective

The design of the proposed architecture considers that several source code versioning tools, multiple languages and programming environments are used in practice. So, the design of the architecture has considered the implementation of services to connect and retrieve source code from Git, Subversion and TFVC repositories, and for the analysis of source code written in RPG, Java, C# and Visual Basic, using Eclipse and Visual Studio. The specification of the architecture is based on several micro services (see *Figure 1*) and consists of three frameworks named *CodeRetriever*, *FactsAnalyzer* and *VisFramework*, which functions are the following:

- CodeRetriever: This component connects to software repositories and retrieves the source code, written in any supported languages, and converts it into a standardized metalanguage. It is a common practice that the development of systems is carried out using different languages, according to their specific needs and the features that are offered by each language. This component requires two sets of connection parameters (URL connection, type of server, credentials). The first set will permit the connection to software repositories, whereas the second one would open a database connection to send the output of *CodeRetriever*.

- FactsAnalyzer: It carries out the static analysis of the metalanguage, calculates metrics and permits the specification of new metrics using a simple scripting language. This component would require a set of connection parameters to send the results to a database server, in a similar manner than *CodeRetriever*.
- VisFramework: This module uses multiple linked views and human-computer interaction techniques to visually represent data in an accessible way to be understood by humans in a short period of time. This component would make possible to decode data and transform it into knowledge.

Figure 1. Architecture to support the evolutionary visual software analytics process



The results of CodeRetriever could serve as the input of FactsAnalyzer or used independently. Therefore, academics and practitioners could enter their connection parameters into a web form to retrieve source code from online software repositories and store the metadata and analysis results produced by either CodeRetriever or FactsAnalyzer on their own database servers. In addition, Scripter (an interactive console) would enable users to write custom metrics based on default metrics included in FactsAnalyzer. The process that will be followed by CodeRetriever and FactsAnalyzer is described below:

- The programmer specifies the connection parameters for a software repository and a database server.
- CodeRetriever executes the SCM component to connect to repositories and retrieve source code in a per revision basis.
- CodeRetriever calls ASTParser with the source code retrieved by SCM.
- ASTParser performs the parsing of source code and creates the AST for the corresponding language.
- MetaEngine process the ASTParser output, generates the metalanguage, store it into the Metalanguage database and sends it to the FactsAnalyzer.
- FactsAnalyzer performs the calculation of the basic metrics, carries out the analysis of source code and store the results in the AnalysisFacts database.

The third major component is VisFramework, which would be responsible of loading the analysis facts from the AnalysisFacts database, create the visual representations and display results using multiple linked views. This element will be integrated into Eclipse and Visual Studio as

an extension, so when a visual element is selected from any visualization the corresponding source code will be displayed, and when the code is modified the views will be updated automatically.

Maleku was implemented in Java and a portion of its source code will be reused in this project, although it will also make use of C# and Python to comply with the requirements of the partnering companies. The analysis and visualization methods of the research will be validated by programmers and project managers with the support of their companies, which will promote the dissemination of methods and tools, both internally and externally.

4. Requirements and design specifications

This section provides details of the major functional requirements of the proposed architecture (see *Figure 1*). The first component of the architecture is CodeRetriever, which is made-up of the SCM, ASTParser and MetaEngine elements. This component is responsible for retrieving source code in different languages and from distinct repositories,

RQ1: The system shall access multiple software repositories, either local or remote, managed by different kinds of version control system to extract source code.

carrying out the transformation of code into a metalanguage and its subsequent analysis to provide insight into the system under analysis. Accordingly, **RQ1** is the main requirement that needs to be satisfied by the SCM component.

The strategy followed for the definition of SCM, as well as the one used for the specification of other components, consists on the use of a combination of the Factory Method and Singleton patterns. This would make the architecture scalable through the addition of more elements to provide access to different types of software repositories. This component is critical for retrieving source code and metadata details, such as revision numbers, the date and time of commits, the list of files modified, the names of programmers and the paths changed.

The requirement **RQ2** requires to call ASTParser with the source code retrieved from the SCM component and then pass the results to the MetaEngine element. Hence, the extraction of metrics, and the detection of clones, dependencies and coupling, for example, requires implementing a specific analysis engine for each language.

RQ2: The system shall provide a metalanguage equivalent to the abstraction of the syntactic elements from distinct programming languages to reduce the complexity of source code analysis.

