A Java-based Framework for Case Management Applications

André Zensen
andre.zensen@fh-bielefeld.de
and Jochen M. Küster
jochen.kuester@fh-bielefeld.de

Bielefeld University of Applied Sciences
Bielefeld, Germany

Abstract. Case Management aims to support knowledge-intensive, flexible and non-routine processes. While modeling of Case Management applications is supported by the notation Case Management and Model Notation, the design and implementation of such applications can be realized using one of the currently available heavy-weight tool suites. In situations where such heavy-weight tool suites cannot be applied, a case management application must be implemented step by step from scratch. In this paper, we propose a framework to systematically create light-weight Case Management applications with CMMN execution semantics. Reusable elements are assembled into case blueprints and executed together with individual implementations. Case Management applications can be realized with less effort compared to a step by step approach. Developers can focus on business logic and supporting graphical user interfaces. As a proof of concept, we use our framework to implement part of a more complex CMMN case study.

1 Introduction

Knowledge work and knowledge-intensive processes gain importance especially in industrialized and information societies. While routine work and known procedures can be supported and automated by IT systems such as work flow systems, this is not necessarily the case for flexible and unstructured work. Yet this kind of work is more and more needed to adapt to new situations. Case Management (CM) aims to support knowledge-intensive, flexible and non-routine processes. The Case Management and Model Notation (CMMN) aims to become a standard notation for CM.

After requirements for a Case Management solution have been specified and a CMMN model has been created, one approach to create a CM application is to use a commercially available proprietary solution, such as by ISIS Papyrus [22] or IBM [15]. However, these often do not use a notation standard and come with the disadvantages of high costs and vendor lock-in. Alternatively, a complex BPMN engine with partial CMMN functionality could be used, such as Camunda [2], with the disadvantage of only partial CMMN support and part (the BPMN part) of the engine being not used at all. Another different approach is to create a CM application from scratch, e.g. by programming a solution directly in Java. However, due to the complexity of the CMMN semantics, such an approach is also costly and time consuming.

In this paper, we present a framework for creating light-weight applications based on CMMN. The idea is to enable the systematic realisation of case management applications based on CMMN models. The framework provides reusable building blocks based on a simplified CMMN structure. The framework itself makes use of proven architectural and design patterns, is flexible and adaptable and not restricted to a specific domain. A case management application can then be assembled by reusing building blocks of the framework for reoccurring functionality of any CM application and complementing them with additional components, such as specific UI components. Using the current framework prototype, an example application based on a case study has been implemented as a proof of concept.

The paper is structured as follows: Section 2 introduces concepts of CM and addresses problems of traditional and activity-centric process management. It also introduces CMMN elements and its execution semantics. Requirements for the framework are derived in Section 3. In Section 4 the overall architecture of the framework is introduced. Section 5 describes how our framework can be used to implement CM applications. Section 6 reports on an implementation of a case study. Related work is discussed in Section 7, before a conclusion and outlook on future research is given in Section 8.

2 Concepts and Modeling of Case Management Processes

Activity-centric process management with tasks along pre-defined paths and structures has its limits with regards to non-predetermined paths and high degrees of flexibility. Modeling such flexibility with traditional tools quickly leads to cluttered models [25, 26].

CM processes are handled as often long lasting cases, which are coordinated and handled by case workers in a collaborative fashion. The goal of a case is usually known, while the path leading there varies or cannot be pre-determined at all [23].

CM organizes activities around a case file, such as those of a legal or medical setting [3]. The roots of case management can be found in patient care: Depending on a patient's needs and state of health, different treatments and services are chosen for the patient to reach the overarching goal of improving the patient's health. How exactly this goal is achieved might not be known beforehand, e.g. a method of treatment might not be available or applicable to the case and unplanned treatments might become necessary at a later point [14].

