FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Designing WYSIWYG Web Forms
Student:	Radek Buša
Supervisor:	Ing. Marek Skotnica
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2018/19

Instructions

- Design a data representation of the DEMO Fact Model: model and instances.
- Design a method of assigning web forms to individual C-Acts and their data representation. Consider the relation to the Action Model.
- Demonstrate the designed data structures on the Rent-a-car case study.
- Design, implement and test a web WYSIWYG editor for the data structures.
- Design, implement and test a web user environment for displaying the forms and storing the data into a relational database.
- Again, demonstrate on the Rent-a-car case study.
- Document your solution.

References

Jan L.G. Dietz. (2012). *The Essence of Organization - an Introduction to Enterprise Engineering*. Sapio bv.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 13, 2017

Bachelor's thesis

Designing WYSIWYG Web Forms

Radek Buša

Department of Software Engineering
Supervisor: Ing. Marek Skotnica

May 8, 2018

Acknowledgements

In the first place, I would like to thank my supervisor Ing. Marek Skotnica who proved to be a patient and erudite mentor. Next, I would like to thank my family for their unconditional and endless support.

Declaration

In Prague on May 8, 2018 . . . .

Citation of this thesis

Buša, Radek. *Designing WYSIWYG Web Forms.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato práce se věnuje tvorbě prototypu webové aplikace pro navrhování webových formulářů na základě modelů podnikových procesů. Z takové aplikace by mohla mít užitek každá společnost, která by chtěla vymodelovat své podnikové procesy a na základě nich rychle vytvořit formuláře na web.

Klíčová slova navrhování WYSIWYG webových formulářů, webová aplikace, modelování v metodice DEMO, modelování podnikových procesů, návrh webové aplikace, implementace webové aplikace, testování webové aplikace, JavaScript, C#

Abstract

This thesis is dedicated to creating a proof-of-concept prototype web application for designing web forms based on business process models. Such solution will help every company which would like to model their business processes and quickly create forms based on the business process models.

Keywords WYSIWYG form design, web application, DEMO methodology modelling, enterprise modelling, web application design, web application implementation, web application testing, JavaScript, C#

Contents

List of Figures

List of Tables

Listings

Introduction

State-of-the-art

Information systems in present-day companies are in most cases custom-made, which brings problems of being time-taxing to produce and often expensive to maintain. In recent years, low code platforms, BPM systems and collaborative platforms such as Microsoft SharePoint have been emerging, allowing its users to support many important tasks of information systems.

Thesis Goals and Result

The primary goal of this thesis is to design, implement and test a proof-of-concept web application for designing web forms based on business process models with additional functionalities to draw DEMO Fact Models and submit the forms to a relational database. Secondary goal is to design a data model for DEMO Fact Model and related forms which the application will take advantage of. Both goals will be demonstrated on the Rent-A-Car case study from the book *The Essence of Organisation.*

The result of this thesis is intended to help everyone who wants to model their business processes or already has DEMO models and would like to use their models to generate web forms from them.

Motivation

I chose this topic because nobody implemented such solution for DEMO methodology before. Implementing such solution will remove the obstacles of creating web forms from scratch while already having complete DEMO models and might possibly increase market share of DEMO methodology among other enterprise modelling methodologies.

Chapter 1

Theories Used

1.1 Business Process Management (BPM)

Mathias Weske in his book [2] briefly explains what is *Business process management*:

> Business process management includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes. The basis of business process management is the explicit representation of business processes with their activities and the execution constraints between them. Once business processes are defined, they can be subject to analysis, improvement, and enactment.

1.2 DEMO Methodology

Akiyoshi Araki and Junichi Iijima [3] will help with brief introduction to *Design & Engineering Methodology for Organizations* (DEMO) to fellow reader:

> DEMO is a methodology for the engineering and implementation of organizations. It can reveal the essential structure of business process and simplify the structure of organization by an ontological model which describes the core of the organizations. DEMO is based on the Performance in Social Action theory, PSI- or ψ-theory. The ψ-theory consists of four axioms and one theorem, i.e. the operation axiom, the transaction axiom, the composition axiom, the distinction axiom and the organization theorem.
>
> - The operation axiom states that the operation of the organization consists of the activities of actors who perform two kinds of acts; production acts (P-acts) and coordination acts (C-acts). By performing P-acts, the actors contribute

to achieving the purpose or the mission of the enterprise. By performing C-acts, the actors enter into and comply with commitments towards each other regarding the performance of P-acts.

- The transaction axiom states that C-acts are performed in transactions that always involve initiator and executor. They aim to achieve a particular result, the P-fact.
- The composition axiom states that transactions are related to each other, i.e., a transaction is enclosed in another transaction, or a transaction is self-activated.
- The distinction axiom states that there are three distinct human abilities playing a role in the operation of actors, i.e., performa, informa and forma ability which relates to ontological action, infological action and documental action, respectively. Actors who use the performa ability to perform P-acts are called business actors (B-actors). The performa ability is the essential human ability for doing business. Actors who use the informa and forma ability to perform P-acts are called intellectual actors (I-actors) and documental actors (D-actors), respectively.
- The organization theorem states that the organization of an enterprise is an integrated social system of B-organization, I-organization and D-organization.

The ontological model of an organisation is composed of four aspect models (submodels), each capturing the organisation from different aspect, as illustrated in Figure 1.1:

- Construction Model (CM),
- Action Model (AM),
- Process Model (PM),
- Fact Model (FM). [1]

The aspect models this thesis will work with will be described in the following chapters.

1.2.1 Fact Model

Alicia Perinforma in the book *The Essence of Organisation* [1] sums up the purpose of Fact Model among other DEMO aspect models:

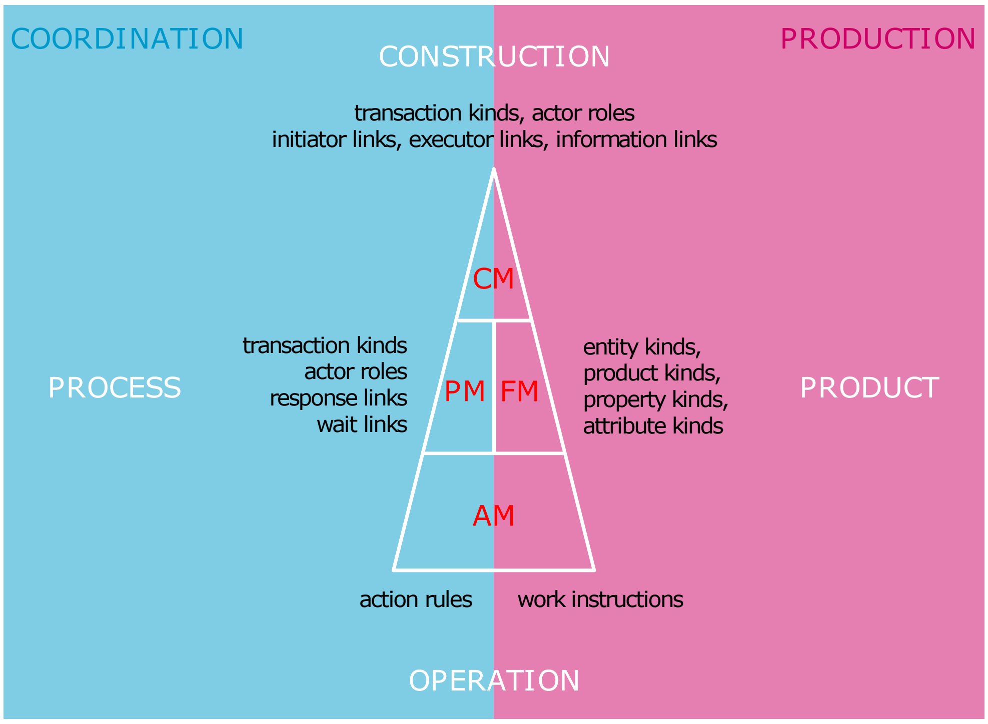


Figure 1.1: DEMO aspect models [1]

> The Fact Model shows the fact kinds in the production world of the organisation and their interrelationships. In current practice, they are called business objects and business facts. In addition, the FM contains the laws that must be obeyed in order to keep every state and every state transition of the production world lawful. As these laws are the declarative version of the (imperative) business rules, as specified in the AM, they are called business laws.

1.2.2 Action Model

Alicia Perinforma [1] in the book also describes the purpose of Action Model among other DEMO aspect models:

> The Action Model (AM) is the most comprehensive aspect model, in the sense that the other three may be derived from it. The AM of an organisation consists of the action rule specifications for every internal actor role. Action rules are guidelines for dealing with the events that actors have to respond to. In current practice, they are often referred to as business rules. In addition, the AM may contain work instructions. These regard the execution of production acts. Work instructions may be useful if the production

acts are material, like baking pizzas. However, also for immaterial products, like judgments and decisions, there can be work instructions. In practice, quite complicated judgment or decision rules or protocols may apply.

Chapter 2

Analysis

This section elaborates on analyzing thesis requirements.

Based on the requirements, a comprehensive functional specification with wireframes of the application in section 2.3 and implementation independent specification of DEMO Fact Model data model and Form Model in section 2.4 are defined.

2.1 Goals

- Implementing a subset of DEMO Fact Model specification sufficient for capturing the Rent-A-Car case study.
- Implementing a WYSIWYG diagram editor for drawing Object Fact Diagram which will save model represented by the diagram to a relational database.
- Implementing a WYSIWYG web form designer for designing web forms based on given Fact Model.
- Implementing a simple interface for rendering and submitting filled-out forms designed by the form designer to a relational database.

2.2 Non-Goals

- Implementing multi-domain solution.
- Implementing the Revocation Patterns.
- Implementing the user interface to comply with the best practices of UX.
- Performance.

- Security.

- Implicit propagation of Object Fact Diagram changes to the forms.

2.3 Functional Specification

DEMO Form Designer will be used for specifying Fact Model by drawing an Object Fact Diagram of given domain and designing WYSIWYG web forms based on attributes and properties of Fact Model mentioned earlier. Furthermore, the application will be able to assign the forms to C-acts in the *Standard Transaction Pattern* depicted in Figure 2.1.

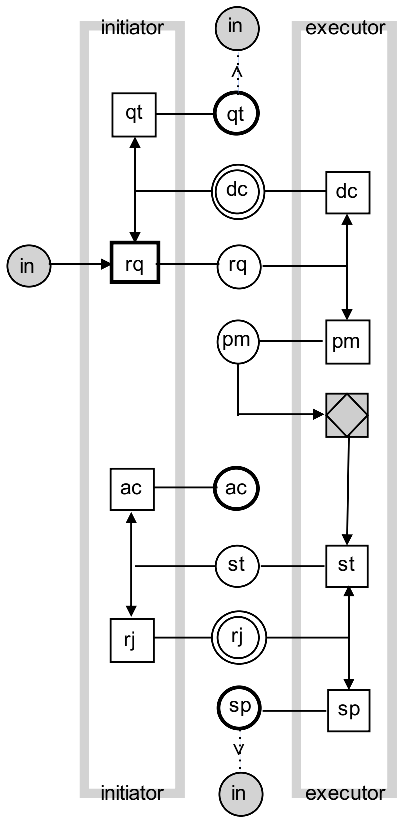


Figure 2.1: Standard Transaction Pattern [1]

The application will also provide a testing interface for submitting the data from completed forms into a relational database.

2.3.1 Process Designer

Process Designer will be used for assigning transaction kinds discovered from the Fact Model to business processes. It will serve as a simplified substitution of full Construction Model implementation.

The interface of Process Designer designated in Figure 2.2 will consist of a panel and boxes where each of them represents a single business process.

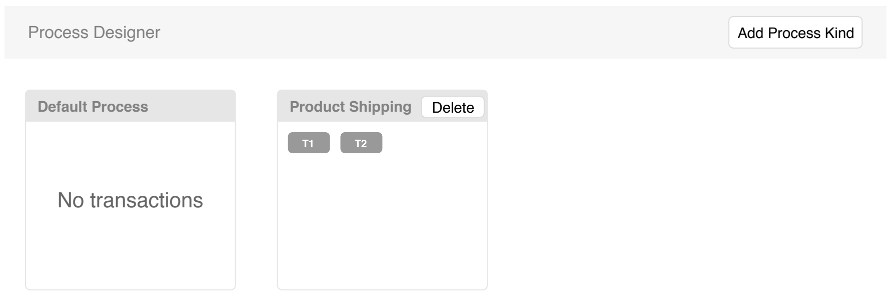


Figure 2.2: Process Designer interface wireframe

By default, all the transaction kinds discovered from the Fact Model will be assigned to a process called *Default Process*.

The user will be able to define additional business processes using the *Add Process Kind* button. After clicking it, a pop-up window illustrated in Figure 2.3 for specifying new business process name will be presented to the user.

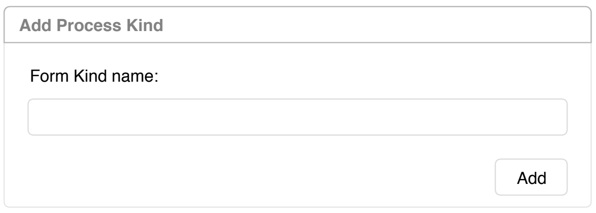


Figure 2.3: Process Designer pop-up window interface wireframe

Apart from creating a custom business process, the user will also be able to delete them using the *Delete* button which every process box possesses, except for Default Process box, which will be reserved for newly discovered transaction kinds from the Fact Model.

Transactions themselves will be assigned to a certain business process by drag-and-drop method to a box representing given business process.

Process boxes which will not have any transaction kinds assigned will warn the user about this fact.

2.3.2 Data Designer

Data Designer will be used for drawing Object Fact Diagram of given domain. Data Designer interface illustrated in Figure 2.4 will consist of a canvas with diagram preview and two toolbars – upper toolbar with action buttons and left toolbar with diagram element picker and edge toggle switch.

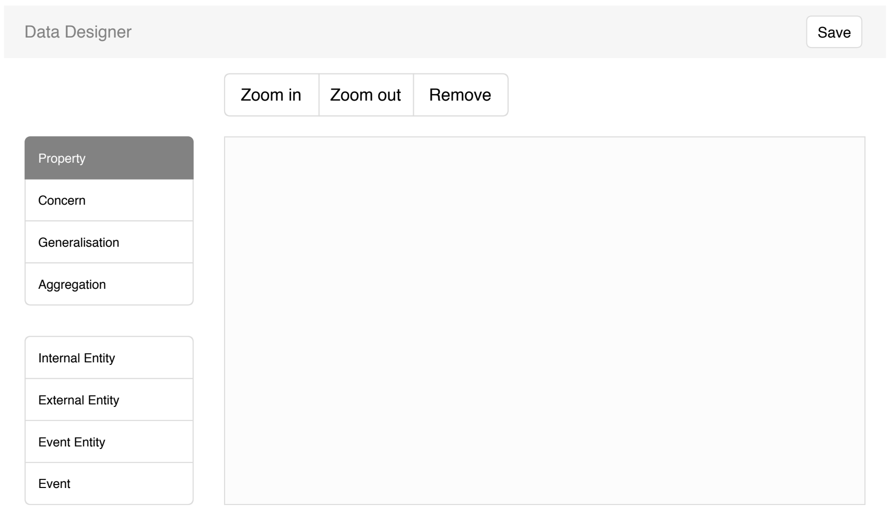


Figure 2.4: Data Designer interface wireframe

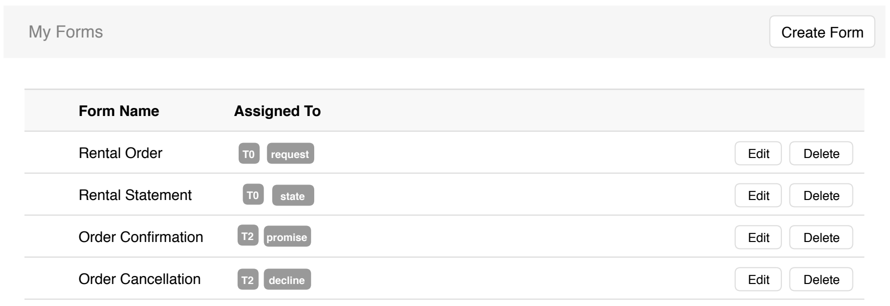
The elements on the canvas will be able to be moved around the canvas, further adjusted (e.g. size, position or content), connected together, deleted from or moved around the canvas. When done drawing the diagram, the user will be able to save it. Saving the diagram will imply that any other content of the application (i.e. forms and processes) will be deleted.