The implementation of an analysis engine involves the dissection of source code using hand coded or generated parsers, or creating an AST to parse the code. Consequently, the design of a metalanguage was considered as an alternative to transform source code and perform the analysis using only one engine. The syntax of the metalanguage would be define taking into account the Knowledge Discovery Metamodel (KDM) (OMG, 2016; Pérez-Castillo, de Guzmán, & Piattini, 2011) and Abstract Syntax Tree Metamodel (ASTM) (OMG, 2011) standards from the Object Management Group (OMG).

ASTParser is also based on a combination the Factory Method and Singleton patterns and looks to support the analysis of Java, C#, Visual Basic and RPG to comply with the requirements

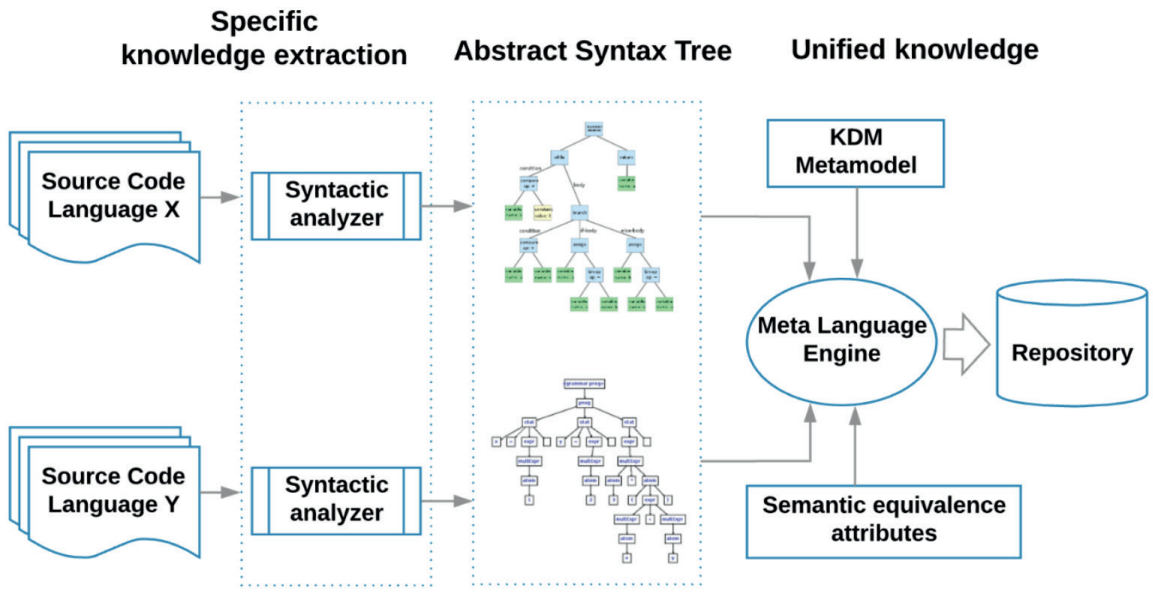
specified by the partnering companies. This component will be implemented in Java and C#, whereas Eclipse JDT and Roslyn will be used to generate the corresponding AST. The creation of the AST for RPG will be carried out in Java, having in consideration that Eclipse is the IDE employed to program in this language. The integration of the elements written in C# and Java will be performed using an adapter. The functions of MetaAnalyzer are shown in *Figure 2* and follow the next sequence:

1. Read the source code.
2. Perform the syntactic analysis of the code.
3. Generate the Abstract Syntax Trees.
4. Carry out the semantic equivalence of attributes.
5. Create the corresponding output to store it into a database.
6. Feed the output of the MetaAnalyzer into FactsAnalyzer for its analysis.

The FactsAnalyzer component is associated to requirement **RQ3** and is made up of several analysis techniques for the calculation of basic metrics and the detection of code clones, direct and indirect coupling and code item dependencies.

RQ3: The architecture shall provide an analysis framework capable of performing the characterization of the architecture, structure, changes and dependencies, the calculation of metrics and the detection of clones and coupling with the aim of simplifying their comprehension and identification of design flaws.