Tasks are performed based on data and generate it. Which tasks are to be performed or necessary data to be collected is decided by the case workers. Task states change, e.g. information becomes available or updated and this leads to different decisions, tasks and (intermediate) goals to be achieved. Paths are thus not pre-determined from the beginning, but evolve as the case progresses over time. This approach is not restricted to healthcare domains: highly flexible processes or parts thereof can also be handled as a case across very different domains such as production [23, 26].

Case management aims to address four criteria found to be problematic in traditional work flow management systems and process management [25]:

Context tunnelling is avoided by making information available to all case workers and not restricting it to single tasks currently worked on.

Flexible paths evolve over time dependent on available information and states instead of being restricted by pre-determined paths and previously absolved tasks.

Division of labour is not restricted to certain authorised execute roles, but shared among participating case workers who have different roles.

Data and information can be added and edited regardless of specific activities and is thus not bound to a temporal order of tasks.

At least three approaches to case management have emerged, mainly differing in their degree of flexibility [7]: Adaptive Case Management (ACM) [10, 19, 9], Dynamic Case Management (DCM)[21] and Production Case Management (PCM) [5]. While ACM is the most flexible and aims to enable case workers to build their case on the fly as a "‘do it yourself"' during execution, DCM enables case workers to adapt and expand on existing case structures which were set up before execution. The least flexible of the three approaches is PCM. It can be seen as a best of breed of previously enacted ACM or DCM cases, using templates of tested and proven flexible structures and partially known paths. These are then combined into a case structure before execution.

The Case Model and Management Notation (CMMN) 1.1 [4] is a declarative approach to process and case modeling. Its specification regards case management as a "‘*proceeding that involves actions taken regarding a subject in a particular situation to achieve a desired outcome*"'. It can be regarded as a modeling notation closest to the characteristics of the DCM and PCM approaches, but also captures ACM characteristics [6]. Its structure and execution semantics are heavily based on the Guard Stage Milestone approach (GSM) [8] and the case handling paradigm described in [25].

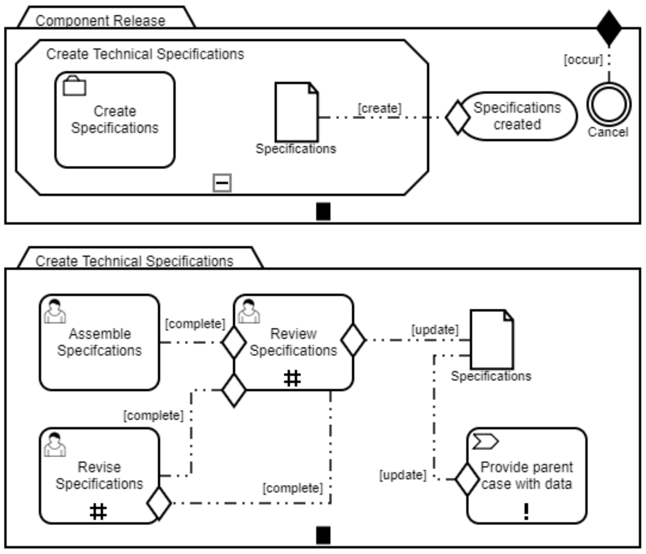


Fig. 1. Simple Component Release Example CMMN Model

Fig. 1 shows two (simplified) CMMN models based on a case study conducted in [26]. Like its ‘sibling-notation’, the Business Process and Management Notation (BPMN) [1], CMMN offers a formalised graphical notation with defined execution semantics for its elements as well as an underlying specific markup language. The shown models express the high flexibility provided by CMMN with its decorators and sentry concept controlling the work flow. The top case is started by case workers, who can also trigger the *EventListener* **Cancel** to terminate the parent case. The required *CaseTask* **Create Specifications** in *Stage* **Create Technical Specifications** starts the bottom case **Create Technical Specifications**. Both cases automatically complete once all required elements are completed (signified by the exclamation mark at the bottom of *HumanTask* **Assemble Specifications**).