2.3.3 Form Editor

My Forms

After entering the *Form Editor* section, the *My Forms* screen designated in Figure 2.5 will automatically open, containing a list of all user-defined forms. The list will contain the name of the form, and form assignment to a transaction kind and C-act pair. The user will be able to create, edit or delete his forms on this screen.

- If the user decides to create a new form, the *Form Creator* screen described in section 2.3.3 will open with an empty editing area.
- In case the user will want to edit a form, the Form Creator screen will open with requested form.



Form Name	Assigned To		
Rental Order	T0 request	Edit	Delete
Rental Statement	T0 state	Edit	Delete
Order Confirmation	T2 promise	Edit	Delete
Order Cancellation	T2 decline	Edit	Delete

Figure 2.5: My Forms interface wireframe

Form Creator

This screen depicted in Figure 2.6 will serve as a WYSIWYG form editor for creating or editing selected form. The Form Creator interface will comprise of *editing area* and *field area*.

The field area will contain two types of fields:

- Generic Fields – fields which will be used to supply additional useful information and elements to the form for submitters such as headings, paragraphs, etc.
- Model Fields – fields which will correspond to the attributes and properties of Fact Model data model described in section 2.4.1.

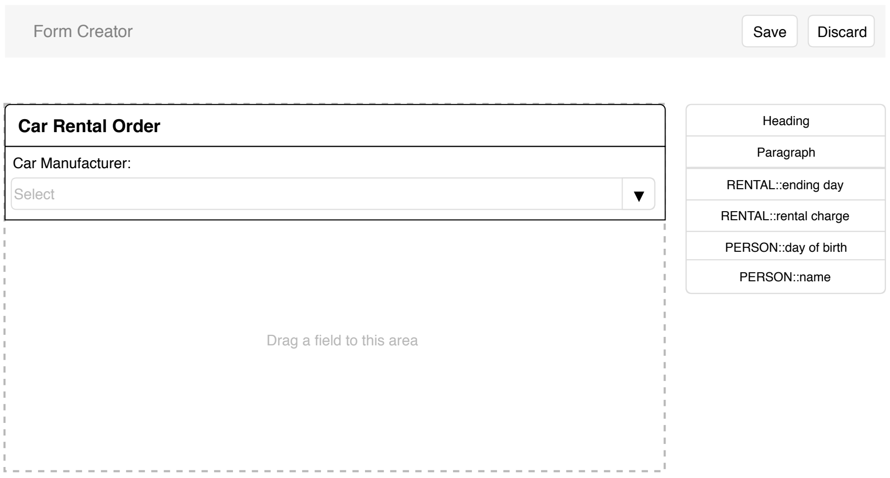


Figure 2.6: Form Creator interface wireframe

The user will be able to drag and drop any of present fields from the field area into the editing area which serves as a live preview of the form. The user will be able to arrange and further customize the fields within this area. After done editing, the user will be able to return to the My Forms screen by pressing either the *Save* or *Discard* button.

- After clicking the Save button, the procedure of saving the form described in section 2.3.3 will commence.
- After clicking the Discard button, all unsaved work in the Form Creator will be lost.

Saving the form

The process of saving the form will consist of following consequential phases:

1. Checking for inconsistencies.

   The inconsistencies are:

   - There are two fields belonging to the same Model Field.

   If any inconsistencies are detected, the user will be warned by a warning message and will be given a chance to correct them. The process of saving the form will not continue until the form does not contain any inconsistencies.

2. Naming the form and assigning it to a transaction and C-act.

   The user will be presented an interface like in Figure 2.7 where the form can be assigned to a pair consisting of a transaction and C-act from the Standard Transaction Pattern of DEMO transactions.

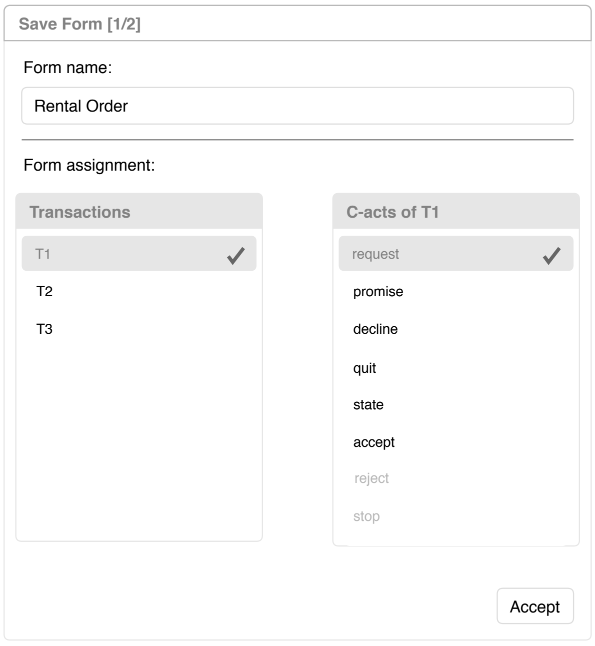
   If another form will be assigned to a pair (Transaction, C-act), the user will not be able to assign the form to the same pair and those assignment options will be disabled in the interface. After assigning the form to an available pair, the Accept button becomes available. After the user clicks the Accept button, the process will continue to step 3.

3. Connecting the form to a Fact Model entity.

   An interface designated in Figure 2.8 will serve the user to connect the form to a single entity in Fact Model data model if needed. Only when connected to Fact Model, the form will read from the database when opened and write to the database when submitted.

   There are two options how to connect the form to an entity:

   - *Instantiate new entity* option will instantiate selected entity and add it to a list of selected transaction entities if not already present there, and fill its attributes and properties with form field data when submitted.


Save Form [1/2]

Form name:

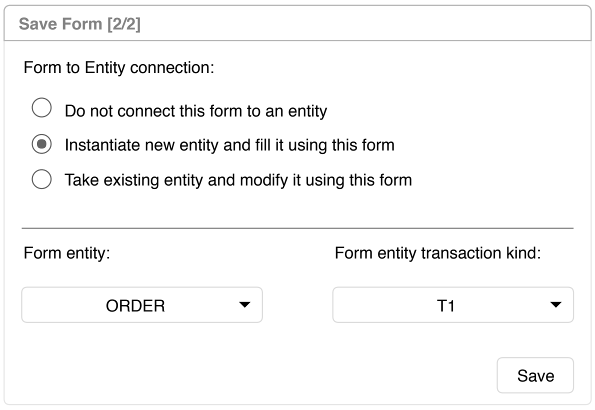
Rental Order

Form assignment:

Transactions	C-acts of T1
T1 ✓	request ✓
T2	promise
T3	decline
	quit
	state
	accept
	reject
	stop

Figure 2.7: Form naming and assignment interface wireframe

- *Take existing entity* option will find selected entity among transaction entities of selected transaction and fill it with form field data when submitted.


Save Form [2/2]

Form to Entity connection:

- Do not connect this form to an entity
- [✓ Instantiate new entity and fill it using this form]
- Take existing entity and modify it using this form

Form entity: ORDER

Form entity transaction kind: T1

Figure 2.8: Form to Fact Model connection interface wireframe

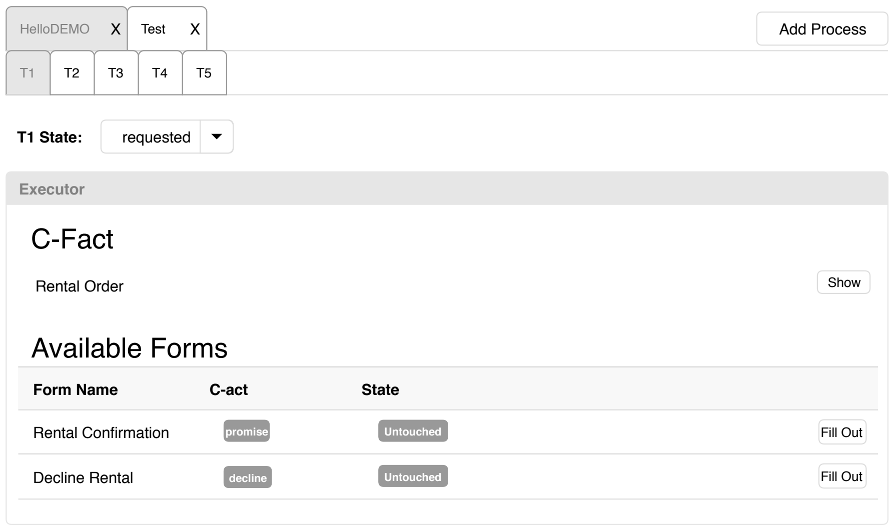
4. Persisting the form into the database.

2.3.4 Form Tester

This section will serve as a testing interface for filling out user-defined forms created in the Form Editor section of the application described in section 2.3.3.

Process Workspace

The *Process Workspace* screen depicted in Figure 2.9 will be available right after opening the Form Tester section. It comprises of process instance tabs, transaction tabs, list of available forms to fill in, C-fact inspector, transaction state dropdown and an *Add Process* button.



Form Name	C-act	State	
Rental Confirmation	promise	Untouched	Fill Out
Decline Rental	decline	Untouched	Fill Out

Figure 2.9: Process Workspace interface wireframe

If the workspace will be empty either because the user removed the processes from the workspace or there are no processes instantiated, process instances can be added to the workspace using the Add Process button. Clicking this button will open a simple interface depicted in Figure 2.10 where the user can either create a new process, or open an existing one.

- *Existing processes* panel allows the user to put an existing process to Process Workspace or delete it.
- *New process* panel allows the user to create a new process with given name and type and add it to the workspace afterwards.

Each process instance tab can be removed from the screen if necessary using the *X* button inside its tab. Under the process instance tab, there will be another set of tabs with transactions of given process instance. In the transaction tab body, there will be a drop-down menu where the user will be

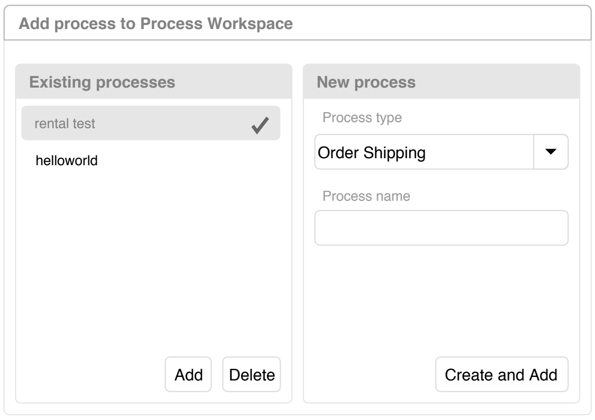


Figure 2.10: Add Process interface wireframe

able to explicitly set the state of selected transaction, a list of available forms by current transaction state and if the transaction will not be in *initial* state, a C-fact inspector.

States of the transactions will correspond to C-facts of Standard Transaction pattern depicted in Figure 2.1.

The list of forms will contain the following information:

- Form name which was set by the user in the Form Creator interface.
- Assignment to a transaction kind and C-act.
- State:
  - *Filled Out*: The form has been filled out and thus has created a C-fact. The user can either review the information of the C-fact or change the entered information using the *Change* button which will take the user to Form Filler described in section 2.3.4.
  - *Untouched*: The form has not yet been filled out and is waiting to be filled out to advance the transaction to the next state. The *Fill Out* button will take the user to Form Filler where the form can be filled out.

Form Filler

Form Filler illustrated in Figure 2.11 will serve as a form renderer for the user-defined forms in Form Editor described in section 2.3.3.

The user will be able to:

- Fill in the fields of the form and submit the form using the *Submit* button. After submitting the form, provided all the fields are correctly

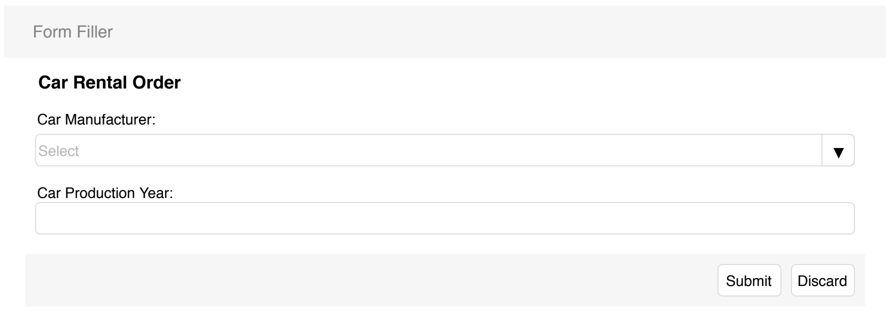


Figure 2.11: Form Filler interface wireframe

filled in, the user will be taken to Process Workspace screen described in section 2.3.4.

- Leave the form and go to Process Workspace by pressing the *Discard* button.

2.4 Data Model

This section contains implementation-independent specifications of all data models used within the application.

Fact Model data and instance models were influenced by *DEMOSL Fact Model metamodel.* [4]

2.4.1 Fact Model (Model)

Entities of this part of the model represent elements of *DEMO Object Fact Diagram.*

Domain model in Figure 2.12 depicts Fact Model data model. It consists of following classes:

EntityKind

`EntityKind` represents an *entity type* of OFD.

Each `EntityKind` can have multiple `generalisations` and multiple `specialisations` of type `EntityKind`; these associations are navigable from both ends.

Each `EntityKind` possesses definitions of its attributes using the `attributes` association.

In order to represent *properties* between `EntityKinds`, each `EntityKind` has `properties` and `inverseProperties` associations.

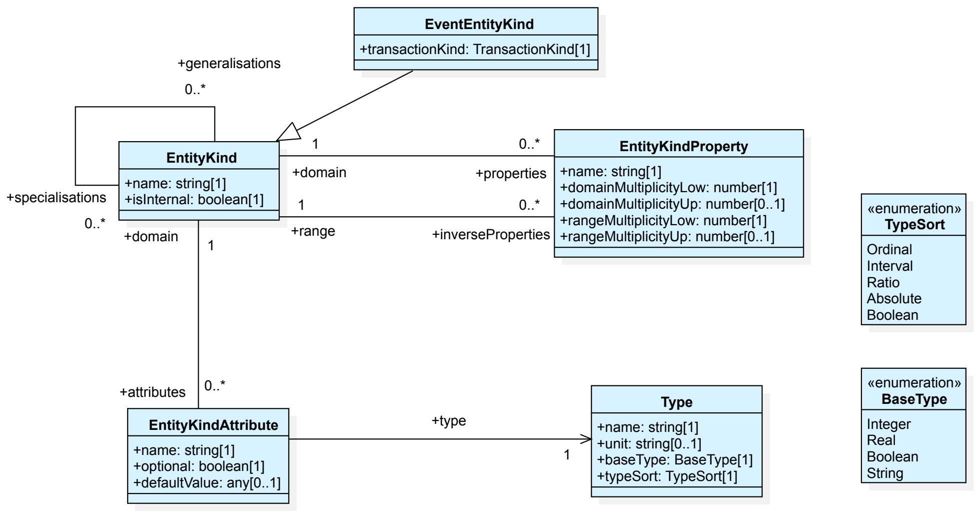


Figure 2.12: Domain diagram of Fact Model data model

Attributes:

name represents name of the entity.

isInternal determines whether the entity is internal or external.

EventEntityKind

`EventEntityKind` is a specialisation of `EntityKind`. It represents a *product kind*.

Attributes:

transactionKind determines which transaction kind this product kind is associated to.

EntityKindAttribute

`EntityKindAttribute` represents an attribute of `EntityKind`.

Each `EntityKindAttribute` has a `domain` association – i.e. which `EntityKind` it belongs to. Moreover, it incorporates, by the `type` association, which *value dimension* will its instance have.

Attributes:

name represents name of the attribute.

optional represents whether the value of an attribute is optional.

defaultValue represents default value of the attribute.

EntityKindProperty

`EntityKindProperty` represents a *property* of `EntityKind`.

Using the `domain` and `range` associations, the direction of the property association can be inferred.

