Exploration and Application of Software Development Automation Technology

Long Lin, He Yang*, Yue Wang, Cuncun Shi

Artificial Intelligence on Electric Power System State Grid Corporation Joint Laboratory (GEIRI), Global Energy Interconnection Research Institute co. Ltd, Beijing, 102209, China

*Corresponding author e-mail: yanghe@geiri.sgcc.com.cn

Abstract. Application development often faces problems such as demand changes, long cycle, and low efficiency. This paper attempts to solve such problems with automated generation technology. The article abstracts the design and development process of a typical application system; analyzes the key issues; then, after designing the overall architecture of the builder, it describes the design of interface presentation layer, control layer (business logic layer) and data model layer; finally this paper describes the challenges encountered by the builder and improvements in practice.

1. Introduction

In the development of application systems for the power industry, we found an embarrassing phenomenon, even if faced with a demand similar to the system developed in the past, the time and labor cost investment has not been greatly reduced. The main reasons are: 1) changes in demand, new and changed demand in the development process; 2) changes in personnel, limited number of experienced employees, insufficient capacity of new employees; 3) error-prone, when facing similar needs and technology, programmers sometimes copy old code into the new system without careful analysis; 4) insufficient testing, time and labor limits cause potential problems cannot be found early, just like a vicious circle.

Based on the above reasons, we designed and implemented a builder based on the idea of the compiler, allowing the builder to help complete more than 90% of the system development work, and invest limited manpower and time within the remaining 10%. This builder is not only a code generator, but also a system-level generator. By analyzing the requirements description, it can build a cross-software layer application system.

This paper describes the design and implementation of this application system builder. The overall architecture is as follows: The second part abstracts the design and development process of the typical application system; the third part analyzes the key issues; the fourth part first designs the overall architecture of the builder., then describes the design and implementation of the interface presentation layer, business logic layer and data model layer; the fifth part describes the challenges that the builder encounters and improvements in practice; we make a conclusion in chapter 6.
2. Abstraction of the application system design and development process

We believe that anything that can be abstracted can be generated in an automated way. To automate coding, the first step is to abstract the design and development process of the application system.

2.1. Abstraction of user requirements

An application system user requirement is generally composed of three parts: interface display, business requirements, and user data.

\[
\text{Requirement}:= (\text{Presentation}, \text{Business}, \text{Data})
\]  

2.2. Abstraction of architectural design

According to the MVC design pattern, the application system design is divided into three parts: Model, View and Controller. The interface is displayed through the design of the View, the design of the Controller is used to meet the business requirements, and the user data model is established through the design of the Model.

\[
\text{Design}:= (\text{View}, \text{Controller}, \text{Model})
\]

In a typical B/S system, a View consists of a series of pages, each of which contains some components. The presentation of each component can be determined by its type, location, style, and bound data.

In the design and implementation, the Controller needs to be completed by the front and back logic code. The front-end logic is responsible for human-computer interaction and interaction between controls, and the back-end logic implements reading and writing of user data. Reducing the number of accesses to the back-end under the condition of not affecting business functions can reduce the load on the application server and increase the throughput of the generated system.

The relational data model is the most common Model, but the user data model can also include network, hierarchical, and other models.

2.3. Abstraction on code implementation

Corresponding to the MVC architecture design, the application code implementation includes interface presentation code, business logic code and data model generation and manipulation code.
You can directly use front-end frameworks such as vue or react to fulfill user interface implementation. As with the implementation of business logic, the front-end logic code must contain the aggressive invocation of back-end logic. Back-end logic can be thought of as a process consisting of a series of processes.

The implementation of the relational data model is mainly the SQL database creation script and the code generated by the object relationship mapping tool.

3. KEY ISSUES

3.1. The composition of the code generator

According to the working objectives of the application system under the standard of MVC architecture, the code generator is divided into a user interface builder (View Builder), a business logic generator (CtrlBuilder), and a model builder (Model Builder). The front-end logic control code, although as part of the business logic, is executed on the user interface side and is also generated by the interface generator.

Builder:=(ViewBuilder, CtrlBuilder, ModelBuilder)
ViewBuilder => {ViewImpl, ActionImpl, Request}
CtrlBuilder => {Flow, Process, DataMapping, Response}
ModelBuilder => {SQL, ORMapping, ...} (4)

3.2. The part that must be clarified in the requirements

Automated generation of an application system is only possible with explicit requirements. Conversely, when a system cannot be generated using automation technology, it is likely to mean that the requirements are not clear enough.

The view builder needs to define the definition of each user interface, including the components, locations, styles, and bound data.

The business logic generator needs to clarify two data mappings. The first is Data Mapping, which is the mapping relationship between the front-end display data and the server-side object data. The second is OR-Mapping, which is the mapping relationship between server-side object data and data model.

