Business Generator Model Based on Domain Driven Design

To cite this article: Chen Chen et al 2019 J. Phys.: Conf. Ser. 1168 052028

View the article online for updates and enhancements.
Business Generator Model Based on Domain Driven Design

Chen Chen¹, Changqing Dong², Jianjun Cai³, Xu Cheng⁴

Automotive Data Center China Automotive Technology&Research Center Co.Ltd
Tianjin China
¹ chenchen@catarc.ac.cn
² dongchangqing@catarc.ac.cn
³ caijianjun@catarc.ac.cn
⁴ chengxu@catarc.ac.cn

Abstract. Aiming at the problem that the current business process of the enterprise frequently changes and becomes more and more complex, this paper proposes a business generator model design scheme that is accomplished by domain-driven model design methodology and model-driven code implementation. This schema can support business restructuring and process management to meet the enterprise relatively flexible dynamic needs. Based on the methodology of domain-driven design, the smart form operation model and the workflow operation model are constructed to realize the business generator that can be directly generated through the interface layer operation. The scheme is verified compared with the common scheme and the results show that development efficiency increases and flexible dynamic requirements improve in some degree.

1. INTRODUCTION

With the deepening of software applications in all walks of life, many businesses rely more and more on IT systems. However, software development in the traditional sense is time-consuming and labor-intensive, and the risk is high. Even the common software has the problems of applicability and scalability, which greatly restricts the software application of the enterprise. In order to solve this problem, experts have launched research work on domain-driven design models at home and abroad to realize the design of business generator models with higher reuse rates and to improve enterprise software application and information processing efficiency.

Domain-Driven Design[1] (DDD) is a method proposed by Evans to deal with the core complexity of a software system. It emphasizes the importance of the domain model and does not care about the specific implementation of the underlying data storage. The domain model[2-3] is used as the core of the business analysis design. Business modeling is performed through model elements such as Entity, Value Object, Service, Repository, and Aggregate[4-6]. Allocating these model elements to each layer can effectively realize the encapsulation of the service generator and ensure the high cohesion and low coupling of each module in the service generator. Allows the system to better maintain and expand, and then achieve the purpose of agile development[7]. In 2006, the DDD method was applied to the Norwegian oil company's oil trading supply chain system[8-9]. Compared with the previous legacy systems, it is found that the new system architecture built using DDD is significantly better than the legacy architecture, and the system's scalability, performance, and code quality have been significantly improved.
Traditional business generators are basically developed for a specific application regardless of C/S mode or B/S mode. And the business generator that is developed out, the user basically has no ability to change, the flexibility is worse. In addition, due to the uncertainty of the requirements sorted out by business personnel, and the problem of understanding deviations between business personnel and developers, application systems often have problems such as the incompleteness of business generator data and the need to modify the program at any time according to requirements. This, in turn, leads to project delays, low code reuse, and increased development costs. To solve the above problems, this paper proposes a method of constructing a business generator based on domain-driven design. The business generator domain model is constructed by designing the smart form operation model and the workflow operation model, and it is continuously reconstructed and refined to improve the reusability and testability of objects in the business domain. The business generator designed by the method in this paper can quickly and effectively generate a form for the customer's proposed business needs and bind it to the corresponding workflow without modifying any program.

2. BASIC CONCEPT

Domain model: It is the various entity classes in the domain and their relationship to each other. DomainModel=<Entity, ValueObject, DomainService>. The Domain model object contains entity, value object, and domain service.

Business object: It is object of the model that need to be distinguished. BizObject=<properties,operations>, is a combination of data and behavior, where properties is a collection of properties of the business object, representing n tuples <p_1,p_2,p_3,…,p_n>; Operations is a collection of actions that the object has, represented as m tuples <o_1,o_2,o_3,…,o_m>.