Note: Because implementing multiplicities is a non-goal of this thesis, the attributes `domainMultiplicityLow`, `domainMultiplicityUp`, `rangeMultiplicityLow` and `rangeMultiplicityUp` are included in the model only for completeness sake and their behaviour will not be implemented.

Attributes:

name represents name of the property.

domainMultiplicityLow denotes lower bound of the multiplicity range on the `domain` side.

domainMultiplicityUp denotes upper bound of the multiplicity range on the `domain` side.

rangeMultiplicityLow denotes lower bound of the multiplicity range on the `range` side.

rangeMultiplicityUp denotes upper bound of the multiplicity range on the `range` side.

Type

`Type` represents *value dimension* of the `value` attribute of `EntityAttribute`.

Attributes:

name represents name of the type.

unit represents units of given type.

baseType denotes which primitive type will the `value` attribute of `EntityAttribute` contain.

typeSort denotes *scale sort* of the value dimension.

2.4.2 Fact Model (Instances)

Domain model of Fact Model instance model is illustrated in Figure 2.13. It comprises of following classes:

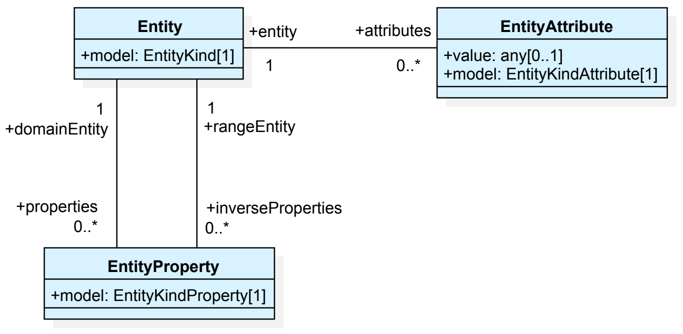


Figure 2.13: Domain diagram of Fact Model instance model

Entity

`Entity` represents an *entity*, i.e. an instance associated with `EntityKind`.

Each `Entity` has associated instances of its `attributes` and also instances of its `properties` and `inverseProperties`.

Attributes:

model denotes which `EntityKind` is this `Entity` instance of.

EntityAttribute

`EntityAttribute` represents instance associated with `EntityKindAttribute`.

Each `EntityAttribute` has an association called `entity` determining which `Entity` it belongs to.

Attributes:

value bears value of the attribute.

model denotes which `EntityKindAttribute` is this `EntityAttribute` instance of.

EntityProperty

`EntityProperty` represents a *property association* between two `Entity` objects.

Attributes:

model denotes which `EntityKindProperty` is this `EntityProperty` instance of.

2.4.3 Form Model

The main purpose of Form Model is to represent processes, their transactions and forms of given transactions.

Form Model domain diagram is portrayed in Figure 2.14. It consists of following classes:

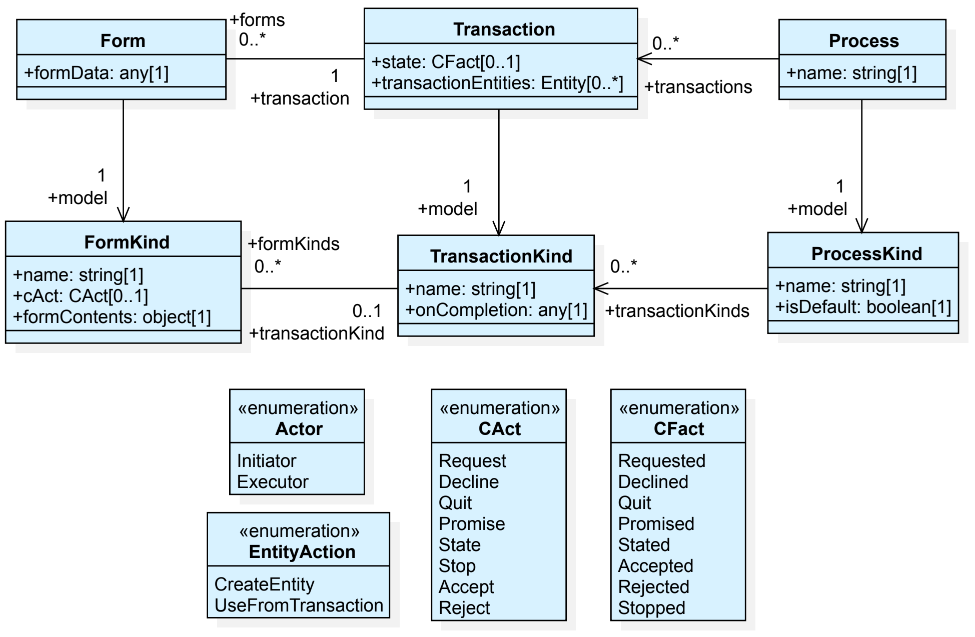


Figure 2.14: Domain diagram of Form Model

ProcessKind

`ProcessKind` represents a business process.

Its `transactionKinds` association represents a collection of transaction kinds it incorporates.

Attributes:

name represents name of the process, e.g. *Order Shipping*.

isDefault represents whether given `ProcessKind` is *Default Process* described in section 2.3.1.

Process

`Process` represents an instance of a business process.

Each `Process` refers to its `ProcessKind` using the `model` association and stores its transactions using the `transactions` association.

Attributes:

name represents name of the process instance, e.g. *John's rental arrangement.*

TransactionKind

`TransactionKind` represents a transaction kind. It stores a collection of `FormKind`s it incorporates using an association called `formKinds`.

Attributes:

name represents ID of the transaction kind, e.g. *T1.*

onCompletion represents *product kind formulation* of product kind associated to given transaction kind.

Transaction

`Transaction` represents a transaction as an instance of a transaction kind.

It aggregates its forms by its `forms` association and refers to its transaction kind using the `model` association.

Attributes:

state represents state of the transaction as a C-fact.

transactionEntities collection stores `Entity` objects created within scope of this transaction.

FormKind

`FormKind` represents a *form prototype*[1].

Using the `transactionKind` association, the `FormKind` object knows about its assignment to a certain transaction kind.

Attributes:

name represents name of the form, e.g. *Rental concluding request.*

cAct represents a C-act this `FormKind` is assigned to.

formContents represents a definition of form structure – an array of objects representing Model Fields and Generic Fields characterized in section 2.3.3.

Form

`Form` represents a *form instance*[2]. It contains concrete form field data after filling the form in.

[1]Form prototype is a declaration of form fields without concrete field values and concrete form-to-model bindings. It is composed of Form Fields and Generic Fields which are bound to EntityKindAttribute and EntityKindProperty objects. Form prototype will, when serving the form for filling out, be transformed to *form instance.*

[2]Form instance is a form with concrete values and concrete form-to-model bindings. When serving the form for filling it out, it will have field values filled in from the data model and its fields will be bound to EntityAttribute and EntityProperty objects.

Attributes:

formData represents values of form fields. This attribute will not be persisted into the database – instead will be populated based on other data model values on the fly when serving the form.

Chapter 3

Existing Solutions for Designing Web Forms

The objective of this chapter is to select an appropriate JavaScript plugin for the application with WYSIWYG web form design capabilities. Firstly, section 3.1 scrutinizes today's most popular online services for designing forms aimed at ordinary users. After researching the services for ordinary users and stating the most important features of them from the usability standpoint, the following sections 3.2 and 3.3 compare distinct plugins potentially usable in the application utilizing both criteria acquired in section 3.1 and criteria important for implementation.

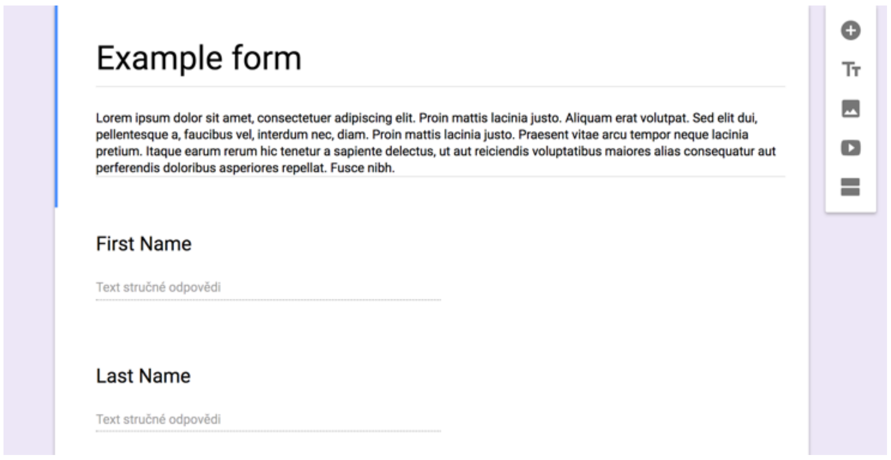
3.1 Popular Services with WYSIWYG Form Editing

3.1.1 Google Forms

Google Forms [5] illustrated in Figure 3.1 are a free online service for creating survey forms that can be shared with target audience via web links. Filled out forms are sent to the service and form author is afterwards able to visualize the data from respondents. Among the features of Google Forms are: drag-and-drop form editing, wizard creation, simple form validation interface with custom error messages and image or video embedding features.

3.1.2 Cognito Forms

Cognito Forms [6] captured in Figure 3.2 are a free online form builder with extra paid functionality. It supports advanced column layouts in conjunction with drag-and-drop form editing. It is business oriented, with a variety of payment oriented functionality, such as signature fields and integration with


Example form

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin mattis lacinia justo. Aliquam erat volutpat. Sed elit dui, pellentesque a, faucibus vel, interdum nec, diam. Proin mattis lacinia justo. Praesent vitae arcu tempor neque lacinia pretium. Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat. Fusce nibh.

First Name

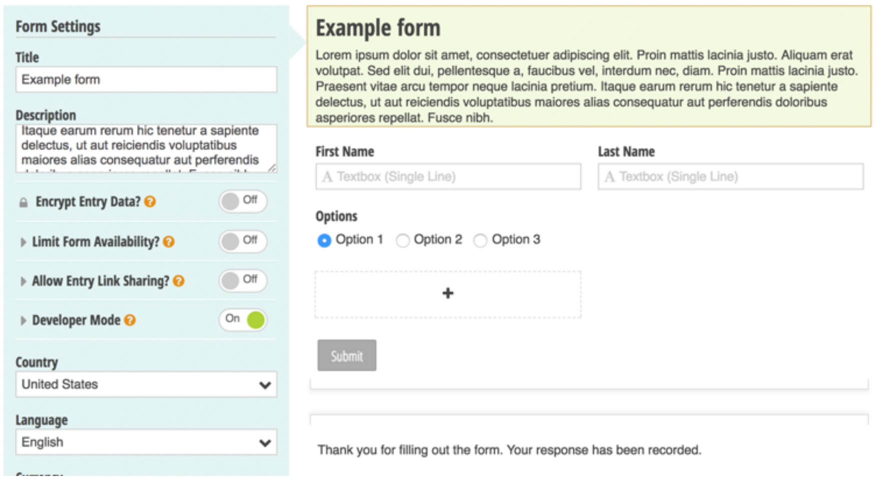
Text stručné odpovědi

Last Name

Text stručné odpovědi

Figure 3.1: Google Forms

payment services. It also supports advanced form elements, such as live calculations or conditional hiding of form fields.


Form Settings

Title: Example form

Description: Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis

Encrypt Entry Data? Off

Limit Form Availability? Off

Allow Entry Link Sharing? Off

Developer Mode On

Country: United States

Language: English

Example form

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin mattis lacinia justo. Aliquam erat volutpat. Sed elit dui, pellentesque a, faucibus vel, interdum nec, diam. Proin mattis lacinia justo. Praesent vitae arcu tempor neque lacinia pretium. Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat. Fusce nibh.

First Name: Textbox (Single Line)

Last Name: Textbox (Single Line)

Options: [✓ Option 1] Option 2 Option 3

Submit

Thank you for filling out the form. Your response has been recorded.

Figure 3.2: Cognito Forms

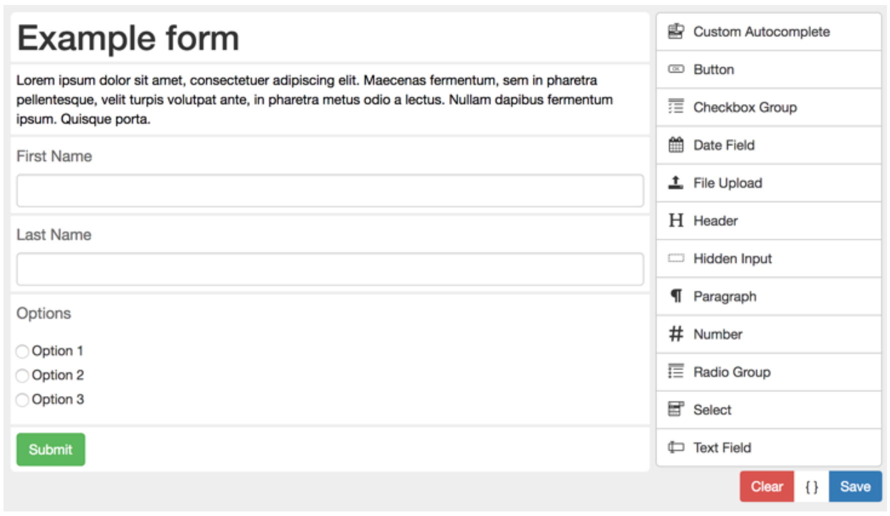
3.1.3 Conclusion

To sum this section up, today's popular services with WYSIWYG form designing functionalities for ordinary users are from usability standpoint generally easy to use by utilizing drag-and-drop form design capabilities along with simple user-friendly graphical interfaces consisting of easy to understand icons, options and dialogues.

3.2 Open Source Plugins

3.2.1 jQuery formBuilder

As of December 2017, an open source, MIT licensed *jQuery formBuilder* [7] depicted in Figure 3.3 is today's most popular plugin for building web forms among developers, according to the number of stars on GitHub among researched open source plugins [8][9][10][11]. As its name suggests, it depends on jQuery. As a companion for formBuilder, there is a separate plugin called *formRender* which serves as a renderer for the forms created using jQuery formBuilder [12].


Example form

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas fermentum, sem in pharetra pellentesque, velit turpis volutpat ante, in pharetra metus odio a lectus. Nullam dapibus fermentum ipsum. Quisque porta.

Figure 3.3: jQuery formBuilder

Advantages

- Exhaustive documentation.
- Bootstrap ready but does not depend on it.
- Easily extendable using custom API.

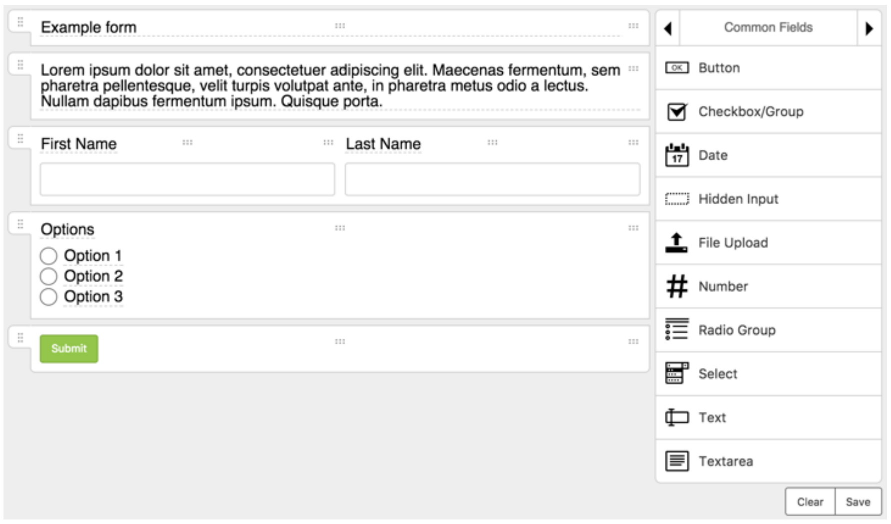
Disadvantages

- Does not support column layouts.

3.2.2 Formeo

Another open source MIT licensed plugin [9], similar to jQuery formBuilder mentioned in section 3.2.1. Figure 3.4 depicts user interface of the plugin.