Model generators require explicit data model representation. Represented by the commonly used relational data model, it is necessary to clarify the table structure design.
3.3. Flexibility and reusability
Application requirements that can be abstracted into templates or patterns are easily implemented using automated development techniques. Non-templated business logic can often only be solved through customized development.

During automated building, the generator sees business logic as a flow of processes consisting of a series of processing units, which has two advantages. First, custom development can be achieved through reflow process and rewrite process; second, code reusability can be achieved by reusing processing units or process segments. In practice, as the demand from different systems accumulates, the processing unit will become more and more abundant, and the proportion of code that can be automatically generated must be higher and higher.

4. Application system automatic generation technology design
Demand is the source of design and development. We first define the representation of the requirements and then transform those requirements into an intermediate information structure through an analyzer. Finally, the user interface generator, business logic generator, and model generator use these intermediate information to generate related code. There are three main ways to generate code. The first is a simple copy, mainly some of the toolkits that the system depends on when it runs, the second is replacement, during which we write to the system configurations according to templates, the third is object code generation.

![Automatic generation design](image)

Figure 1. Automatic generation design

This design has two advantages. One is that the generated system is hierarchical. Whether it is the user interface layer, the business logic layer or the data model layer, it can be manually modified or even replaced. Second, the generator has good scalability. Sub-generators can have different implementations based on object-oriented programming ideas. For the user interface generator, we have the Vue generator and the Angular generator. For the controller generator, we have Python generator and Java generator. For the model generator, we mainly implement the relational data model generator. But for different databases, we have PostgreSQL generator and MySQL generator.

4.1. Design, description and generation of User interface layer
According to the paradigm in Formula (2), we can think of the user interface as consisting of a series of Pages, each containing some components that can be nested.

4.1.1. User interface framework. The user interface can be abstracted into a common framework, and then each page is embedded therein, as shown in Fig.2.
In this framework, we design a tree data structure, each leaf represents a real page, non-leaf nodes represent a menu item with child nodes, and then the page structure represented by the tree structure is placed in the main navigation bar and the side navigation bar which, in theory, contains arbitrary levels of menus, but due to the limitation of the page size, actually 3 to 4 levels are most suitable.

In the operation experience, when the user clicks on the non-leaf node, the next level will be expanded. Otherwise, the main user interface in Fig.2 displays the real page represented by the leaf node.

In the permission control, you can select only a part of the first level menu to display. This function can be used for permission control. The pages seen by different roles or organizations are different.

With the design of the generic framework, we only need to describe the mapping relationship between each leaf node and the page to be generated. The generator will automatically complete the construction of the whole user interface framework when building the system.

4.1.2. Pages and components. The description of the page contains the components and their location on the page.

The list is the most common component. We extend the description based on the html table element, including the corresponding field name of each column, the display column name, whether one column is the hyperlink or not, the field name of the hyperlink address, and the business interface of obtaining the bound data, as well as the definition of the operation of the table, etc.

Forms are another common component. We extend the description based on html form elements, including: specifying the business interface to be submitted, the form's display name, the actual field name, the password field, the uneditable item, the relation between tables, and the definition of the operation of the form, etc.

Most of the operational definitions are related to the invocation of business interfaces, such as adding, deleting, editing, and querying table data. Additional operations such as uploading files and downloading files can be added as needed. Other operations are also used in actual system generation, such as pop-up and collapse of the interaction layer.

As with the interface framework, the operation and expected behavior in the application system are also standardized and generalized. We abstract the relevant javascript code template as part of the interface framework. When the builder runs, it will automatically generate the application system based on the definition and description of the page and component through replacing a small amount of key content.
4.2. Design, description and generation of business logic layer

**Figure 3.** Design of business logic layer

4.2.1. Interface section. In everyday development, each time we implement a requirement, in addition to the coding of business logic, we often need to complete the other two tasks. The first is to add the interface of the front-end interface to invoke the business logic, mainly the mapping relationship between the request field and the business object, and the second is to define the mapping relationship between the business object and the relational model. As mentioned earlier, in order for the system to automatically complete this part of the work at the time of construction, both of these pieces of information need to be clearly described. However, the content in the description is actually a mutual mapping of a large number of field names, which is boring and error-prone. To reduce the amount of work and reduce errors, we defined common default rules to help the builder build the application correctly even if the description is incomplete.

4.2.2. Business logic section. The application systems in the power grid are mostly developed around the data management of each department, and there are few long and complicated cross-department business processes. Therefore, when describing the business logic, we did not adopt the BPMN standard, but abstracted it by abstracting a few simple concepts.

- Proc, which represents an abstraction of a processing unit.
- SerialProcs, a special kind of Proc, can execute multiple Proc (processing units) in sequence.
- ParallelProcs, also a special Proc, can execute multiple included Proc (processing units) in parallel.