The repetition decorators on *HumanTasks* **Review Specifications** and **Revise Specifications**) in the bottom case are used to model a loop structure. They evaluate a boolean *Property* of *CaseFileItem* **Specifications** which expresses whether or not the specifications have been approved or need to be revised and reviewed again.

Once the specifications are assembled, either in the context of work on the *HumanTask* or by other means, one of the *EntrySentries* on *HumanTask* **Review Specifications** is satisfied and transitions into an active state.

Apart from other *Sentries* observing state transitions of other elements and data in order to activate an element they are attached to, an *IfPart* controls entry to *ProcessTask* **Provide parent case with data**. Once the specifications are updated and have been approved, the *ProcessTask* is used to transfer data to the *CaseFileItem* of the parent case.

Then the *CaseFileItem* of the parent case undergoes its *create* transition, the *EntrySentry* on *Milestone* **Specifications created** is satisfied and the *Milestone* occurs, leading to a completion of the parent case.

The next chapter discusses requirements to build CM applications to support such cases.

3 Requirements for a Light-Weight Case Management Framework

CM applications can be characterized as distributed applications for collaborative work on long running cases. The application enables tasks depending on the overall context, data and state of a case instance. Case workers view and manipulate data to drive the case.

Requirements can be derived from CM characteristics [3, 10] and our project work on component release processes as cases [26]. The requirements can be divided into base (1-6), CMMN specific (7-9) and those addressing problems of traditional process management (10-13). The list is not all-encompassing, but focus on minimum requirements for our framework to support light-weight, CMMN based CM applications.
Base requirements of the framework cover characteristics of CM:
REQ-1 *Offer a structured approach for implementing CM applications.* The framework needs to enable developers to implement CM applications in a systematic fashion.
REQ-2 *Remain light-weight.* Vendor-specific technology should be avoided, e.g. specific application server and database management system. Instead open standards should be used.
REQ-3 *Support case workers with graphical user interfaces.* Case workers need to view and edit data, often based on unstructured documents.

REQ-4 *Support multiple case workers with different roles.* A role system is required to restrict access to sensitive functions.
REQ-5 *Provide access for external systems.* Web-services are required for case workers accessing an application via web browsers and for external systems performing automated tasks, e.g. for call backs upon finishing the task.
REQ-6 *Support long running case instances.* A persistence mechanism is required to store and retrieve case instances which can last up to several years.

CMMN-specific requirements encompass building blocks and their behaviour:
REQ-7 *Provide foundational building blocks for a CM application.* The framework should provide classes representing CMMN elements, such as *Stage*, *HumanTask* and *ProcessTask*, in order to build case structures.
REQ-8 *Capture CMMN execution semantics and behaviour.* A common understanding of how the case instances behave needs to be established by basing behaviour of building blocks on the CMMN specification of execution semantics including those of decorators.
REQ-9 *Support the sentry concept.* Central to dynamic flows in CMMN are *Sentries*. The framework needs to support these to link elements together in order to influence states of elements, the availability and necessity of tasks to help guide case workers and to open possible pathways depending on element states and data values.

The following requirements address four problem areas of traditional process management, such as context tunnelling, rigid roles and path flexibility (see Sec. 2):
REQ-10 *Avoid context tunnelling.* All case workers should be able to access all information of a case instance to make better decisions.
REQ-11 *Enable flexible paths.* CMMN structures and execution semantics enable highly flexible and context dependent paths. *Milestones* further aid to focus on achieving a goal and what *can* be done instead of rigid paths focusing on what *should* be done.
REQ-12 *Enable flexible division of labour.* A simple role system should enable case workers to choose and work on tasks as well as trigger events they deem necessary, going beyond a restricted execute role.
REQ-13 *Make access to data flexible.* Case workers should be able to view and edit data regardless of current tasks.
How these requirements are supported and fulfilled by the framework and its architecture is described in the next chapter.

4 Architecture of the Framework

In this section we present our framework architecture and reference how it fulfills the requirements of the previous chapter. The architecture is viewed from two perspectives. A high-level perspective shows the structure of the framework itself and how it is embedded in an application server. A design-level view shows a domain model covering classes representing CMMN elements used to build case models. Also discussed are basic services provided and used by the framework.