Unlike jQuery formBuilder, this plugin, however, features advanced column layouts for the form fields.



Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas fermentum, sem pharetra pellentesque, velit turpis volutpat ante, in pharetra metus odio a lectus. Nullam dapibus fermentum ipsum. Quisque porta.

Figure 3.4: Formeo

Advantages

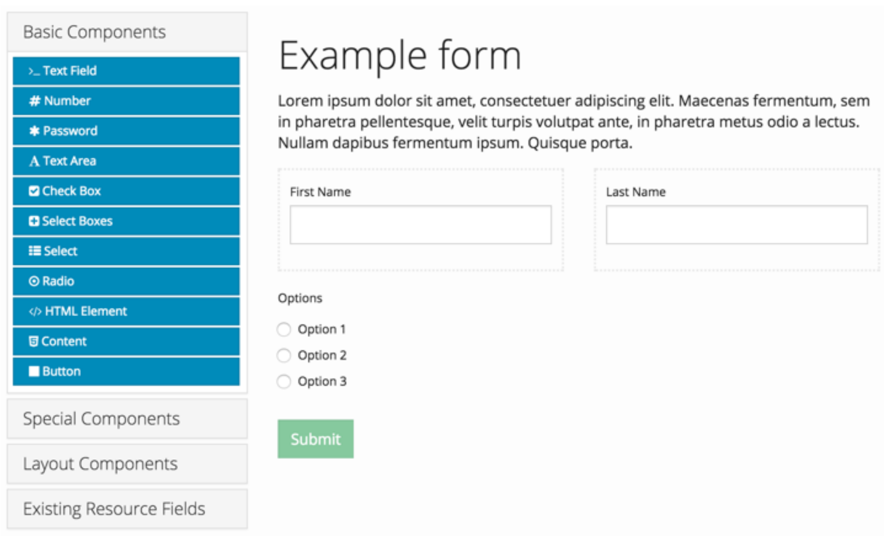
- The plugin has no dependencies.
- Supports column layouts.
- Easy drag-and-drop form layout adjustment.

Disadvantages

- Form builder and renderer are not separate plugins.
- Poor documentation – most importantly there are no examples on how to customize the plugin.
- As of January 2018, the project received last minor update in May 2017 [9].

3.2.3 form.io Form Builder Application

This open source MIT licensed plugin [10] is illustrated in Figure 3.5 and is developed by the provider of commercial service *form.io* which is a form-oriented data management platform for serverless web applications [13]. Compared to similar plugins, this plugin has advanced features, namely advanced form validation, custom error messages and CSS styling. This plugin also offers a variety of separate renderers for different clients written in different frameworks (Angular 2, React).



Example form

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas fermentum, sem in pharetra pellentesque, velit turpis volutpat ante, in pharetra metus odio a lectus. Nullam dapibus fermentum ipsum. Quisque porta.

Figure 3.5: form.io Form Builder Application

Advantages

- Has built-in form validation and other advanced features.
- Supports column layouts.
- Supports wizard creation.
- Various renderers are available for the plugin, namely for Angular 2 and React.

Disadvantages

- Depends on AngularJS (1.x) – restricts framework to use for the Form Creator interface frontend to AngularJS.
- The plugin is tightly coupled to form.io services, thus would require heavy modifications.
- There is no advanced documentation about how to customize the plugin.

3.2.4 fl-form-builder

The last open source MIT licensed plugin featured in this chapter called *fl-form-builder* [11] is compared to other open source and commercial solutions basic in terms of form field types and other features, such as form validation. This plugin comes with no renderer.

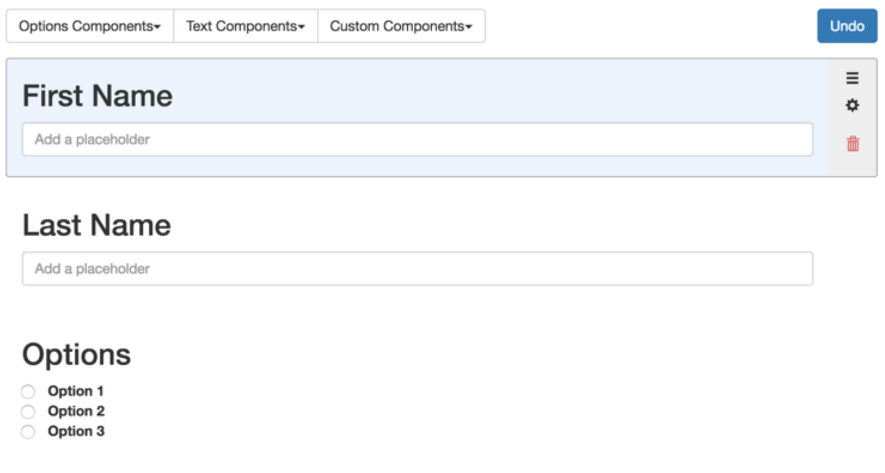


Figure 3.6: fl-form-builder

Disadvantages

- Does not support column layouts.
- Insufficient number of built-in components (most importantly no headings, no paragraphs and no buttons).
- Does not support localization.
- No form renderer provided.

3.3 Commercial Plugins

3.3.1 Form Designer

Paid tool for building web forms portrayed in Figure 3.7 which generates HTML, CSS and JavaScript code. Costs $17 for use in non-commercial projects or $85 otherwise. [14] Supports basic form controls and has form validation features. Compared to open source solutions mentioned section 3.2, this web form editor does not support drag-and-drop form editing.

Advantages

- Has built-in support for form templates.

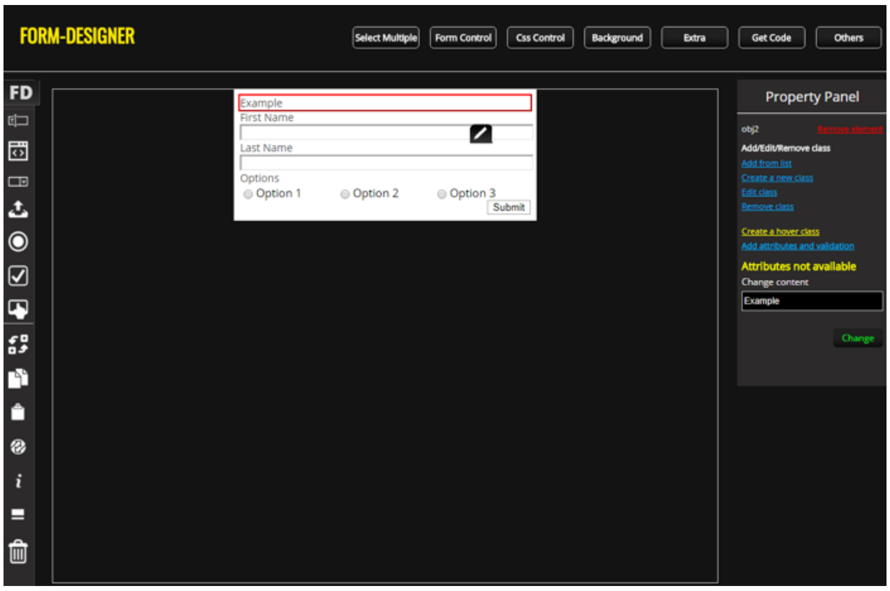


Figure 3.7: Form Designer

Disadvantages

- Does not support drag-and-drop form editing.
- Generates HTML code, which might limit portability of the forms only to web-based environments.

3.4 Conclusion

No plugin mentioned above is ideal for the needs of the application. *jQuery formBuilder* is close, though. It will, therefore, be used in the application as WYSIWYG form editor component.

The main reasons for choosing jQuery formBuilder are:

- It has useful documentation describing all the aspects of plugin's API.
- It can be easily adjusted and extended.
- It does not depend on any framework, so framework for the application can be selected freely.
- It has minimalistic separate form renderer.

Chapter 4

Technical Design

Technical design is an important aspect of application development which often determines technical quality, clarity, maintainability and many other aspects of the application for its whole lifetime. This chapter will state the most important technical design decisions while also justifying why were they used.

Section 4.1 will cover the most important design decisions, regarding the structure of the application in terms of layers, components and modules, and distribution of those among physical devices and runtime environments.

Section 4.2 will briefly introduce chosen implementation technologies to the reader.

Sections 4.3 and 4.4 will be more of documentational nature – they will capture what do modules and packages of the application exactly contain and how do client and server communicate.

Section 4.5 will conclude this chapter by stating software testing methods used when developing the application.

4.1 Software Architecture

4.1.1 Design Approach

In order to achieve user experience similar to a desktop application, *single-page application* (SPA) design approach will be preferred over more traditional approach – server-side rendering design.

The main difference between SPA and server-side rendering is that presentation layer of SPA is contained within the client, i.e. rendering of the views happens on the client, but the server still plays a significant role for the application as it will retrieve data from a database and send data to the SPA or process transactions for the SPA. [15, p. 7]

Among the most significant advantages of SPAs are:

- Reduction of round-trips between client side and server side, as view changes are processed on the client side which results in less distracting page reloads. [15, p. 7]
- Presentation layer contained within SPA is decoupled from other layers on the server. This also means that client or server can be updated separately. [15, p. 13]

4.1.2 Architectural Style

The Client and Server components of the application will use Multitier Business microarchitectures. Figure 4.1 illustrates tiers of the application as a whole.

Tier responsibilities:

Presentation Layer will present the data provided by *Client Service Layer* to the user. It will contain View and ViewModel.

Client Service Layer will contain Model and also fetch the data from *Service Layer* located on the server side.

Service Layer will expose a web service interface and will retrieve the data from *Data Layer*, or in some cases process transactions using artefacts located in *Business Layer*.

Business Layer will process any data requests more complex than simple CRUD operations, such as business process simulation, data model parsing. After decomposing resolved problems, it will delegate simple CRUD operations to *Data Layer*.

Data Layer will contain data model definitions and process CRUD data operations.

According to Martin Fowler [16], such architectural style offers better testability because each of the layers can be mocked. Last but not least, this architectural style allows to easily create another client for the application or to swap data source.

4.1.3 Components

UML component diagram describing component structure of the application can be found in Figure 4.2.

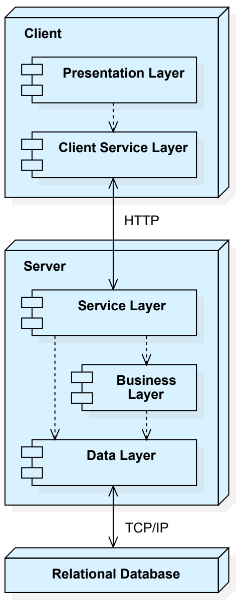


Figure 4.1: Component structure the application

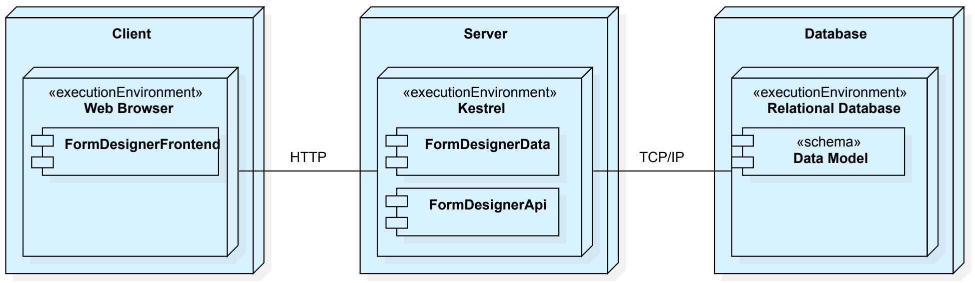


Figure 4.2: Component structure of the application

Client

Since SPA design approach moves presentation logic from the server side to client side and presentation logic will consume the data from the web service the server side will provide, the client component will consist of a single module named `FormDesignerFrontend`, containing *Presentation Layer* and *Client Service Layer* depicted in Figure 4.1.

Server

The server which will mainly process data will provide an interface for XHR calls from the client and will communicate with a DBMS which will provide data to it.

The server component will be divided into two modules:

- `FormDesignerData` will contain *Data Layer* and *Business Layer*.
- `FormDesignerApi` will contain *Service Layer*.

The division into such modules allows for easy reuse of `FormDesignerData` for different types of clients, such as a desktop or mobile client.

4.1.4 User Interface Patterns

Because the application will contain a considerable amount of code related to user interfaces, the Presentation Layer will take advantage of a few patterns known for improving code quality.

MVVM

Emmit Scott in his book [15, p. 24] briefly summarizes the MVVM pattern:

> *Model-View-ViewModel* (MVVM) was proposed by John Gossman in 2005 as a way to simplify and standardize the process of creating user interfaces [. . .]. It's another design pattern that emerged to try to organize the code associated with the UI into something sensible and manageable, while still keeping the various components of the process separate.
>
> - *Model* – The model typically contains data, business logic, and validation logic. [. . .] The model is never concerned with how data is presented.
> - *View* – The view is what the user sees and interacts with. Its a visual representation of the models data. [. . .]
> - *ViewModel* – The ViewModel is a model or representation of the view in code, in addition to being the middleman between the model and the view. Anything needed to define and manage the view is contained within the ViewModel. This includes data properties as well as presentation logic. Each data point in the model that needs to be reflected in the view is mapped to a matching property in the ViewModel. [. . .] It's aware of changes in both the view and the model and keeps the two in sync.

Presentational and Container Components

Presentational and Container Components, also known as *Smart and Dumb Components* or *Fat and Skinny Components* is a pattern which resolves separation of concerns among code written in frameworks utilizing *UI components*, including but not limited to Angular, Vue.js and React – today's major frameworks for SPA development.

Presentational Components consume data and render something based on received data. They typically do not depend on the rest of the application, thus are reusable.

Container Components specify the behaviour of a part of the application and provide data fetched from data layer to their descendant presentational components.

Benefits of the approach include more understandable code and better reusability of components. [17]

4.2 Technologies

4.2.1 Client

Because the application will be an SPA with jobs consisting of mainly presentating or manipulating the data, a template rendering solution with data binding capabilities will be preferred over manual DOM manipulation. Such solution will greatly decrease application code complexity and increase testability [18, p. 34].

Angular 5

Angular 5 is a cross-platform MVW framework for SPA development, providing the developers with the essentials for SPA development, including REST client, template rendering system, data binding functionalities, and many more features. TypeScript is preferred language to write Angular 5 applications with. [19]

CoreUI

CoreUI is an Open Source Bootstrap 4 based administration interface template featuring useful add-ons, transparent code and file structure. It offers versions for all of today's popular JavaScript frameworks. [20] The Angular 5 version will be used for the application.

TypeScript

"TypeScript is a typed superset of JavaScript that compiles to plain JavaScript" compliant with ECMAScript 3 specification.

TypeScript allows web developers to:

- Use highly productive development tools, such as refactoring and static analysis.
- Use interfaces and decorators to deliver higher quality object-oriented code.
- Use the latest ECMAScript specification features, such as `async/await` syntax, which greatly increases asynchronous code readability. [21]

MxGraph

"mxGraph is a JavaScript diagramming library that enables interactive graph and charting applications to be quickly created that run natively in any major browser that is supported by its vendor." [22]

4.2.2 Server

ASP.NET Core 2.0

"ASP.NET Core is a web development platform built on .NET Core, which is a cross-platform version of the .NET Framework without the Windows-specific application programming interfaces (APIs)." Unlike its predecessors .NET and ASP.NET, the new .NET Core and ASP.NET Core are fully open source. [23, p. 5]

Entity Framework Core

Entity Framework Core is an ORM framework designed specifically for use in .NET Core applications. It abstracts tables, columns and rows from relational databases to plain C# objects. [23, p. 208]

4.3 Module Structure

4.3.1 Client

Client module utilizes *NgModules* which are provided by Angular 5 out of the box. Each NgModule is reusable as a whole, provided that its dependencies are satisfied.

FormDesignerFrontend

Figures 4.3 and 4.4 illustrate NgModules of `FormDesignerFrontend` module and their dependencies.