Business operation: operation=<input, output, atomicOperations,∏,C>, Where input represents the input, output represents the output of the operation, and atomicOperations represents the set of atomic operations contained in the operation. ∏=[input,atomicOperations]→output, indicates that input will output after atomic operation. C represents the constraints that are satisfied after the operation is performed. Based on this form, the operation of the business object can be defined.

Entity: In domain-driven design, an entity is an object in a model that needs to distinguish individuals, and is a combination of entity data and behavior. The three parts are mainly composed of public attributes, public operations, and the transformation of attributes under operation. Entity=<properties={id,p_1,p_2,…,p_n-1}, operations={get, set, submit, accept, reject, otherOperations}, H>. Where properties are the property set of the entity, id represents a unique identification. Operations is the collection of operations, and H is the conversion of properties under operations.

ValueObject: An object identified by an object's attribute value, which combines several related attributes into one concept as a whole. ValueObject=<properties, operations>. Compared to entities, value objects represent only the descriptive characteristics of a certain concept. They have one less identifier than the entity, properties={p_1,p_2,p_3,…,p_n-1}.

Aggregate/AggregateRoot: <Entity, ValueObject> is an aggregation. It represents a set of domain objects and a complete domain concept, including entity and value objects. Each aggregation has a root entity. AggregateRoot=<Entity, ValueObject>. This root entity is called the aggregation root.

DomainService: This definition is a dual group, DomainService=<{repositoryInterface, other domainServices}, domainOperations>. The definition is a double tuple. The first element represents the DDD element that it depends on. Domain services depend on data access layer services and other domain service’s services. The second element is a collection of operational methods for domain services encapsulated based on business logic and it is used to identify bad handling or some variable business operations in the domain model.

3. SMART FORM OPERATIONAL MODEL

Smart forms are channels for information collection with high efficiency and accuracy. Through intelligent control of the user, users can flexibly design forms according to their needs. At the same time, because the form operation is connected to the back-end database, the data entered in the form...
can be automatically written into the back-end database. In addition, operations such as data verification and permission definition can also be performed. With smart form technology, developers and managers can quickly and easily build application systems. The smart form construction model proposed in this paper is composed of form objects and form data objects, as shown in Figure 1.

![Smart form operational model](image)

**Figure 1. Smart form operational model.**

### 3.1 Form operation model

Basic operations mainly include the addition, deletion, and modification of smart forms.

- **Definition 1 (new operation):**
  
  \[
  \text{NewOperation} = \langle \text{NULL}, \{\text{formObject}\}, \{\text{edit, persist}\}, \{\text{NULL} \rightarrow \text{formObject}\}, \{\text{formObject}.\text{state} = \text{NORMAL}\}\rangle.
  \]

  Where edit represents an edit operation in memory, persist represents the state of the object as NORMAL and persists, and formObject.state=NORMAL represents the state of the object after the completion of the new operation.

- For the operation of saving the form after modification, it should be seen whether the original form is associated with a data object. The same business object can be associated with different form objects, that is, a business object may correspond to multiple form objects.

- **Definition 2 (modify operation):**
  
  \[
  \text{ModOperation} = \langle \{\text{formObject}\}, \{\text{newFormObject}\}, \{\text{edit, save&relate}\}, \{\text{NULL} \rightarrow \text{formObject}_{\text{new}}\}, \{\text{formObject}.\text{state} = \text{NORMAL}, \text{newFormObject}.\text{state} = \text{NORMAL} & \text{newFormObject}.\text{modifiedBy} = \text{someone}\}\rangle.
  \]

  After the operation is complete, two form objects of new and old will exist if there is a data object associated with the original form. Moreover, the old form object cannot be overwritten, formObject⊥formDataObject→newFormObject indicates that the form is associated with a form data object.

- **Definition 3 (find operation):**
  
  \[
  \text{FindOperation} = \langle \{\text{conditions}\}, \{\text{formObjects}\}, \{\text{select}\}, \{\text{conditions} \rightarrow \text{formObjects}\}, \{\text{formObject}.\text{state} = \text{NORMAL}\}\rangle.
  \]

  In this definition, the object state returned by the lookup operation is limited to NORMAL, which enables the logical deletion of the business object.