Figure 2. Source code transformation from different languages into a metalanguage



An integral element of FactsAnalyzer is Scripter, which will allow defining custom metrics based on basic measurements. The basic metrics included by FactsAnalyzer are Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Response for a Class (RFC), Lack of Cohesion in Methods (LCOM) (Chidamber & Kemerer, 1994), Cycloma-

tic Complexity Number (McCabe, 1976), Number of Methods, Access to Foreign Data, Number of Classes (Lanza & Marinescu, 2006), cohesion and polymorphism (Tahir & MacDonell, 2012). These metrics help on the understanding of the complexity and quality of systems and are included by most source code analyzer tools. However, most of these type of tools do not include the detection of code clones, Direct and Indirect Coupling between Object Classes (CBO) (Yang, 2010) and the network of dependencies of the system.

Code clones are source code fragments with some degree of similarity to other fragments. These can be produced by copy and paste actions, limitations of programming languages, deliberate code duplication, automatic code generation, portability compliance, or accidental coding (Murakami, 2013; Murakami, Hotta, Higo, Igaki, & Kusumoto, 2012). Clones can make software development, maintenance and refactoring tasks difficult and expensive, and can be classified into four types (Solanki & Kumari, 2016):

- **Type-1:** Identical or almost identical copies.
- **Type-2:** Syntactically parameterized copies.
- **Type-3:** Near-miss are syntactically rearranged copies.
- **Type-4:** Semantic copies.

The automatic detection of clones includes text, token, AST, Program Dependency Graphs (PDG), metrics, index and cluster based techniques, as well as hybrids and non-categorized methods (Schwarz, 2014). However, the use of any approach in large software systems is a resource-consuming task that requires an efficient and scalable method. Therefore, it is required to define novel methods to detect code clones, which could be based on Hadoop and the use of the MapReduce pipeline (Vogt, Nierstrasz, & Schwarz, 2014) to execute algorithms using parallel computation.

Coupling captures quality attributes such as complexity, maintainability, and understandability and a low level of coupling more desirable than a high one. Direct coupling between entities means the existence of a direct dependency relationship between them. Coupling relationships that are not direct are tagged as indirect coupling, and correspond to either transitive closure of direct coupling chains or to use-def chains (Yang, 2010).

Use-def chains are sequences of reaching definitions related to local variable definitions, return values, field references or parameter passing. Indirect coupling detection could rely on data-flow analysis, program slicing, ASTs, and PDGs (Yang, Tempero, & Berrigan, 2005). Several levels of granularity could be used to measure coupling, including package, class, and method levels (Almugrin, Albattah, & Melton, 2016; Almugrin & Melton, 2015).

Scripter is a component that will be used to define metrics taking the set of fundamental metrics as base. This element will be responsible of providing a mechanism to create new analysis methods, in a dynamic manner on FactsAnalyzer, without carrying out the modification of the source code (see requirement **RQ4**). The elements that conform Scripter are:

1. A simple scripting language.
2. A code generation routine that writes new code into FactsAnalyzer.

RQ4: The architecture shall provide a simple scripting language and a console for the creation of new metrics based in existing metrics.

The architecture requires producing intermediate outputs such as the metalanguage generated from source code and the results of the analysis, and therefore, the requirements **RQ5** and **RQ6** must be satisfied. Requirement **RQ5** states that the metalanguage should be produced using the metadata and source code of a given software repository, which string connection should be provided by the user to retrieve them, as well as the connection parameters for the database in which the resulting metalanguage will be stored.

RQ5: The architecture shall transform source code into a metalanguage using as source a given software repository and store the result into the database specified by the user.

Requirement **RQ6** refers to the analysis of the metalanguage to generate facts and store them in the corresponding database, based on the parameters provided by the user for such purpose.

RQ6: The architecture shall analyze a metalanguage and store the analysis facts into database specified by the user.

The results of the analysis of software systems provide useful information, but it does not provide sufficient information to carry out the tasks of understanding changes in a satisfactory fashion. Therefore, visual analytics may offer solutions to the problem of supporting programmers and managers during software development and maintenance, because it is a process which offers a comprehensive approach for the visual representation of the analysis results.

Information visualization frequently is referred as visual analytics, however, there exist several differences between both. Visual analytics makes intensive use of data analysis, coordinated and multiple views and combines the advantages of machines with human strengths, such as analysis, intuition, problem solving and visual perception (A. González-Torres et al., 2016). Therefore, it offers the potential to explore different levels of detail using multiple visual representations, coordinated together and supported by the use of interaction techniques (North & Shneiderman, 2000).