Obviously, the simplest business logic, only one Proc, is equivalent to SerialProcs or ParallelProcs that only contain one Proc. Complex business logic may include nesting of the three. In order to ensure that any two Proc can be connected to each other, we define a single data structure and require that each Proc’s input and output are of the same data structures.

When the system is built, the generator will extract the generated code from the nested Proc in the business logic.

4.3. Design, description and generation of the data model layer

The relational data model is the most common data model in an application system. In the definition of requirements, the description of the relationship-entity contains complete relational data model information. Mainly the definition of each table, including but not limited to the table name, field names in the table, field types, primary key, foreign key, non-empty constraint, uniqueness constraint, column
annotation, as well as the initialization data of each table, which may facilitate functional verification and problem debugging when necessary.

![DAG for DDL generation](image1)

![UAG for DML generation](image2)

**Figure 4. Data model generation**

4.3.1. *Generation of DDL (Data Definition Language).* According to the description of the data table in the requirement, the analyzer can obtain the DAG (directed acyclic graph) indicating the dependence of each table, as shown in Fig.4 (a), the nodes in the graph represent the data table, and the edges in the graph represent foreign key reference. We may build tables in the order of T2, T1, T4, T3, and T0. With this graph, the builder can generate SQL scripts for building tables based on dependencies.

4.3.2. *Generation of DML (Data Manipulation Language).* After the analyzer establishes UAG (undirected acyclic graph), when executing querying, adding, deleting, and editing operations, the data model builder can automatically analyze the correlation between data tables and automatically generate efficient DML statements. For example, during conditional query under multi-table joins, the requirement only needs to indicate the column name of the primary and secondary tables, logical operators and condition values, and the analyzer will automatically find all the tables to be joined to generate DML statements. In order to avoid the resource and time overhead caused by the large amount of data in the multi-table joins, these DML will first query the data table based on the query conditions, and then gradually join with the related table. As shown in Fig.4 (b), we first perform a conditional query operation on T2 and T4, and then perform the joining operation.

The generator also indexes the relevant columns in the generated DDL according to the fixed conditions in the requirements, further improving the operating efficiency of the system.

5. **Typical applications and improvements**

Based on the above key technologies, we developed an application system generator. The generator is written in Python and automates the building of an application by parsing the requirements description file in yaml format. The built system uses the tornado application server, the vue front-end architecture and the PostgreSQL database. Although this generator has helped us develop several power business application systems, it is difficult to automate when developing the "electric artificial intelligence public service platform." The main problem is that the requirements not only contain a large number of system command operations, but also the interaction of database information and command information. Therefore, we have extended the design of the business logic layer.

First, based on the original DBProc, we abstracted the processing unit Shellproof. ShellProc is primarily responsible for invoking system commands in business processes.

Second, we added the MemProc processing unit, which is responsible for the interaction between ShellProc information and DBProc information. As for information synchronization, we can specify
which side is preferred. As for information merging, we can specify to merge according to which fields, whether to intersect or union.

Moreover, we have improved the way the process is executed. Process reuse is further enhanced by specifying the starting and ending points of the process.

Obviously, the improvement of the generator can not only meet the immediate needs, but also meet all similar needs in the future.

Figure 5. Electric Power Artificial Intelligence Platform

6. Conclusion
In the project, the application system was previously expected to be completed in 6 person months, but after developers completed the demand analysis and used the builder, it cost them less than one person month to ensure the system is online. This proves the practicability and advancement of the technology.

Acknowledgments
This work is supported by the project (SGGR0000JSJS1800569) from State Grid Corporation of China which is ‘the Research of Electric Power Artificial Intelligence Platform for Experiment and Open Service’.

References
[1] D. Zima, "Modern Methods of Software Development," TASK Quarterly, tom 19, nr 4, 2015.
[2] Abrahamsson P, Salo O, Ronkainen J, et al. Agile software development methods [J]. Critical Sociology, 2002, 38 (38): 851 - 862.
[3] West D, Grant T, Gerush M and D’Silva D 2010 Agile Development: Mainstream Adoption Has Changed Agility, Forrester Research
[4] Fowler M and Highsmith J 2001 Software Development 9 (8) 28
[5] Rohrbeck R, Steinhoff F, Perder F. Virtual customer integration in the innovation process: Evaluation of the web platforms of multinational enterprises (MNE) [C]/ Management of Engineering & Technology, 2008. PICMET 2008. Portland International Conference on. IEEE, 2008.
[6] Gasque M, Rolando A. Argumentation-based dialogues over cooperative plans [D]. 2013.
[7] Humble J, Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and
[8] Wettinger J, Uwe Breitenbücher, Leymann F. Compensation-Based vs. Convergent Deployment Automation for Services Operated in the Cloud [M]// Service-Oriented Computing. Springer Berlin Heidelberg, 2014.