  The delete operation is divided into physical deletion and logical deletion. Physical deletion is really removing the form object from the persistence media. Logical deletion is generally performed by setting the state of an object and achieving implicit deletion through state filtering during the search.

- **Definition 4 (physical delete operation):**
  
  \[
  \text{PhysicalDelOperation} = \langle \{\text{formObject}\}, \{\text{NULL}\}, \{\text{delete}\}, \{\text{formObject} \rightarrow \text{NULL}\}, \{\text{formObject}.\text{state} = \text{DELETED}\}\rangle.
  \]

  Where delete represents a permanent delete operation.

- **Definition 5 (logic delete operation):**
  
  \[
  \text{LogicDelOperation} = \langle \{\text{formObject}, \text{delFormObject}\}, \{\text{edit, persist}\}, \{\text{formObject} \rightarrow \text{delFormObject}\}, \{\text{delFormObject}.\text{state} = \text{DELETED}\}\rangle.
  \]

  As can be seen from this definition, logical delete is a special modification operation, which is to modify the status to DELETED.

### 3.2 Form Data Manipulation Model

After the form template is generated, the form can be filled out and the data manipulation can be performed according to the rules. The data objects in the form use the state to distinguish.

- **Definition 6 (Create and Save Form Data Operations):**
  
  \[
  \text{NewSaveOperation} = \langle \{\text{NULL}\}, \{\text{formDataObject}\}, \{\text{edit, save}\}, \{\text{NULL} \rightarrow \text{formDataObject}\}, \{\text{formDataObject}.\text{state} = \text{NORMAL/DRAFT} & \& \text{formDataObject}.\text{createdBy} = \text{someone} & \& \text{formDataObject}.\text{ID} = \text{someone}\}\rangle.
  \]

  The new Save Form Data operation is similar to the New operation, except that the status of the form data object can be set to draft status.
DRAFT. From a business logic point of view, if individual create a new object that saves form data and the status is DRAFT, others cannot see it. Only the creator of someone can see and continue to open, edit, store. When the creator changes the status to NORMAL, other people can see the form data. According to definition 3, the general search operation cannot obtain a new form of data object whose status is DRAFT, thereby ensuring the concealment of the new draft form data object. In addition, individuals need to set its associated form object ID when creating a new form of the data object. The data can be loaded into the corresponding form only in this way.

Definition 7 (Modify and save form data operations): ModSaveOperation=<{formDataObject}, {newFormDataObject}, {edit, save&relate}, {formDataObject→newFormDataObject}, {formDataObject.state=NORMAL/DRAFT&&formDataObject.modifiedBy=someone}>.

Definition 8 (Find form data operations): FindFormDataOperation=<{formDataObject.createdBy=someone||formDataObject.modifiedBy=someone&Find.input}, {formDataObjects}, {select}, {conditions→formDataObjects}, {formDataObject.state=NORMAL/DRAFT&&formDataObject.createdBy&&formDataObject.state≠DELETED}>. If it is to find a new form object, individual need to enter the formDataObject.createdBy creator as a filter. If it is to find the modify object, enter the formDataObject.modifiedBy modifier as a filter condition.

4. WORKFLOW OPERATION MODEL
Workflow processing can be thought of as a composite operation over business object operations. For example, a form approval process can be viewed as inserting a process operation during the creation of a new form of the business object, as shown in Figure 2. After the approval is passed, the form object is created successfully. After the approval fails, the creation of the form object fails.

Figure 2. Form approval process.