Visual analytics facilitates the discovery of relationships and knowledge by means of the analytic reasoning of the analyst. However, the application of visual analytics to the analysis of software systems and their evolution is new, and it has become a good option to support software development and maintenance because the tasks performed by programmers and project managers, and their information needs, are complex. Therefore, the needs of these individuals require the design and implementation of solutions with specific characteristics. In general, visual analytics tools must:

- Allow analysts to understand the massive and constant growing data collections.
- Support multiple levels of data and information abstraction.
- Allow the analysis of temporal data.
- Aid the understanding of unclear, confusing and incomplete information.

However, the documentation on the design of architectures for visual analytic applications is scarce, although there are several works that describe the use of design patterns on the implementation of visualization libraries. Overall, design patterns play an important role in software development, because they allow the use of known and effective solutions to solve

certain problems. Hence, the proposal of a catalog of design patterns to implement visualizations (Chen, 2004; Heer & Agrawala, 2006) and the rapid development of prototypes (Giereth & Ertl, 2008) have represented an important effort.

The design of visual analytic applications requires the programming of two or more interactive and configurable visualizations, which can be supported by animations. The use of several views is intended to provide different information perspectives to assist in the discovery of relationships. Multi-view systems are usually based on a three-dimensional model, which considers the selection, presentation and interaction between the views (Wang Baldonado, Woodruff, & Kuchinsky, 2000). The detail of each of these dimensions is presented below:

- **Selection of views:** It is the first phase in the design process and involves the identification of a set of views to be used, in a coordinated way, to support a task.
- **Presentation of views:** Once the views have been selected, it must be decided how they will be presented, that is, sequentially (for example, the user can use a menu to switch between different views) or simultaneously.
- **Interaction between views:** Each view can be accessed independently, using a selection or navigation interaction. Often, these views are linked to the actions that are performed in one view and have an effect in another view. A common interaction technique is the master-slave relationship, in which actions in one view produce effects in others. Another interaction technique is linking, by means of which the data of one view relates to those of another view. A specific linking type is brushing, in which the user highlights the elements in one view and the system highlights the corresponding elements in another view.

The development of this type of systems is complex and constitutes a challenge that demands to make decisions about the design and implementation of sophisticated coordination mechanisms. Hence, there is a need for tools to support the design and implementation of scalable and flexible visual analytic facilitating the following aspects:

- The design and implementation of visualizations.
- The linking of visualizations to data.
- The connection of the visualizations to each other.
- The development of the source code associated with the events that are fired when an action is performed in one of the representations.
- The representation of data in the corresponding visualizations in response to an event.

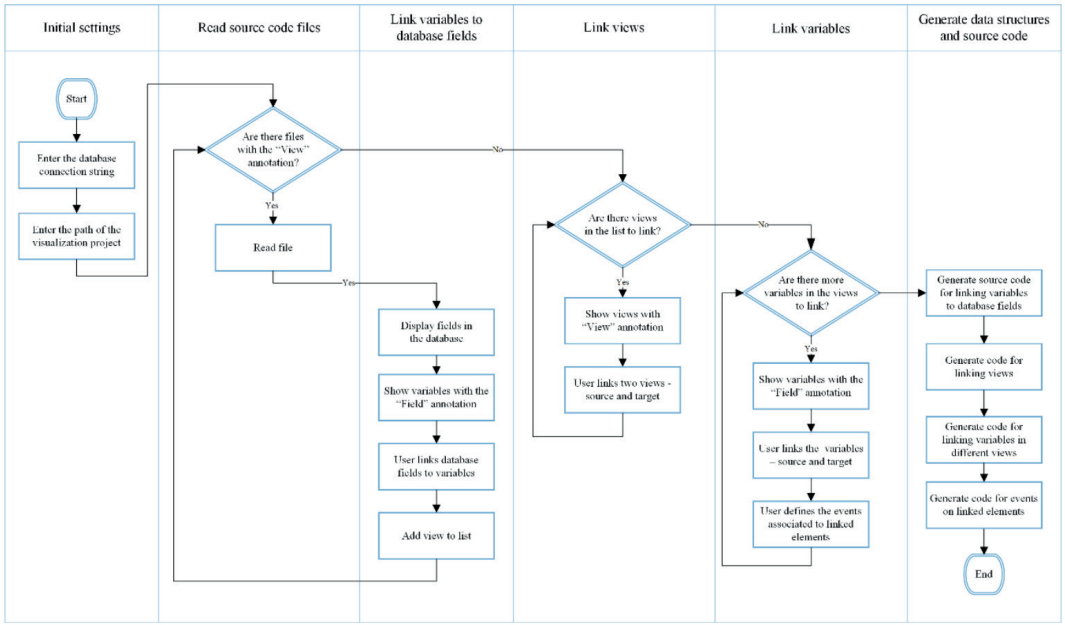
Therefore, requirement **RQ7** is based on the need of designing and implementing a scalable visual analytics architecture to support project managers and programmers during software maintenance and development.