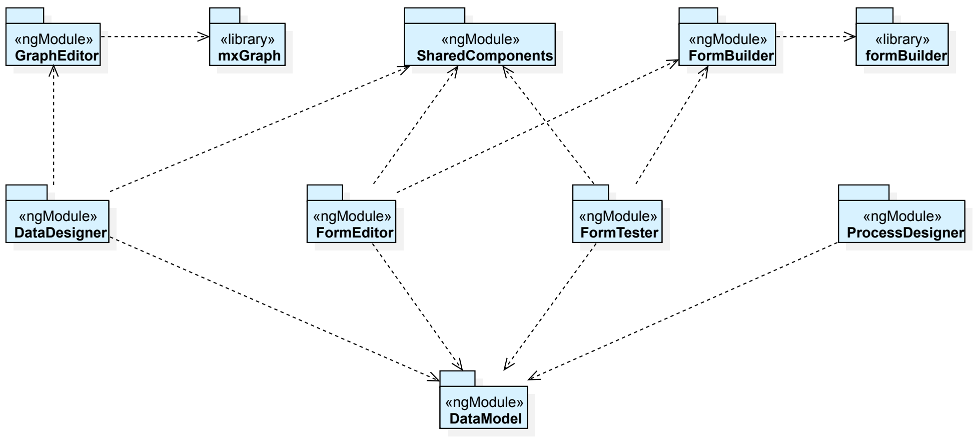


Figure 4.3: NgModules of FormDesignerFrontend and their dependencies

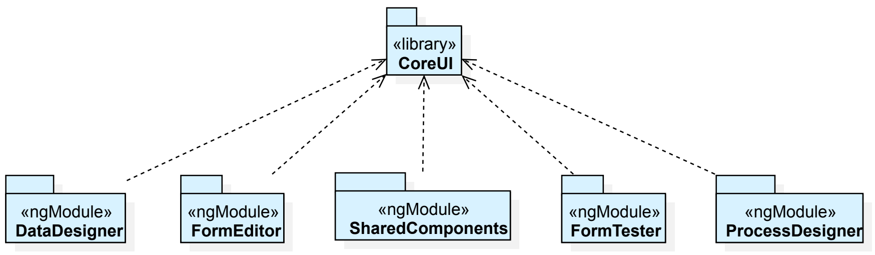


Figure 4.4: NgModules of FormDesignerFrontend depending on CoreUI

- `DataDesigner` module contains the *Data Designer* section of the application.
- `DataModel` module contains data model definitions, DTOs and HTTP client classes for invoking the web service endpoints.
- `FormBuilder` module contains integration of jQuery formBuilder library for the Angular application.
- `FormEditor` module contains the *Form Editor* section of the application.
- `FormTester` module contains the *Form Tester* section of the application.
- `GraphEditor` module contains integration of MxGraph library for the Angular application.
- `ProcessDesigner` module contains the *Process Designer* section of the application.
- `SharedComponents` module contains Angular presentational components which are shared by multiple other modules.

4.3.2 Server

Server modules are further divided into namespaces to simplify navigation across their respective source code files. Each namespace corresponds to a directory in given project.

FormDesignerData

- `DataAccess` namespace contains DAOs for executing CRUD operations on the data in the relational database.
- `DataControl` namespace contains classes which manipulate the data in relational database in non-CRUD matters.
- `DataTransfer` namespace contains DTOs for necessary operations.
- `FormProcessing` namespace contains classes for form-to-model binding – i.e. populating form field values, mapping form fields to the data model and handlers for submitting the forms.
- `Model` namespace contains data model entity classes, their associated interfaces and database context for querying the database directly.
- `ModelManagement` namespace contains classes for managing Fact Model instances – i.e. instantiating entities, loading existing entities, etc.
- `ModelProcessing` namespace contains data model parser, command classes and related interfaces.

FormDesignerApi

- `Controllers` namespace contains definitions of RESTful web services.

4.4 Client-Server Communication

The server will expose a RESTful web service API for the client.

RESTful web service style was chosen in order to reduce client's direct coupling to remote procedures on the server by using standard HTTP methods for CRUD operations [24, p. 38]. The resources were designed in order to satisfy use-cases of the application and with data bandwidth conservation in mind.

4.4.1 Resources

/api/assignments

This resource represents an assignment between `ProcessKind` and `TransactionKind`.

POST method creates an assignment.

DELETE method detaches specified `ProcessKind` from specified `TransactionKind`.

/api/dataModel

This resource represents specification of Fact Model data model as a whole. Since the application supports only one domain at a time, only one data model is supported by the API, so this resource does not take advantages of routing by ID.

GET method retrieves current state of the model.

POST method updates the model according to request body content.

/api/entityKinds

This resource represents a collection of all `EntityKind` entities present in Fact Model data model.

GET method retrieves the collection.

/api/formFields

This resource represents a set of Model Fields to be used in the interface of Form Creator. The Model Fields themselves can be created, updated or deleted using the POST method of the `/api/dataModel` resource mentioned earlier.

GET method retrieves the set.

/api/formKinds

This resource represents a collection of `FormKind` entitities.

GET method retrieves the collection.

POST method adds a `FormKind` to the collection of `FormKind` entites.

/api/formKinds/{id}

This resource represents a single entity of collection of `FormKind` entitities.

PUT method updates the entity.

DELETE method removes the entity from the collection.

/api/forms/{id}

This resource represents a single `Form` entity.

GET method retrieves given `Form` entity.

POST method updates the `Form` entity but has nonstandard behaviour, as it is intended for submitting the form – thus request body will contain only contents of the formData attribute of the Form entity with given ID.

/api/processes

This resource represents a collection of entities of type `Process`.

GET method retrieves the collection.

POST method adds a `Process` entity to the collection.

/api/processes/{id}

This resource represents a single `Process` entity as a member of the collection of `Process` entities described in the previous section.

GET method retrieves the collection.

/api/processKinds

This resource represents a collection of entities of type `ProcessKind`.

GET method retrieves the collection.

POST method adds a `ProcessKind` entity to the collection.

/api/processKinds/{id}

This resource represents an entity from the collection of `ProcessKind` entities.

GET method retrieves the entity.

/api/transactionKinds

This resource represents a collection of entities of type `TransactionKind`.

GET method retrieves the collection.

POST method adds a `TransactionKind` entity to the collection.

4.4.2 Status Codes and Error Handling

The API handles common situations with standardized HTTP status codes. However, when an internal exception in `FormDesignerData` occurs, the server will return a response with status code 500 and a custom envelope containing exception message as response body illustrated in Listing 4.1.

```
{
    "message": "<exception message>"
}
```

Listing 4.1: Error envelope structure

4.5 Testing

4.5.1 Unit Testing

Since the Data Layer uses a third-party well-tested Entity Framework Core library mentioned in section 4.2.2 and Service Layer just delegates all its calls to either Business Layer or Data Layer directly, only *Business Layer* with its non-trivial responsibilities will be covered by unit tests.

Classes and their responsibilities covered by the unit tests:

- `FormFilter`, `FormKindFilter`, `FormSubmitter` and `ModelManager` – form-to-model binding will be tested.
- `ProcessTransactionAssigner` – assigning transactions to business processes will be tested.
- `XmlModelParser` – model parsing from XML generated by the *Data Designer* will be tested.

4.5.2 Testing by Developer

The modules, especially the client component of the application, i.e. `FormDesignerFrontend`, will be tested using this testing method every time after implementing a new functionality.

Generally, the steps of this testing method will be:

- visual examination of the user interface,
- testing user interaction with parts of the application that are both directly and indirectly affected by the new functionality using scenarios defined by the developer.

The scenarios mentioned above will be designed according to tested user interface structure.

CHAPTER 5

Proof of Concept

The last chapter of this thesis will demonstrate capabilities of the application on a real use-case – Rent-A-Car case from the book *The Essence of Organisation.*

Section 5.1 will describe the case and depict the Fact Models vital for the demonstration.

Section 5.2 will describe the whole process of the demonstration – from redrawing the Fact Model of the domain to simulating business processes using user-defined forms.

5.1 Rent-A-Car Case Study

Alicia Perinforma in the book *The Essence of Organisation* describes the case *DEMO Form Designer* will demonstrate upon:

> Rent-A-Car (or RAC for short) is a company that rents cars to persons, both private ones and representatives of legal bodies, like companies. It was founded by the twin brothers Janno and Ties back in the eighties. They started to hire out their own (two) cars, and they were among the first companies that allowed cars to be dropped off in a different location than where they were picked up. To this end, Janno and Ties had made agreements with students in several cities. For a small amount of money, a student would await the arrival of a rented car, e.g. at an airport, and drive it back to the office of RAC, after which the student would go home by public transport. Currently, RAC operates from over fifty geographically dispersed branches in Europe. Many cities have a branch, some even several, and there are branches located near all airports. One of the branches is the original office where Janno and Ties started and where both are still around. Being mechanical engineer by

education, they have kept loving to drive and maintain cars, even since they are the managing directors of a million euro company.

The head of the front office of the home branch is Chiara. There are two more desk officers working in this department. Customer orders are placed through several channels: walk-in, telephone, fax, and e-mail. Walk-in customers are usually people who want to rent a car immediately. Through the other channels one makes in general advance reservations. These can be made up to 200 days in advance, called the rental horizon. In all cases, an electronic rental form is filled out by one of the desk officers, as input to RACIS (RAC Information System). The next groups of data must be provided:

RENTAL: identification number (automatically generated), start date, end date, issuing branch, return branch, car group.
RENTER: identification (passport or driving license), first name, last name, address, date of birth, place of birth.
DRIVER: driving license (also for identification), first name, last name.
FINANCIAL: rental rate per day (basically determined by the car group).

Although it is the task of the desk officers to take the orders for renting a car, Janno or Ties may drop by and help a walk-in customer or pick up the telephone. Chiara does not really like these distortions but she thinks she cannot do much about it. The problem with these spontaneous actions of Janno and Ties is that they often forget to record things properly, resulting in misunderstandings and even disputes with customers afterwards. Next, they sometimes act against the rules, for example by promising a car for a lower rate than the listed one.

The cars of RAC are divided in car groups. A car group may contain several types (brands and models). The common feature of the cars in a group is that they have the same rental rate per day. The board of directors, i.e. Janno and Ties, decide which brands and models belong to which group as well as what the rental rate is for every group. Normally they do this once a year.

For a walk-in customer the starting day is usually the same day as on which the contract is established. Advance reservations have some future day as the starting day. RAC applies a maximum rental period (currently 10 days).

After the renter has signed the contract, the rental is concluded by the desk officer (Note: the signing by the renter counts as promising to pay the rental charge, which is the contracted duration times the daily rental rate. Because the rental may be an advance reservation, the payment may be delayed until the starting day).

On the starting day, the driver can pick up a car at the distribution department, located at the backside of the building, on presentation of a copy of the contract. There are three employees working in this department: Mik, Ferre, and Carlo, but not all of them are always present, as we will see. As soon as a driver shows up, one of them checks whether there is a car available of the contracted group. If there is one, he will allocate the car to the rental contract and sign the contract as being picked up. If there is no car available of the contracted group, he will upgrade the contract and select a car from the next higher car group. The driver will get this upgraded car, for the price of the contracted group.

After the car of a rental has been dropped off at some branch, the possibly incurred fines have to be paid. There may be a penalty charge for returning the car after the contracted end date. It amounts to the number of extra days times the late return penalty rate. In addition, the car may have been dropped off at another branch than the contracted return branch. In that case a location penalty charge has to be paid. This amounts to the distance between the actual and the contracted drop off branch times the penalty rate per kilometer.

The distribution department is also responsible for transporting cars between branches because cars may be dropped off at other locations, as we have seen. To this end, Mik schedules every morning the transportations that have to be performed that day. The transportations are carried out by all three of them, so also by Ferre and Carlo. That is why often some of them are away from the office.

5.1.1 Fact Model

Along with case description listed above, Alicia Perinforma also provides Object Fact Diagrams of the case study:

Primary processes

Fact model of primary processes of the Rent-A-Car case study is present in Figures 5.1 and 5.2.

Secondary processes

Fact model of secondary processes of the Rent-A-Car case study is present in Figure 5.3.

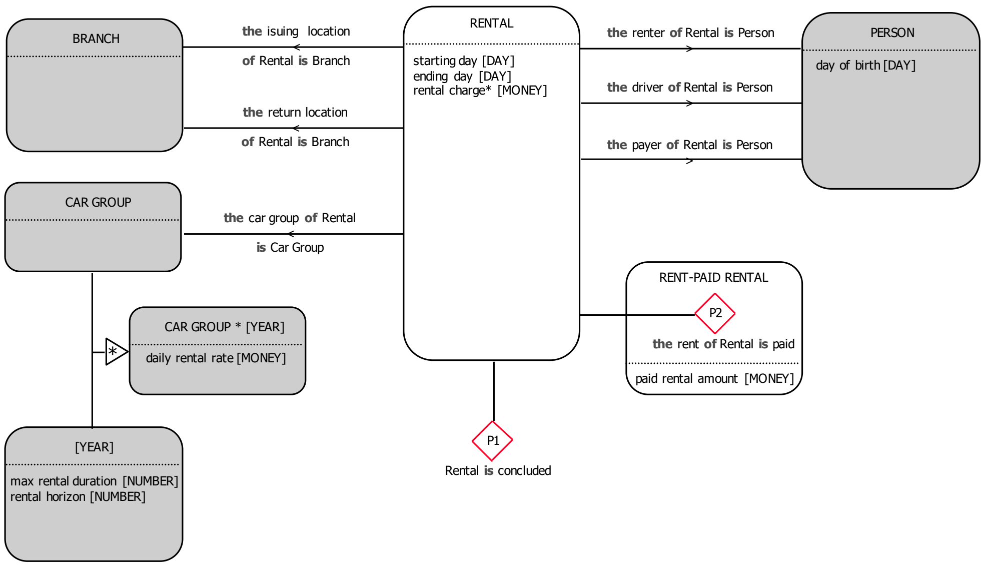

BRANCH
the isuing location of Rental is Branch
the return location of Rental is Branch
RENTAL
starting day [DAY]
ending day [DAY]
rental charge* [MONEY]
the renter of Rental is Person
the driver of Rental is Person
the payer of Rental is Person
PERSON
day of birth [DAY]
CAR GROUP
the car group of Rental is Car Group
CAR GROUP * [YEAR]
daily rental rate [MONEY]
[YEAR]
max rental duration [NUMBER]
rental horizon [NUMBER]
RENT-PAID RENTAL
P2
the rent of Rental is paid
paid rental amount [MONEY]
P1
Rental is concluded

Figure 5.1: Primary processes of Rent-A-Car, part one [1]

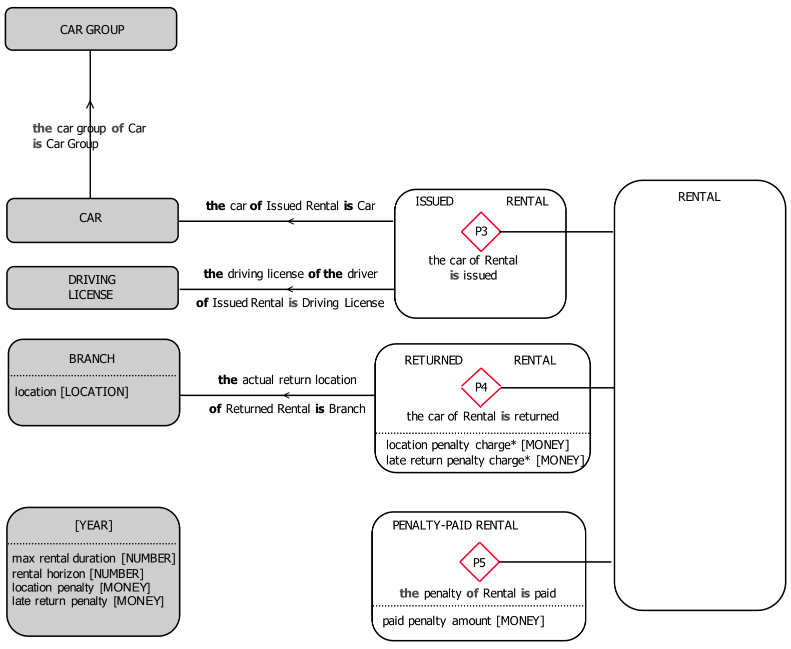

CAR GROUP
the car group of Car is Car Group
CAR
the car of Issued Rental is Car
ISSUED RENTAL
P3
the car of Rental is issued
DRIVING LICENSE
the driving license of the driver of Issued Rental is Driving License
RENTAL
BRANCH
location [LOCATION]
the actual return location of Returned Rental is Branch
RETURNED RENTAL
P4
the car of Rental is returned
location penalty charge* [MONEY]
late return penalty charge* [MONEY]
[YEAR]
max rental duration [NUMBER]
rental horizon [NUMBER]
location penalty [MONEY]
late return penalty [MONEY]
PENALTY-PAID RENTAL
P5
the penalty of Rental is paid
paid penalty amount [MONEY]

Figure 5.2: Primary processes of Rent-A-Car, part two [1]

5.1.2 Action Model

Due to the considerable length of the action rules, only parts of action rules concerning the case study will be used later when demonstrating method-

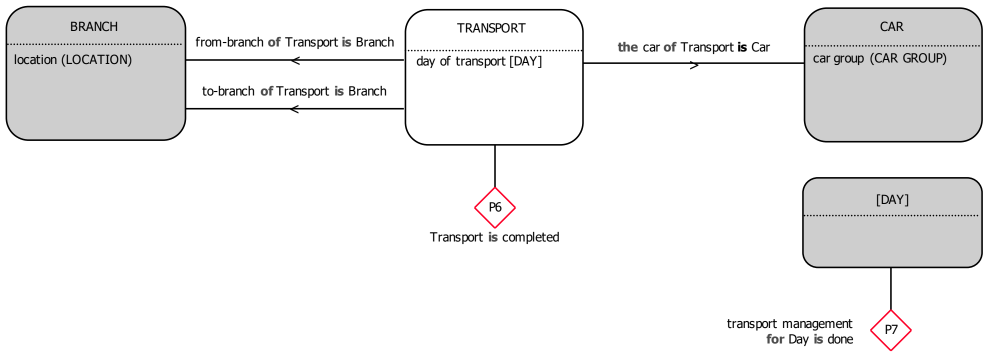

BRANCH
location (LOCATION)
from-branch of Transport is Branch
to-branch of Transport is Branch
TRANSPORT
day of transport [DAY]
the car of Transport is Car
CAR
car group (CAR GROUP)
P6
Transport is completed
[DAY]
transport management for Day is done
P7

Figure 5.3: Secondary processes of Rent-A-Car [1]

ologies of determining the structure of the forms and assigning the forms to transactions and C-acts.