Definition 9 (Workflow processing): WorkProcess=<{nodes, transitions, operations, states, Σ}>. Where nodes are the set of processing nodes in the workflow, including the start node. Transitions are predefined flow paths between nodes, include serial, parallel, and decision-making. Operations={start, accept, reject} is the related operation in process processing. Start is the first node to initiate the process operation, accept and reject are the processing operations in subsequent nodes. Accept is to continue the process according to the transitions, and reject is the termination process. States={SUCCESS, FAIL, DEALING} indicates three states of process success, failure, and processing, respectively. Σ is that different nodes can make the process in different states under different operations. That is, {nodes, transitions, operations}→states.

Definition 10 (Initiating process): StartProcess=<{NULL&p.n0}, {pendingObject&p.n1}, {edit, persist&start}, {NULL→pendingObject, p.n0→p.n1}, {pendingObject.state=PENDING&p.n1≠p.nend}>. Where p.n0 is the process initiation node in the workflow, p.n1 is the first node in the workflow, and p.nend is the process termination node in the workflow. Persist & start indicates that the process is initiated when the object is persisted. The status of the newly created object is PENDING, indicating that the process is still in progress.

Definition 11 (Agree and complete the process): AcceptProcess=<{pendingObject&p.nend}, {pendingObject&p.n1}, {persist&accept}, {pendingObject.state=NORMAL, p.nend→p.nend}, {pendingObject.state=NORMAL&&p.nend→true}>. When in the last step, the process automatically performs an accept operation and flows to the last node of p.nend, and the pendingObject state will become NORMAL.

Definition 12 (Reject and complete the process): RejectProcess=<{pendingObject}, {object}, {persist&accept}, {pendingObject.state→FAIL}>. When the processor rejects the flow, the reject operati-
on will be executed and the flow will fail. At the same time, the pendingObject state will change to FAIL.

Because the processing of modified objects in the workflow also needs to maintain the accessibility of the old objects, the modified object values cannot directly overwrite the old objects. Instead, they need to be temporarily stored in some way, and at the same time establish associations with the old objects.

Definition 13 (Initiate process modification): StartModifyProcess=<{object&pendingObject&n0}, {objectpendingObject&n1},{edit,persist&start}, {objectpendingObject→object→pendingObject, p.n0→p.n1}, {objectpendingObject.state=PENDING, pendingObject.state=PENDING&&p.n1≠p.nend}>. Where objectpendingObject represents pendingObject as a temporary object in the process of modifying the process, and associated.

Definition 14 (Agree and complete process modification): AcceptModifyProcess=<{objectpendingObject&m}, {object&mend}, {persist&accept}, {objectpendingObject↑object}, {object.state=NORMAL}>. Where objectpendingObject↑object means replacing object with a temporary object pendingObject and setting the state to NORMAL.

Definition 15 (Reject and end process modifications): RejectModifyProcess= <{objectpendingObject&m}, {object&mend}, {reject}, {objectpendingObject↓object}, {object.state=NORMAL, pendingObject.state=FAIL}>. The objectpendingObject↓object indicates that the modification of the pendingObject is discarded, and the state of the object is set to NORMAL, and the pendingObject state is FAIL. The pendingObject cannot be deleted directly, because the user can also initiate a new process based on the modified pendingObject.

In process processing, if the process does not end, the state of the business object will not change, and will only advance the process to the next node.

Definition 16 (Agree and advance the process): AgreeProceed=<{pendingObject&m}, {p.nend}, {proceed}, {p.nm→p.nm+1}, {pendingObject.state=PENDING&&p.nm+1≠p.nend}>. For new and modified operations, agreeing and moving to the next step has no effect on business objects.

5. Business Generator Model DDBizModel

5.1 Model Introduction
DDBizModel is a set of business generator model based on domain-driven design. It consists of entity, value object, aggregation, storage, repository, and service. Among them, entity, value object, and aggregation not only have data but also have rich business behaviors. They are the main elements of the logic of the packaged business. Aggregation controls access to internal objects. External access to internal objects must be achieved through aggregation. Only in this way, the business logic constraints within the aggregate can be guaranteed. Repository is a storage repository for resources and shields the underlying persistence technology from the outside. It can store three types of elements: entity, value object, and aggregation. The factory is responsible for building the domain object, similar to the factory in the design mode. Services are divided into Application Service and Domain Service. Application Service focuses on the definition of functions and is equivalent to an external facade. The Domain Service focuses on implementing business logic in which the logic that belongs to entities, value objects, and aggregations is encapsulated.