RQ7: The architecture shall provide a scalable and extensible visual analytics architecture to facilitate software development and maintenance upon the results of source code analysis and the calculated metrics.

Consequently, this research work proposes a tool design to facilitate the creation of visual analytics applications, by means of an assistant to link the visualizations with the data in a dynamic way and link the visualizations to each other using the master-slave technique. Its definition will consider the actions listed below, which are shown in *Figure 3* with some additional detail:

1. Enter the connection string of the database to be used.
2. Create minimal visualization structures that contain the definition of variables. The visualizations should be annotated with the "View" tag and the variables with the "Field" label.
3. Link the internal variables of each visualization to the appropriate database fields.
4. Develop the data structures to conform the skeletons of the internal structures of each visual representation.
5. Map the elements of the data structures to the appropriate visual objects on each visualization.
6. Carry out the programming of the visualization layouts.
7. Apply the visualization layouts to arrange the visual objects in each visualization in a proper manner, according to the corresponding design
8. Create the relationship between visualizations to update a view when an event is triggered in a linked view.
9. Define events for visual objects that may trigger actions local to the visualization or in other visualizations, according to the relationship between views.
10. Generate source code.

Figure 3. Steps to aid the linking of variables to database fields, the creation of relationships between views and the association of variables in different visualizations



The development of the abstract data structures, the mapping of these to visual objects and the application of the layout to visual objects are tasks that needs to be performed in an individual basis for each view, when a new visualization is designed. Besides, the programming of the visualization layouts can be carried out only once and then, these can be applied several times to many visualizations. Hence, the design of a scalable and extensible visual analytics architecture should offer support to reusability and include at least two independent component libraries: one for the visualizations and other for the visualization layouts. So, the architecture

can offer the possibility to add new visualizations and layouts to the libraries, and the functionality to create a visual analytics application by choosing a combination of existing visualizations and apply the most appropriate layout to each visual representation.

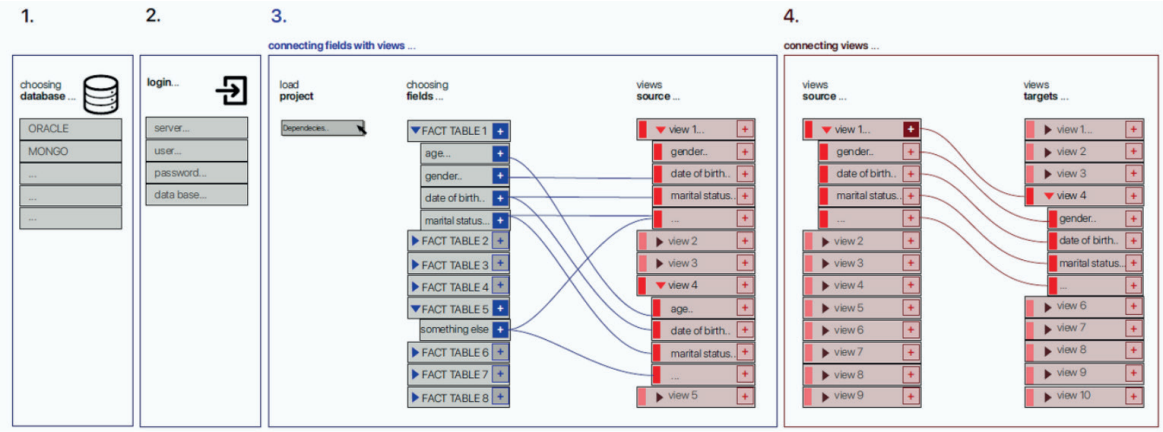
Some tasks such as the linking of database fields, the creation of the relationship between visualizations and the definition of events are repetitive works that can be carried out with the aid of an assistant tool, such as a wizard, available in the sidebar of the IDE as a plugin or extension. It is important to highlight that this proposal is aimed to aid programmers creating visual analytics applications and hence, the input of the tool shall be the source code under development and the output that it is going to produce is the modification of such code with the creation of new methods and statements to carry out the desire functionality.