5.2 Demonstration

5.2.1 Drawing the Fact Model

Since the *Data Designer* section of the application is all graphical and straightforward to use, all the parts of supplied OFD were just redrawn to the canvas to create a single diagram.

Please note that in order to make the demonstration more sensible later, some attributes have been added to the OFD, namely:

- *location* and *name* attributes for BRANCH entity type,
- *brand*, *model* and *year* attributes for CAR entity type,
- *name* attribute of CAR GROUP entity type,
- *license number* attribute of DRIVING LICENSE entity type,
- *name* attribute of PERSON entity type.

The result is depicted in Figure 5.4.

5.2.2 Declaring Business Processes

Discovered from the construction model of the domain in Figure 5.5, there are three business processes:

- *Rental Order* which consists of T1 and T2;
- *Car Manipulation* which consists of T3, T4 and T5;

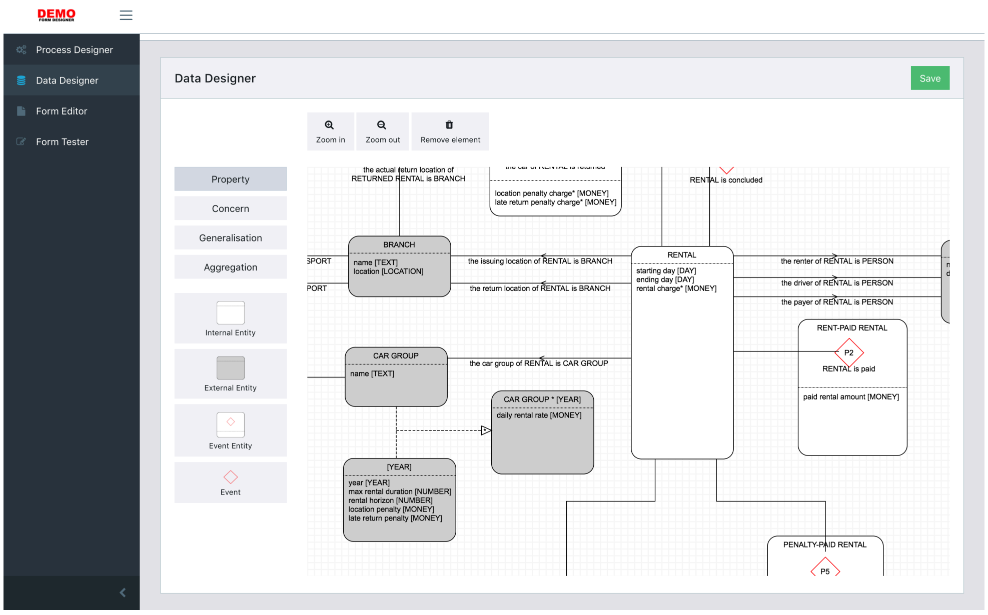


Figure 5.4: OFD drawn in *Data Designer*

- *Transport Management* which consists of T6 and T7.

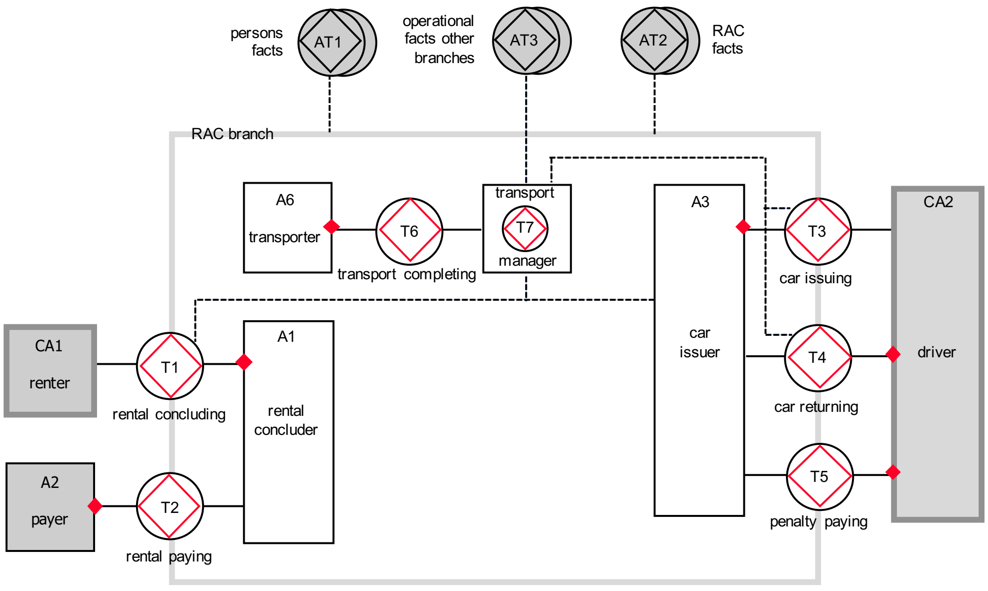


Figure 5.5: Construction Model of Rent-A-Car [1]

The processes mentioned above were declared in the application and transactions were assigned to them afterwards, as seen in Figure 5.6.

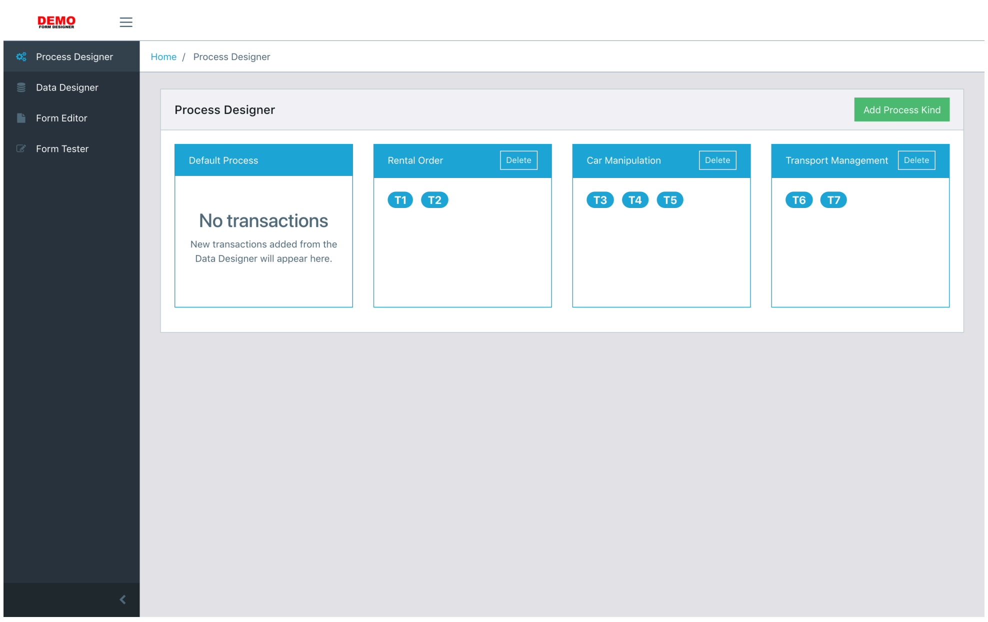


Figure 5.6: Declared business processes in *Process Designer*

5.2.3 Designing Web Forms

Determining Form Structure

Determining form structure is understood to mean *defining which form fields will be put to given form* based on action rule contents.

There is an *event part* of a sample Rent-A-Car action rule in Figure 5.7 for demonstrating the methodology for determining form structure.

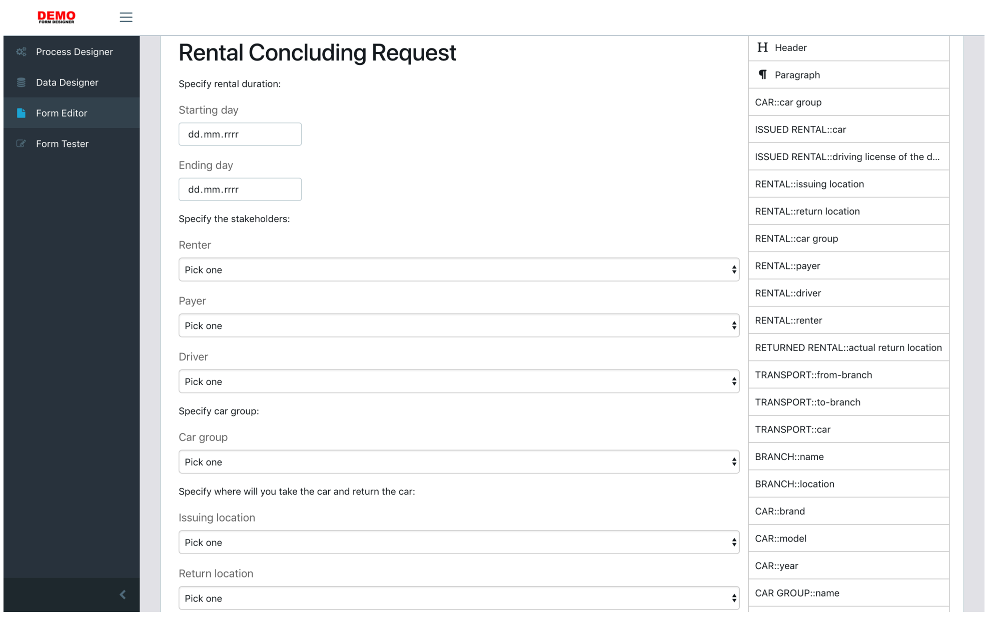
when rental concluding **for** new Rental **is** requested (T1/rq)
with **the** starting day **of** Rental **is some** day
the ending day **of** Rental **is some** day
the renter **of** Rental **is some** person
the payer **of** Rental **is some** person
the driver **of** Rental **is some** person
the car group **of** Rental **is some** car group
the issuing location **of** Rental **is some** branch
the return location **of** Rental **is some** branch

Figure 5.7: Sample Rent-A-Car action rule (event part) [1]

The form fields will be put to the form according to the fact types within the *with* clause of the event part of the action rule.

Only *original fact types* (i.e. non-derived fact types) will be included in the form, since in Object Fact Diagram itself, there are no derived fact types.

Given the rules for determining form structure, a sample form for has been specified in Figure 5.8.



Rental Concluding Request

Specify rental duration:

Starting day

dd.mm.rrrr

Ending day

dd.mm.rrrr

Specify the stakeholders:

Renter

Pick one

Payer

Pick one

Driver

Pick one

Specify car group:

Car group

Pick one

Specify where will you take the car and return the car:

Issuing location

Pick one

Return location

Pick one

H Header

¶ Paragraph

CAR::car group

ISSUED RENTAL::car

ISSUED RENTAL::driving license of the d...

RENTAL::issuing location

RENTAL::return location

RENTAL::car group

RENTAL::payer

RENTAL::driver

RENTAL::renter

RETURNED RENTAL::actual return location

TRANSPORT::from-branch

TRANSPORT::to-branch

TRANSPORT::car

BRANCH::name

BRANCH::location

CAR::brand

CAR::model

CAR::year

CAR GROUP::name

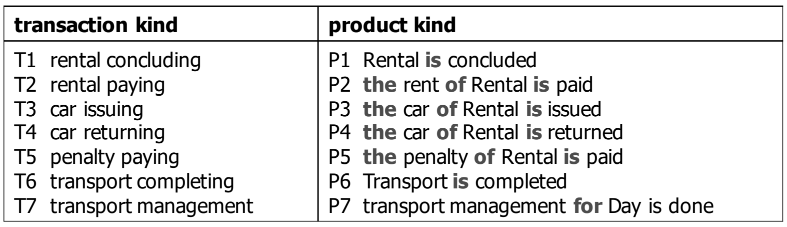
Figure 5.8: Form designed using *Form Creator* based on the action rule

Placing model fields to the form in the application, however, implies that the form will need to be connected to a *RENTAL* entity of the Fact Model.

Assigning the Forms to a Transaction Kind and C-act

When saving the form, it will be assigned to:

- transaction kind specified in the *when* clause of the *event part* of the action rule – *transaction kind ID* can be found in the TPT of the domain, TPT of the case study is located in Figure 5.9;
- C-act corresponding to a C-fact in the *when* clause of the *event part* of the action rule.



transaction kind	**product kind**
T1 rental concluding	P1 Rental **is** concluded
T2 rental paying	P2 **the** rent **of** Rental **is** paid
T3 car issuing	P3 **the** car **of** Rental **is** issued
T4 car returning	P4 **the** car **of** Rental **is** returned
T5 penalty paying	P5 **the** penalty **of** Rental **is** paid
T6 transport completing	P6 Transport **is** completed
T7 transport management	P7 transport management **for** Day is done

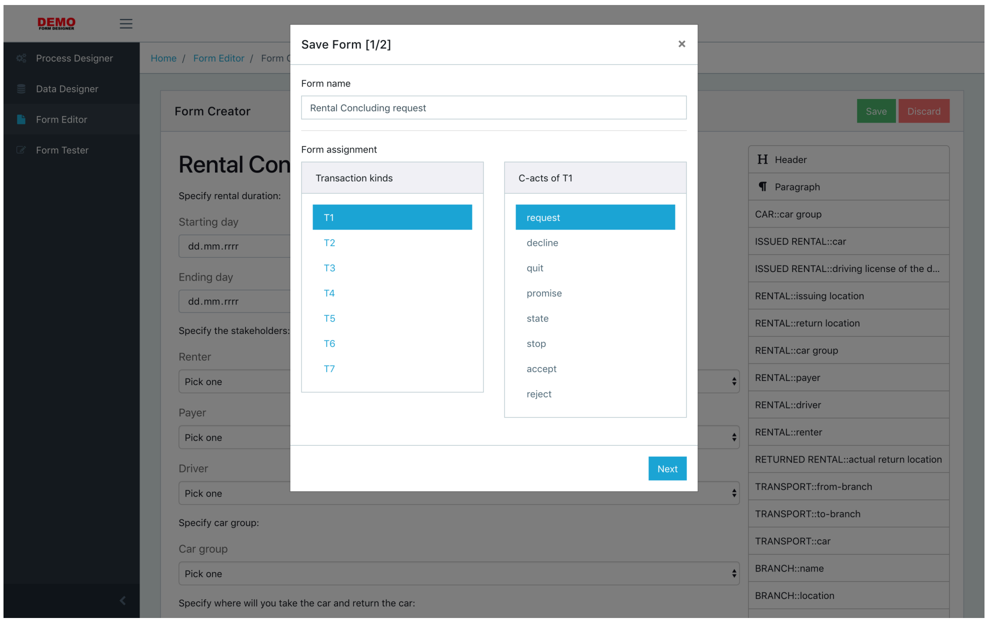
Figure 5.9: TPT of Rent-A-Car domain [1]

Table 5.1 can be used to find a C-act corresponding to given C-fact.