5.2 Model layer
According to the domain-driven design methodology, the business generator model is divided into domain layer, application layer, presentation layer, and infrastructure layer, as shown in Figure 3. The core is the domain layer. In this layer, business logic is encapsulated with design elements such as entity, value object, service, and factory to avoid exposing the business logic to other layers, then loose coupling with other layers can be realized, so the replacement of other layers can be realized with the least cost.
Definition 17 (Hierarchical model): \( \text{DDDBizModel} = \langle \text{DomainLayer}, \text{ApplicationLayer}, \text{InfrastructureLayer}, \text{ UILayer}, \ H \rangle \). Among them, \( \text{DomainLayer} = \langle \{ \text{Entity}, \text{ValueObject}, \text{Aggregate}, \text{DomainService}, \text{Factory}, \text{RepositoryInterface} \}, \{ \text{InfrastructureLayer} \} \rangle \) represents the domain layer, and mainly includes entities, value objects, aggregations, domain services, factories, and repository interfaces.

\( \text{ApplicationLayer} = \langle \{ \text{AppService} \}, \{ \text{DomainLayer}, \text{InfrastructureLayer} \} \rangle \) represents the application layer, which mainly contains application services and is dependent on the domain layer and infrastructure layer.

\( \text{InfrastructureLayer} = \langle \{ \text{RepositoryImplementation} \}, \{ \text{DomainLayer} \} \rangle \) represents the infrastructure layer, which mainly contains the implementation of \( \text{RepositoryImplementation} \) of the data warehouse, which depends on the data access interface and entity of the domain layer.

\( \text{UILayer} = \langle \{ \text{Controller} \}, \{ \text{ApplicationLayer}, \text{DomainLayer} \} \rangle \) indicates the interface layer, which mainly contains the interface control class and depends on the application layer and domain layer.

\( \ H \) represents the dependency call relationship between layers, as shown in Figure 2. For simplicity, similar operations such as addition, deletion, modification, and finding, just follow the A→B1 call path. This part of the operation can be achieved through some template methods. For the operation of more complex business logic, its business logic is mainly encapsulated in \( \text{DomainService} \) and \( \text{Entity} \), so it basically follows the path of A→B2→C1→D. For some of the more complex construction logic operations, the path will follow A→B3→C2 and the construction logic will be encapsulated in the factory. For example, an object's attributes are spliced from the attributes of other objects according to business logic.

5.3 Model implementation

\( \text{DDDBizModel} \) is a business generator model that combines smart forms and workflows. Through this model, a business operation module can be generated quickly and efficiently.

The entire development framework is based on the JAVA language and includes four independent components that correspond to the domain layer, data access layer, application service layer, and UI layer. In the domain layer, workflow domain services, form domain services, and form data domain services are encapsulated, and their respective business logics are implemented. The data access layer encapsulates the method of accessing data sources and the data access abstract class \( \text{GenericRepository} \). This class contains the persistence methods common to business objects, such as get, select, insert, and other atomic operation logic. This class can be inherited or extended as needed. The \( \text{AbstractServiceService} \) class is predefined in the application service layer and uses the facade pattern in the design pattern to encapsulate business operations. This class directly exposes the public persistence interface method of the business object, and the UI layer can directly call it. The UI layer defines the framework of the front-end interface, including menu loading, page loading, and so on.