Figure 4 shows the interface design with the sequence of the steps to link the views to the data, link the views with each other and generate the corresponding source code. The proposal is based on the need of implementing visualizations that are independent of the data and the creation of dependencies between visualizations dynamically, to show the information using a context design + detail (Shneiderman, 1996). The steps illustrated by Figure 4 are the following:

- **Steps 1 and 2:** These steps are used to enter the initial settings, which consist on entering the connection string of the database to which the visualizations will be associated.
- **Step 3:** The path of the visual analytics project is entered, and its source code files, annotated as a "View", are processed: Then, the variables tagged with the "Field" label are linked to the database fields.
- **Step 4:** The visualizations with the "View"' tag are linked according to the analysis flow that will be followed by the users of the visual analytics application. The flow can be unidirectional or bidirectional. Furthermore, in this step the variables that will link visualizations are match, so when the user clicks on a visual object associated to a variable in one visualization other view is affected and an action is performed, such as highlighting an element or carry out a query.

The final step involves the definition of the events to trigger actions based on the mouse or keyboard behavior, a code is generated for linking variables to database fields, visualizations between each other, variables in different visualizations and to create mouse and keyboard events to respond to user actions.

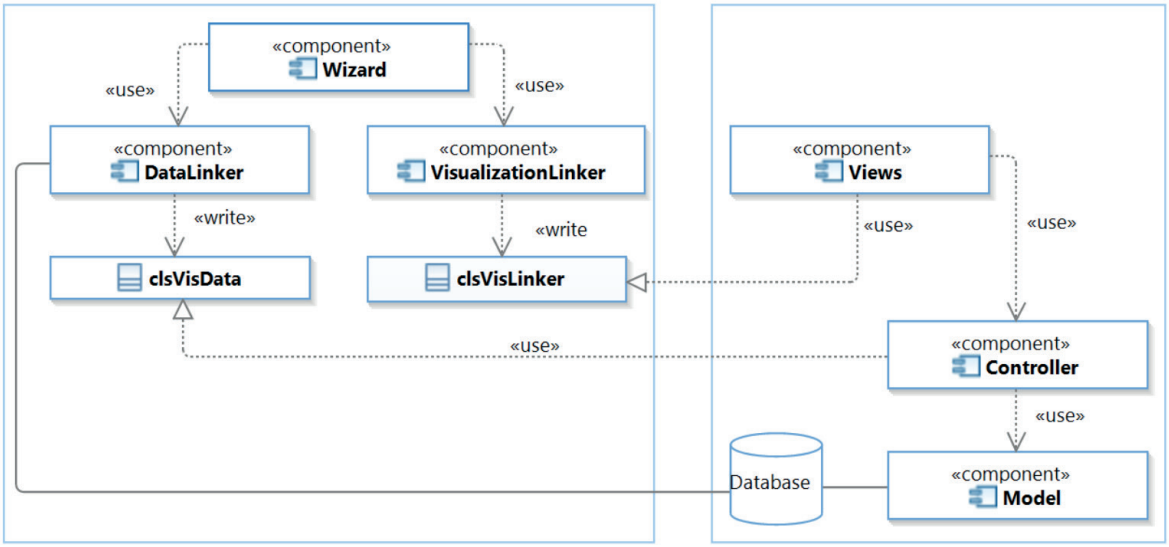
Figure 4. Interface design for the steps to link the views to data and the views with each other



The component diagram of the tool is shown in *Figure 5*, which shows the module **Wizard** based on two components that act as assistants, the **DataLinker** and the **VisualizationLinker**. These modules are described below:

- **DataLinker:** Allows to perform the database connection, read the visualizations, and provides the interface for the programmer to make the association between the fields of the database and the variables of the visualizations. The component creates the class **clsVisData** and generates the code to perform the binding.
- **VisualizationLinker:** This module facilitates the linking between the visualizations through a mapping between these and their visual elements. It reads the classes, displays a list with the variables required to link the visualizations, creates the class **clsVisLkinker** and generates the necessary code.