C-fact	Corresponding C-act
requested	request
declined	decline
quit	quit
promised	promise
stated	state
stopped	stop
accepted	accept
rejected	reject

Table 5.1: C-facts and their corresponding C-acts

With the rules formulated above, given an action rule in Figure 5.7, the form will be assigned to transaction kind *T1* and C-act *request*. The assignment interface in the application with the assignment setup is illustrated in figure 5.10.



Save Form [1/2]

Form name: Rental Concluding request

Form assignment

Transaction kinds	C-acts of T1
T1	request
T2	decline
T3	quit
T4	promise
T5	state
T6	stop
T7	accept
	reject

Rental Con

Specify rental duration:

Starting day: dd.mm.rrrr

Ending day: dd.mm.rrrr

Specify the stakeholders:

Renter: Pick one

Payer: Pick one

Driver: Pick one

Specify car group:

Car group: Pick one

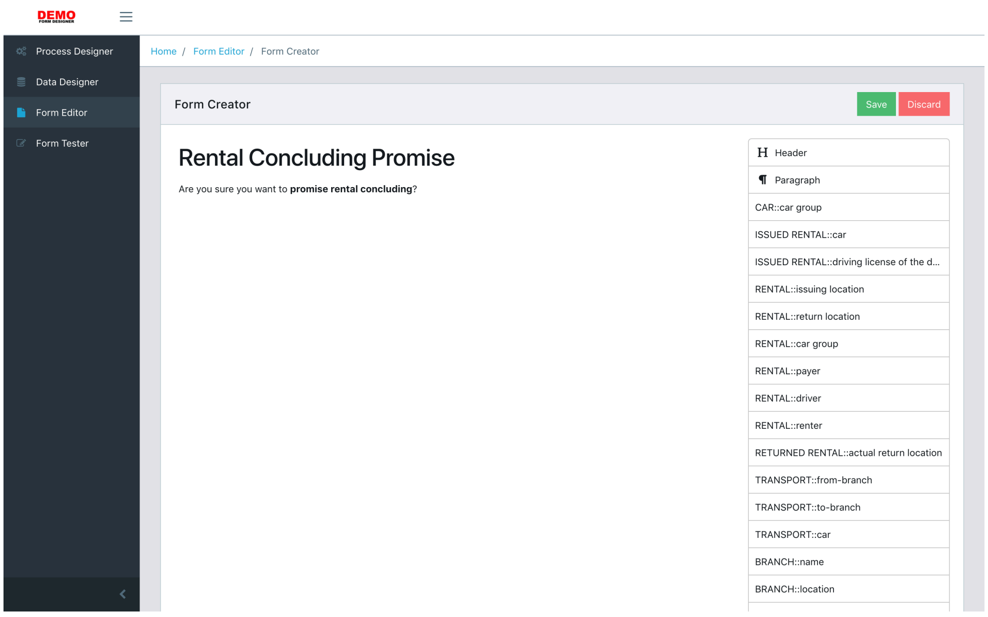
Specify where will you take the car and return the car:

CAR::car group
ISSUED RENTAL::car
ISSUED RENTAL::driving license of the d...
RENTAL::issuing location
RENTAL::return location
RENTAL::car group
RENTAL::payer
RENTAL::driver
RENTAL::renter
RETURNED RENTAL::actual return location
TRANSPORT::from-branch
TRANSPORT::to-branch
TRANSPORT::car
BRANCH::name
BRANCH::location

Figure 5.10: Form assigned to T1/rq using *Form Creator*

Generating Placeholder Forms

Because a great number of Rent-A-Car action rules lack the *with* clause within their event part and all transactions must have all their C-acts assigned to a form for business process simulation to work properly, a simple form generator has been created which generates forms with a question dialogue similar to the one in Figure 5.11.



Rental Concluding Promise

Are you sure you want to **promise rental concluding**?

H Header
¶ Paragraph
CAR::car group
ISSUED RENTAL::car
ISSUED RENTAL::driving license of the d...
RENTAL::issuing location
RENTAL::return location
RENTAL::car group
RENTAL::payer
RENTAL::driver
RENTAL::renter
RETURNED RENTAL::actual return location
TRANSPORT::from-branch
TRANSPORT::to-branch
TRANSPORT::car
BRANCH::name
BRANCH::location

Figure 5.11: Generated form example

The generator is located in file `/src/tools/form-generator.js` on enclosed CD.

The user is required to fill the `transactionKinds` array in with information about transaction kinds and `skipCActs` array with information about which pairs of transaction kind and C-act to skip as described in Listing 5.1.

```
const transactionKinds = [
    { // T1
        name: "rental concluding",
        dbId: 1
    }
];

const skipCActs = [
    { // T1/rq
        transactionKind: transactionKinds[0],
        cAct: cActs[0]
    }
];
```

Listing 5.1: Form generator setup example

The `transactionKinds` array contains objects with the following attributes:

name represents transaction kind name.

dbId represents numeric transaction kind ID from the database (transaction kinds are located in the `TransactionKinds` table).

The `skipCActs` array contains objects with the following attributes:

transactionKind represents reference value referring to an object of the `transactionKinds` array.

cAct represents reference value referring to an object of the `cActs` array located in the generator.

After both arrays being filled in with correct information, the script can be run using *Node.js*.

After running the script, SQL INSERT statements for all transaction kind – C-act pairs will be generated to be executed using database client of choice.

5.2.4 Simulating Business Processes

Populating External Entities

Prior to the simulation itself, the external, out-of-scope entities should be added to the database in order to demonstrate how the forms work with out-of-scope entities. To simplify the process, the classes located in `FormDesignerData` module expose their public APIs which accomplish the task.

A simple stand-alone tool called `FormDesignerEntityPopulator` built around the APIs mentioned above located on enclosed CD in `/src/application` directory is able to populate external entities in the database.

User is encouraged to change the lines designated by comments according to his or her needs. An example usage of the APIs for entity populating can be seen in Listing 5.2.

```
// mgr is an object of ModelManager type
var carGroupKind = mgr.GetEntityKind("CAR GROUP");
var carKind = mgr.GetEntityKind("CAR");

var g1 = carGroupKind.Instantiate();
g1.SetAttributeValue("name", "SUVs");

var c1 = carKind.Instantiate();
c1.SetAttributeValue("brand", "BMW");
c1.SetAttributeValue("model", "X5");
c1.SetAttributeValue("year", "2010");
```

```
c1.AddProperty("car group", g1);

// ctx is an object of DataContext type
ctx.Entities.Add(c1);
ctx.Entities.Add(g1);
```

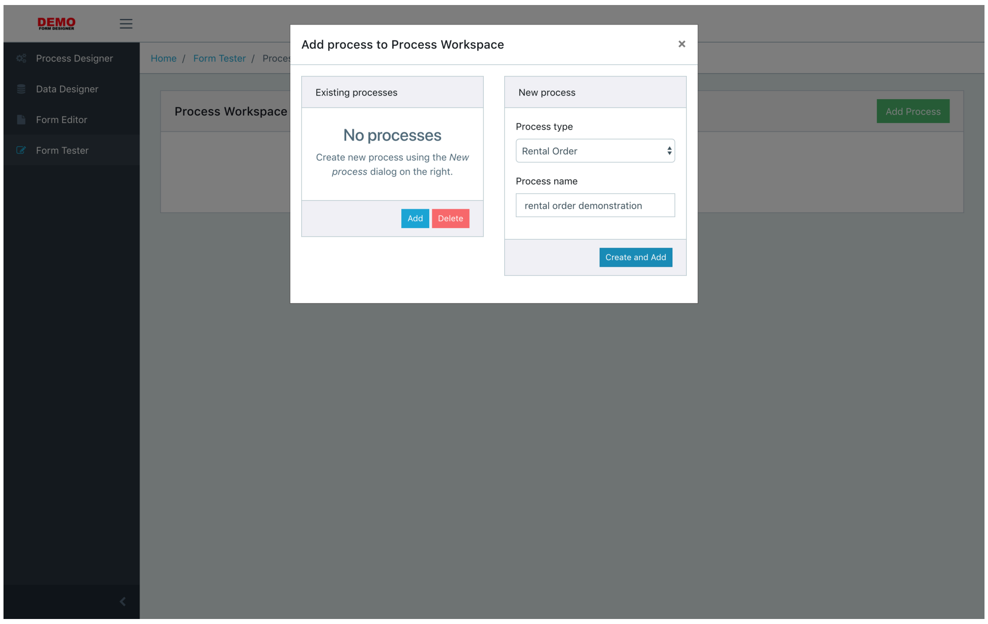
Listing 5.2: Entity populating APIs usage example

Usage of the APIs is rather simple:

1. Fetch entity types using `ModelManager` API.
2. Instantiate the entity types – this creates entities from them.
3. Set entity attributes, eventually properties.
4. Save the entities to the database using `DataContext` API.

Instantiating Business Processes

Rental Order business process will be demonstrated, so it will need to be instantiated first. The instantiation is done using the dialogue depicted in Figure 5.12.



Add process to Process Workspace

Existing processes

No processes

Create new process using the *New process* dialog on the right.

New process

Process type: Rental Order

Process name: rental order demonstration

Figure 5.12: Process instantiation interface

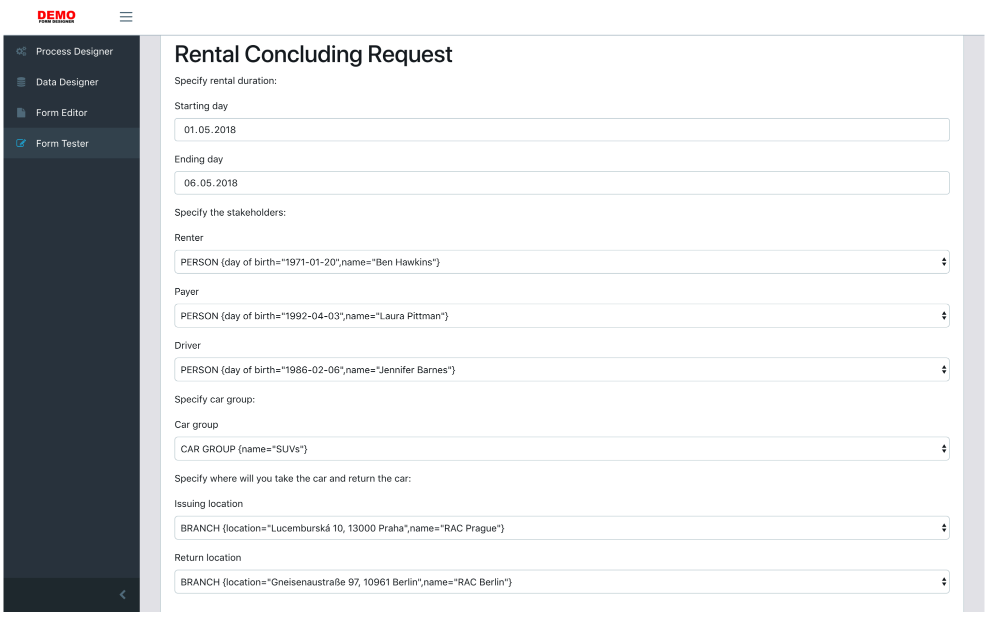
Simulating Business Processes

Simulated actors will now act in compliance with the action rules of the domain. Actors will have to evaluate the assessments in the *assess part* manually and respond to the actions according to the *response part* by filling out appropriate forms.

Happy path of the *Rental Order* business process declared in section 5.2.2 will now be simulated:

1. **Requesting T1 (rental concluding):**

   Renter requested the transaction by filling out the form in Figure 5.13.



Rental Concluding Request

Specify rental duration:

Starting day

01.05.2018

Ending day

06.05.2018

Specify the stakeholders:

Renter

PERSON {day of birth="1971-01-20",name="Ben Hawkins"}

Payer

PERSON {day of birth="1992-04-03",name="Laura Pittman"}

Driver

PERSON {day of birth="1986-02-06",name="Jennifer Barnes"}

Specify car group:

Car group

CAR GROUP {name="SUVs"}

Specify where will you take the car and return the car:

Issuing location

BRANCH {location="Lucemburská 10, 13000 Praha",name="RAC Prague"}

Return location

BRANCH {location="Gneisenaustraße 97, 10961 Berlin",name="RAC Berlin"}

Figure 5.13: Requesting *rental order* transaction

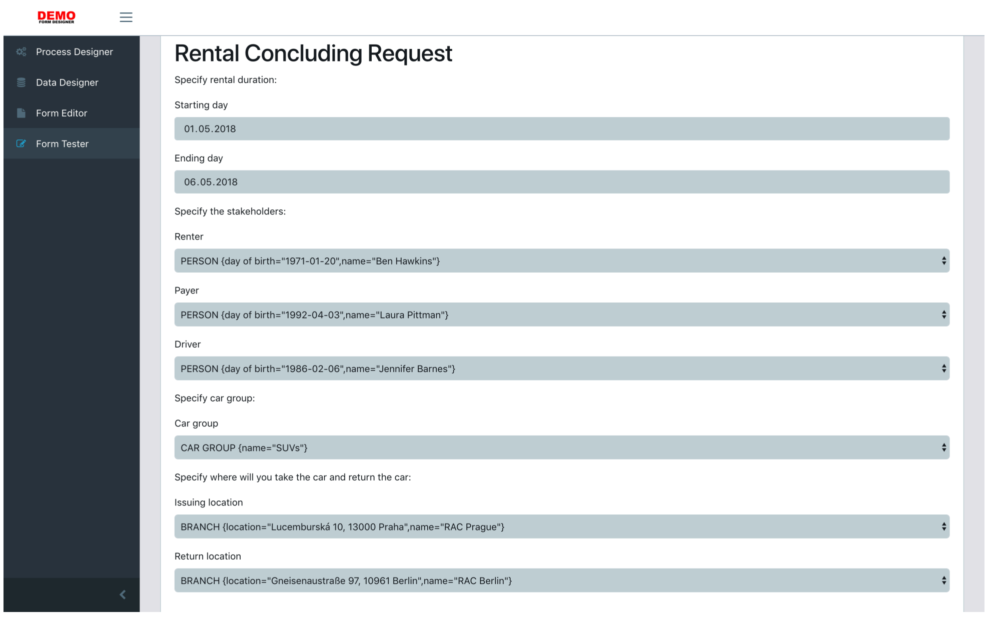
   Rental concluder will receive the request, review the C-fact as seen in Figure 5.14 of it and will assess it in compliance with the action rule in Figure 5.15. Let's assume that the assessment is complied with. Thus rental concluder will request rental payment as the next step.

2. **Requesting T2 (rental payment):**

   Rental concluder requested the rental payment with payment amount as seen in Figure 5.16.

3. **Promising T2 (rental payment):**

   Payer promised the payment using a generated form as illustrated in Figure 5.17.