5.4 Case Analysis

This article implements a business trip application process based on \( \text{DDDBizModel} \). The main requirements include: a business trip application form is automatically created by the business generator. When the applicant fills out the business trip application form, the form can be saved as a...
draft, the draft editing can be continued until the final application is submitted, and the business trip application form can be configured to implement the function of whether to approve and which person to approve.

In the UI layer, the FormController object is created, which includes four methods: initFormView, initFormCreate, modifyForm, and deleteForm. They are used to process the logic of loading the form, creating a new form, modifying the form, and deleting the form. There is also a FormDataController object that contains the following methods: initFormDataView, formSubmit, formSave, and modifyFormData. They are used to process the loading logic of form data, submit form data, save form draft data, and modify form data.

In the application service layer, a new application layer service class can be established by inheriting the basic service class, and the get, insert, submit, and save methods of the basic service class can be directly called, and can also be extended as needed.

The persistence method in the GenericRepository can be directly used in the data access layer, so there is no need to set up a data access class specifically.

In the domain layer, Form and FormData entities are created. Among them, FormData contains a collection of Forms, forming an aggregate root, and persisting the FormData while persisting operations on the FormData. The default WorkFlowDomainService is a service for workflow processing. The newProcess, startProcess, and acceptProcess methods are used to control the process creation, startup process, and approval process for incoming processing objects.

From the above design development process we can see:

1. DDDBizModel defines the definition of each layer clearly. The common operations of the business system are modeled, such as smart form operation model and workflow operation model, which can guide system design and code development.

2. For general business logic, this service generator can be automatically generated without modifying the program.

3. It can improve development efficiency and reduce defects. Public and conventional business logic has been encapsulated in the development framework, and business systems do not require coding for direct invocation.

6. CONCLUSION

This paper proposes a business generator model DDDBizModel based on domain-driven design. This model encapsulates smart form operations and workflow operation logic, and provides guidance for DDD design and development. The contribution of this article mainly includes the following points:

1. Field modeling of operations on business objects and unification of operations.

2. A business generator model based on DDD is proposed. The hierarchical model, smart form operation model and workflow operation model are defined, and the allocation of DDD elements at each layer is clarified.

3. According to the model, the development framework is implemented, and the public business logics such as workflow, form operations and form data operations are encapsulated, which can better support domain-oriented design and development, improve development efficiency, and reduce defects.

References

[1] Evans. Domain-Driven Design: Tacking Complexity In the Heart of Software[M]. 2004.

[2] Fowler M. Analysis patterns reusable object models[J]. Computational Linguistics Medical Informatics Software Engineering, 1997, 10.

[3] Martin Fower. Patterns of Enterprise Application Architecture[M]. Machinery Industry Press, 2010-4-1.

[4] Yan Xinzhe. Research and Application of Domain Driven Design Method[D].Nanchang University,2010.
[5] Hu Jidong, Ju Weigang. Research on Test Cases Design and Development Method Based on Domain-driven[J]. *Computer Technology And Development*, 2016, 26(09):56-60.

[6] Li Yin, Yuan Feng. Application model based on domain-driven design[J]. *Computer Engineering and Applications*, 2013, 49(16):1-8.

[7] Luo Qian, Ye Hua, Yang Yanlan. Design and implementation of MES software based on domain-driven design[J]. *Machine Design and Manufacturing Engineering*, 2017, 46(11):24-28.

[8] Wesenberg H, Landre E. Using domain-driven design to evaluate commercial off-the-shelf software[C]// *Companion To the, ACM Sigplan Symposium on Object-Oriented Programming Systems, Languages, and Applications*. ACM, 2006:824-829.

[9] Landre E, Wesenberg H, Olmheim J. Agile enterprise software development using domain-driven design and test first[C]// *Companion To the, ACM Sigplan Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. DBLP, 2007:983-993.

[10] Tang Wenzhong, Mo Weidong. Model-driven intelligent form system frame oriented to fields[J]. *Journal of Beijing University of Aeronautics and Astronautics*, 2007, 33(9):1086-1089.