Figure 5. Component diagram of the visual analytics tool



VisualizationLinker is independent of the data and whose visualizations are not linked to each other but can be dynamically modified by **DataLinker** to connect the data and link the views. This component uses an architecture based on Model-View-Controller (MVC).

5. Conclusions

Developers and project managers need to understand the software they are developing and maintaining, when they have no prior knowledge or documentation of those systems. This situation acquires greater importance with the fact that software evolution is a process which usually last several years and produces data that shares many of the typical characteristics of Big Data. Thus, the capacities of programmers and project managers are particularly limited when they need to analyze large projects and are not able to extract useful information.

Therefore, the use of visual analytics is a practical alternative due to its advantages to transform large volumes of data into knowledge using a funnel approach powered using advanced and automatic analysis, visual representations and the abilities of humans to detect patterns

and make decisions. However, there is no evidence of the use of visual analytics in industry and the use of simple visual tools is limited. This research has considered such factor and that the analysis of source code is a non-trivial process that needs methods and techniques that have been proved and validated in other areas to support analytical reasoning and decision making.

The main contribution of this research is the design of an architecture to define frameworks based on the evolutionary visual software analytics process. The architecture was defined using as basis the previous research carried out for the definition of Maleku and the requirements of programmers and project managers of the partnering companies. The implementation of the architecture will be reusing a large portion of source code implemented in the previous work but requires programming several new components to satisfy the requirements. Furthermore, this research will incorporate new methods for the analysis of indirect coupling, code clones, program dependencies as well as novel visualizations to support analytical reasoning.

The role of the companies is key to validate results and to make feasible the knowledge transfer and generate more impact in society. Consequently, the outcomes of this investigation will be validated by the partnering companies and practitioners from other organizations.

References

Almugrin, S., Albattah, W., & Melton, A. (2016). Using indirect coupling metrics to predict package maintainability and testability. *Journal of Systems and Software*, *121*, 298-310. https://doi.org/10.1016/j.jss.2016.02.024

Almugrin, S., & Melton, A. (2015). Indirect Package Coupling Based on Responsibility in an Agile, Object-Oriented Environment. In *2nd International Conference on Trustworthy Systems and Their Applications, TSA 2015* (pp. 110-119). IEEE Computer Society. https://doi.org/10.1109/TSA.2015.26

Cárdenas, G. O., & Aponte, J. (2017). Evaluación de la Técnica de Visualización Basada en Grafos: Un Experimento Controlado. *Enfoque UTE*, 8(1), 201-216. https://doi.org/10.29019/enfoqueute.v8n1.134

Chen, H. (2004). Toward design patterns for dynamic analytical data visualization. In *Visualization and Data Analysis 2004* (Vol. 5295, pp. 75-87).

Chidamber, S. R., & Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, *20*(6), 476-493. https://doi.org/10.1109/32.295895

D'Ambros, M., Gall, H. C., Lanza, M., & Pinzger, M. (2008). Analyzing software repositories to understand software evolution. In *Software Evolution*.

Giereth, M., & Ertl, T. (2008). Design Patterns for Rapid Visualization Prototyping. In *2008 12th International Conference Information Visualisation* (pp. 569-574). https://doi.org/10.1109/IV.2008.36

Gonzalez-Torres, A. (2015, May). *Evolutionary Visual Software Analytics*. Department of Computer Science, University of Salamanca.

González-Torres, A., García-Peñalvo, F. J., Therón-Sánchez, R., & Colomo-Palacios, R. (2016). Knowledge discovery in software teams by means of evolutionary visual software analytics. *Science of Computer Programming*, *121*. https://doi.org/10.1016/j.scico.2015.09.005

González-Torres, A., García-Peñalvo, F. J., Therón, R., González-Torres agtorres@usal.es, A., García-Peñalvo theron@usal.es, F. J.., & Therón fgarcia@usal.es, R. (2013). Human–computer interaction in evolutionary visual software analytics. *Computers in Human Behavior*, 29(2), 486-495. https://doi.org/http://dx.doi.org/10.1016/j.chb.2012.01.013

Gonzalez-Torres, A., Navas-Su, J., Hernandez-Vasquez, M., Solano-Cordero, J., Herna, & Ndez-Castro, F. (2018). A Proposal towards the Design of an Architecture for Evolutionary Visual Software Analytics. In *2018 International Conference on Information Systems and Computer Science (INCISCOS)* (pp. 269-276). https://doi.org/10.1109/INCISCOS.2018.00046

Hassan, A. E. (2005). *Mining software repositories to assist developers and support managers*. University of Waterloo, Waterloo, Ont., Canada, Canada.