Rental Concluding Request

Specify rental duration:

Starting day: 01.05.2018

Ending day: 06.05.2018

Specify the stakeholders:

Renter: PERSON {day of birth="1971-01-20",name="Ben Hawkins"}

Payer: PERSON {day of birth="1992-04-03",name="Laura Pittman"}

Driver: PERSON {day of birth="1986-02-06",name="Jennifer Barnes"}

Specify car group:

Car group: CAR GROUP {name="SUVs"}

Specify where will you take the car and return the car:

Issuing location: BRANCH {location="Lucemburská 10, 13000 Praha",name="RAC Prague"}

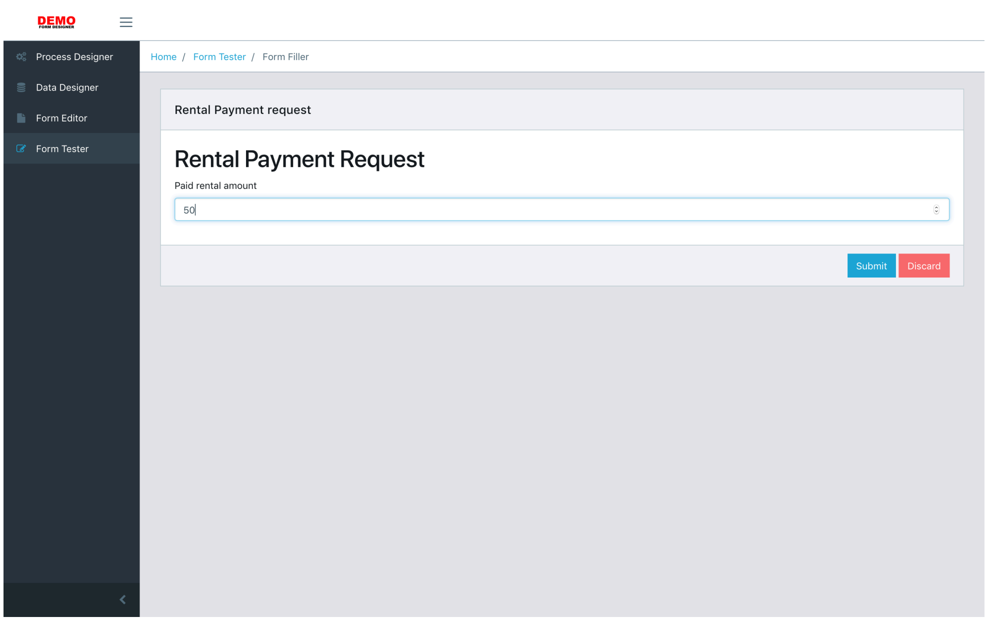
Return location: BRANCH {location="Gneisenaustraße 97, 10961 Berlin",name="RAC Berlin"}

Figure 5.14: Reviewing T1/rq C-fact

assess	*justice:*	the performer of the request is the renter of Rental;	
		the addressee of the request is a rental concluder;	
	sincerity:	< no specific condition >	
	truth:	the starting day of Rental is in the Rental Horizon of the year of the starting day of Rental;	
		the ending day of Rental is in the Rental Horizon of the year of the ending day of Rental;	
		the ending day of Rental is equal to or greater than the starting day of Rental;	
		the duration of Rental is less than or equal to the max rental duration in the year of the starting day of Rental;	
		the number of cars in the car group of Rental on every day in the rental period of Rental is greater than zero	
if	*complying with the assessment is considered justifiable*		
then	request	rental paying for Rental	[T2/rq]
	with	the addressee of the request is the payer of Rental;	
		the requested production day of rental paying for Rental is less than or equal to the starting day of Rental;	
		the requested paid rental amount of Rental is equal to the rental charge of Rental;	
else	decline	rental concluding for Rental	[T1/dc]
	with	the addressee of the decline is the renter of Rental	

Figure 5.15: Action rule for T1/rq (*assess* and *response* parts) [1]

Rental concluder will now, after complying with the assessment, promise rental concluding according to the action rule designated in Figure 5.18.



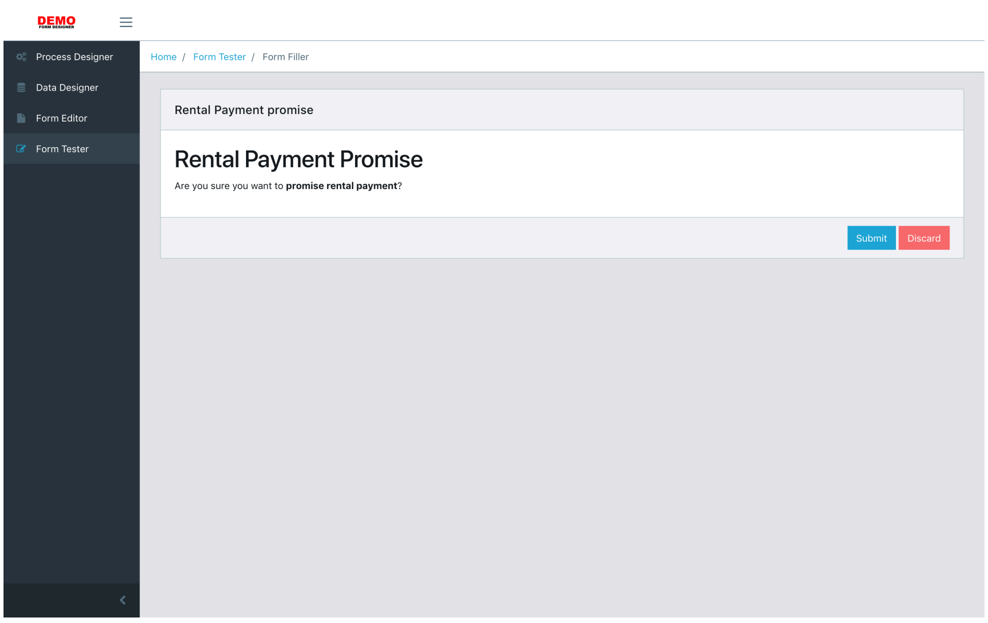
Rental Payment request

Rental Payment Request

Paid rental amount

50

Figure 5.16: Requesting *rental payment* transaction



Rental Payment promise

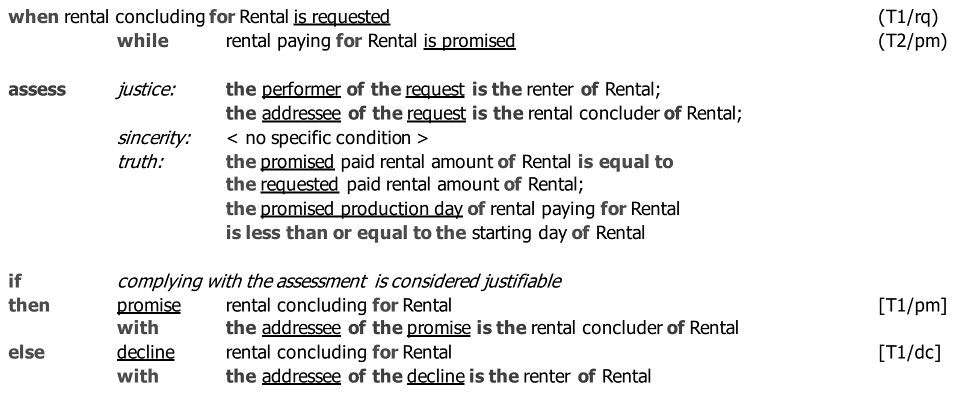
Rental Payment Promise

Are you sure you want to **promise rental payment**?

Figure 5.17: Promising *rental payment* transaction

4. **Promising T1 (rental concluding):**

   Rental concluder promised rental concluding – T1 now holds up until rental payment is accepted.



```
when rental concluding for Rental is requested                                  (T1/rq)
          while       rental paying for Rental is promised                      (T2/pm)

assess    justice:    the performer of the request is the renter of Rental;
                      the addressee of the request is the rental concluder of Rental;
          sincerity:  < no specific condition >
          truth:      the promised paid rental amount of Rental is equal to
                      the requested paid rental amount of Rental;
                      the promised production day of rental paying for Rental
                      is less than or equal to the starting day of Rental

if        complying with the assessment is considered justifiable
then      promise     rental concluding for Rental                              [T1/pm]
          with        the addressee of the promise is the rental concluder of Rental
else      decline     rental concluding for Rental                              [T1/dc]
          with        the addressee of the decline is the renter of Rental
```

Figure 5.18: Action rule for T2/pm

5. **Stating T2 (rental payment):**

   Payer paid for the order stated the payment using a generated form.

   Rental concluder will now assess the action rule for T2/st. Assuming the assessment is complied with, i.e. the payment amount is correct and is in time, rental concluder will accept T2.

6. **Accepting T2 (rental payment):**

   Rental concluder accepted the payment using a generated form.

   Accepting rental payment implies that rental the execution phase of rental order transaction can continue and thus it can be stated.

7. **Stating T1 (rental concluding):**

   Rental concluder stated the order to renter using a generated form.

8. **Accepting T1 (rental concluding):**

   Renter accepted the order using a generated form.

The *Rental Order* business process is done simulating.

5.3 Conclusion

Fact Model of the Rent-A-Car case study and related forms described in section 5.1 have been demonstrated:

- Data structures designed in section 2.4 are able to fully represent Rent-A-Car case study Fact Model and related forms.
- WYSIWYG editors of all the data structures were demonstrated and are capable of operating with the data structures.

- Testing interface for filling out the forms is able to save form field values to a relational database and retrieve the values of form fields back later.
- Proposed concept of business process simulation works for the use case.

Conclusion

Thesis Goals

The primary goal of this thesis was to design, implement and test a proof-of-concept web application for designing web forms based on business process models with additional functionalities to draw DEMO Fact Models and submit the forms to a relational database. The secondary goal was to design a data model for DEMO Fact Model and related forms which the application will take advantage of. Both of these goals had to be demonstrated on the Rent-A-Car case study from the book *The Essence of Organisation*.

Evaluation

In the thesis, all thesis goals were analyzed and based on the requirements, application wireframes and functional specification have been created.

Based on DEMO Fact Model specification, the data model for model and instances of Fact Model were designed with relational database principles in mind.

Besides, a data model for representing forms and subset of Process Model was designed in order to assign designed forms to C-acts.

When choosing implementation technologies for the application, special attention was paid to adopting a convenient WYSIWYG drag-and-drop JavaScript component for designing web forms, therefore multiple solutions solving this task were researched.

The application itself was implemented using the SPA (single-page application) design approach, meaning the application was split into two components – client and server. The client component of the application was tested by the developer, and for testing business layer logic of the server component, unit testing was used.

After the implementation was complete, the application demonstrated its features on the Rent-A-Car case study from the book *The Essence of Organisation.*

Future Development

As a next development milestone, the application could be extended with support for Process Model and Construction Model modelling which could remarkably improve quality of business process structure declaration and simulation. Such extension might make the application ready for production environments.

Since the application is modular and already has integrated a multi-purpose diagramming solution, the application could be extended quickly.

Bibliography

[1] Perinforma, A. *The essence of organisation: an introduction to enterprise engineering*. Netherlands: Sapio Enterprise Engineering, third revised edition, 2017, ISBN 978-90-815449-4-8.

[2] Weske, M. *Business process management: concepts, languages, architectures*. Berlin: Springer-Verlag Berlin Heidelberg, second edition, 2012, ISBN 978-3-642-28615-5.

[3] Araki, A.; Iijima, J. A Pension System Redesign Case – Limitations and Improvements on DEMO. In *Advances in enterprise engineering VIII : 4th Enterprise Engineering Working Conference, EEWC 2014, Funchal, Madeira Island, Portugal, May 5-8, 2014. Proceedings*, edited by D. Aveiro; J. Tribolet; D. Gouveia, Cham: Springer Berlin Heidelberg, 2014, ISBN 978-3-319-06504-5, pp. 31–45, doi:10.1007/978-3-319-06505-2. Available from: `https://www.springer.com/us/book/9783319065045`

[4] Dietz, J. DEMOSL-3 : Demo Specification Language: Version 3.6 [online]. 2017, [cit. 2018-04-30]. Available from: `http://www.ee-institute.org/download.php?id=207&type=doc`

[5] Google. *Google Forms [online]*. 2018, [cit. 2018-01-05]. Available from: `https://docs.google.com/forms`

[6] Cognito LLC. *Cognito Forms [online]*. 2018, [cit. 2018-01-05]. Available from: `https://www.cognitoforms.com/`

[7] Kevin Chappell and Kelly Brent and Ismo Vuorinen. *jQuery formBuilder — Drag & Drop Form Creation [online]*. 2018, [cit. 2018-01-05]. Available from: `https://formbuilder.online/`

[8] Kevin Chappell and Kelly Brent and Ismo Vuorinen. *jQuery formBuilder GitHub Project Page [online]*. 2018, [cit. 2018-01-05]. Available from: `https://github.com/kevinchappell/formBuilder`

[9] Draggable llc. *Formeo GitHub Project Page [online]*. 2018, [cit. 2018-01-05]. Available from: `https://github.com/Draggable/formeo`

[10] Form.io LLC. *Form.IO Form Builder Application GitHub Project Page [online]*. 2018, [cit. 2018-01-05]. Available from: `https://github.com/formio/ngFormBuilder`

[11] Four Labs. *fl-form-builder GitHub Project Page [online]*. 2018, [cit. 2018-01-05]. Available from: `https://github.com/fourlabsldn/fl-form-builder`

[12] Kevin Chappell and Kelly Brent and Ismo Vuorinen. *jQuery form-Builder Docs [online]*. 2018, [cit. 2018-01-05]. Available from: `http://formbuilder.readthedocs.io/en/latest/`

[13] Form.io LLC. *Form.io Front Page [online]*. 2018, [cit. 2018-01-05]. Available from: `https://form.io`

[14] Dotcomfactory. *Form Designer - A JQuery Form Builder Tool [online]*. 2018, [cit. 2018-01-05]. Available from: `https://codecanyon.net/item/form-designer-a-jquery-form-builder-tool/17765354`

[15] Scott, E. *SPA Design and Architecture: Understanding single-page web applications*. Shelter Island, NY: Manning Publications Co., 2016, ISBN 978-1-6172-9243-9.

[16] Fowler, M. Presentation Domain Data Layering. *martinFowler.com [online]*, August 2015, [cit. 2018-03-30]. Available from: `https://martinfowler.com/bliki/PresentationDomainDataLayering.html`

[17] Abramov, D. Presentational and Container Components. *medium.com [online]*, March 2015, [cit. 2018-04-02]. Available from: `https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0`

[18] Freeman, A. *Pro Angular*. United States New York, NY: Apress, second edition, 2017, ISBN 978-1-4842-2306-2.

[19] Google. *Angular Project Front Page [online]*. 2018, [cit. 2018-04-05]. Available from: `https://angular.io/`

[20] Lukasz Holeczek et al. *CoreUI GitHub Project Page [online]*. 2018, [cit. 2018-04-17]. Available from: `https://github.com/coreui/coreui-free-bootstrap-admin-template`

[21] Microsoft Corporation. *TypeScript Project Front Page [online]*. 2018, [cit. 2018-03-31]. Available from: `https://www.typescriptlang.org/`

[22] JGraph Ltd. *mxGraph 3.9.3 Project Front Page [online]*. 2018, [cit. 2018-04-05]. Available from: `https://jgraph.github.io/mxgraph/`

[23] Freeman, A. *Pro ASP.NET Core MVC*. Berkeley, CA: Apress, sixth edition, 2016, ISBN 978-1-4842-0398-9.

[24] Daigneau, R. *Service design patterns fundamental design solutions for SOAP/WSDL and restful Web services*. Upper Saddle River, NJ: Addison-Wesley, 2012, ISBN 978-0-321-54420-9.

Appendix A

Acronyms

API	Application Programming Interface
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DAO	Data Access Object
DBMS	Database Management System
DEMO	Design & Engineering Methodology for Organizations
DOM	Document Object Model
DTO	Data Transfer Object
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
MVVM	Model-View-ViewModel
MVW	Model-View-Whatever
OFD	Object Fact Diagram
ORM	Object-relational mapping
REST	Representational State Transfer
SQL	Structured Query Language
TPT	Transaction Product Table
UX	User Experience
WYSIWYG	What you see is what you get

XHR	XmlHttpRequest
XML	Extensible Markup Language

Appendix B

Contents of Enclosed CD

```
│ README.md.........................the file with CD contents description
└─ src ................................. the directory with source codes
   ├─ application ............. implementation sources of the application
   │  └─ README.md ... instructions for building and running the application
   ├─ thesis .............the directory of LaTeX source codes of the thesis
   └─ tools ........................ implementation sources of the tools
└─ text ........................................ the thesis text directory
   └─ thesis.pdf.........................the thesis text in PDF format
```