Heer, J., & Agrawala, M. (2006). Software Design Patterns for Information Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5), 853-860. https://doi.org/10.1109/TVCG.2006.178

Kagdi, H., Collard, M. L., & Maletic, J. I. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2), 77-131.

Keim, D. A., Kohlhammer, J., Ellis, G., & Mansmann, F. (2010). *Mastering the Information Age - Solving Problems with Visual Analytics*. Eurographics Association. Retrieved from https://books.google.co.cr/books?id=rKx0MQAACAAJ

Lanza, M., & Marinescu, R. (2006). *Object-Oriented Metrics in Practice*. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg. https://doi.org/10.1007/3-540-39538-5

Lehman, M. M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997). Metrics and Laws of Software Evolution-The Nineties View. In *Proceedings of the 4th International Symposium on Software Metrics* (p. 20). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=823454.823901

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

Mens, T., & Demeyer, S. (Eds.). (2008). *Software Evolution*. Springer.

Murakami, H. (2013). *Type-3 Code Clone Detection Using The Smith-Waterman Algorithm*. Osaka University.

Murakami, H., Hotta, K., Higo, Y., Igaki, H., & Kusumoto, S. (2012). Folding repeated instructions for improving token-based code clone detection. In *IEEE 12th International Working Conference on Source Code Analysis and Manipulation, SCAM 2012* (pp. 64-73). IEEE Computer Society. https://doi.org/10.1109/SCAM.2012.21

North, C., & Shneiderman, B. (2000). Snap-together visualization: can users construct and operate coordinated visualizations. *International Journal of Human-Computer Studies*, 53(5), 715-739. https://doi.org/10.1006/ijhc.2000.0418

OMG. (2011). Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM), v1.0. Retrieved from http://www.omg.org/spec/ASTM

OMG. (2016, September). Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM), v1.4. Retrieved from https://www.omg.org/spec/KDM/1.4/

Pérez-Castillo, R., de Guzmán, I. G.-R., & Piattini, M. (2011). Knowledge Discovery Metamodel-ISO/IEC 19506: A Standard to Modernize Legacy Systems. *Comput. Stand. Interfaces*, *33*(6), 519–532. https://doi.org/10.1016/j.csi.2011.02.007

Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E., & Winter, C. (2015). Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (pp. 598–608). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2818754.2818828

Schwarz, N. (2014). *Scaleable Code Clone Detection*. Bern University.

SciTools. (2018, July). Understand.

Shneiderman, B. (1996). The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages* (pp. 336–343). https://doi.org/10.1109/VL.1996.545307

Solanki, K., & Kumari, S. (2016). Comparative Study of Software Clone Detection Techniques. In *IEEE Management and Innovation Technology International Conference (MITiCON-2016)* (pp. 152-156). Bang-Saen, Thailand: IEEE Computer Society.

SonarSource. (2018, July). SonarQube Platform.

Tahir, A., & MacDonell, S. G. (2012). A systematic mapping study on dynamic metrics and software quality. In *IEEE 28th International Conference on Software Maintenance (ICSM)* (pp. 326–335). IEEE Computer Society. https://doi.org/10.1109/ICSM.2012.6405289

Thomas, J. J., & Cook, K. A. (2006). A visual analytics agenda. *IEEE Computer Graphics and Applications*, 26(1), 10-13.

Vogt, S., Nierstrasz, O., & Schwarz, N. (2014). *Clone detection that scales*. University of Bern.

Wang Baldonado, M. Q., Woodruff, A., & Kuchinsky, A. (2000). Guidelines for using multiple views in information visualization. In *Proceedings of the working conference on Advanced visual interfaces* (pp. 110-119).

Yang, H. Y. (2010). *Measuring Indirect Coupling*. University of Auckland.

Yang, H. Y., Tempero, E., & Berrigan, R. (2005). Detecting Indirect Coupling. In *Australian Software Engineering Conference (ASWEC 2005)2* (p. 10). IEEE Computer Society.