Currently, planning elements (such as classes *PlanItemDefinition* or *PlanFragment*) are not supported by the framework and have to be transformed into elements making use of available elements (e.g. turning a discretionary item into a manually started item).

4.1 A CM Application built with the Framework

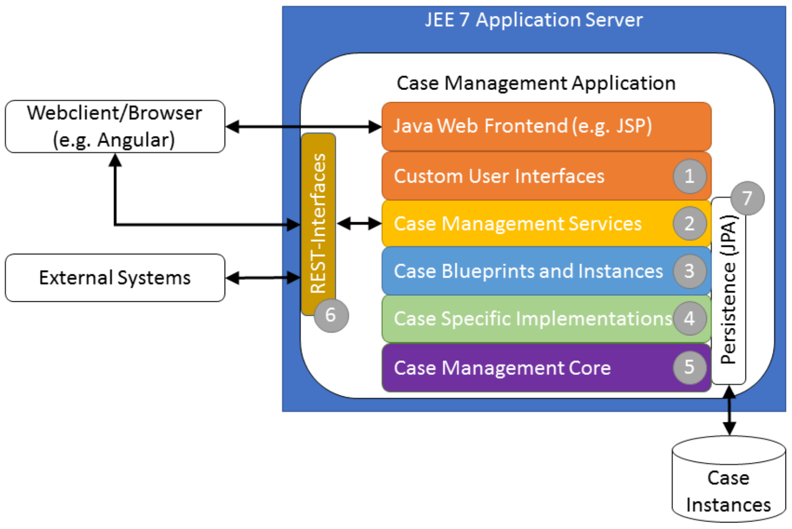


Fig. 2. Layers of the Application in the JEE Server Environment

The components of a CM application built with the framework is shown in Fig. 2. The application is run in a JEE7 application server and consists of five central layers (1-5). Access to the application is provided either directly via an integrated Java-based frontend framework or via Representational State Transfer (REST) interfaces (6). Both use stateless services provided by the framework (2). The services are described in Sec. 4.3.

External systems can interact with the application the same way as web clients, i.e. via the REST interfaces provided. These are a common approach to realize access to required web-services. The basic architecture fulfills REQ-1. These and additional components are described in this section.

The standard [20] is used to implement the framework. It fulfills REQ-2 by enabling portability among application servers and avoiding vendor-specific annotations. Furthermore, JEE7-conform application server and database management system supported by JPA can be used to run CM applications built with the framework, avoiding vendor lock-in.

The different components of the layered structure cover almost all requirements:

1. ***Custom User Interfaces*** grant case workers access to the application and include overviews of case instances, views for current tasks and data. Interfaces use services (2) to interact with the application. Together with these, REQ-3 and 4 are fulfilled

by providing graphical user interfaces to case workers to access case information and data.
2. ***Case Management Services*** provide functions for central elements and access to case blueprints and instances (3). Using the services, REQ-10, 12 and 13 are fulfilled by providing access to instance data and a selection of available tasks to suitable roles.
3. ***Case Blueprints and Instances*** build on the core classes and individual implementations (4), fulfilling REQ-7 by providing case instance structures with CMMN based building blocks.
4. ***Case Specific Implementations*** include reusable custom class specialisations and implementations generated during runtime, e.g. for logic of *ProcessTask* classes or *Rules* for *decorators*. Context for the reusable components is provided by referenced element instances. These fulfill REQ-8 and 11 by implementing behaviour based on CMMN execution semantics.
5. ***Case Management Core*** elements are used by all other layers. Core classes capture the CMMN execution semantics, which fulfills REQ-1 and 7 by providing a systematic base structure. Requirement 4 is fulfilled by enabling associations of roles to tasks. Like the previous item, the core also fulfill REQ-8, 9 and 11.
6. The ***REST interfaces*** build on internal services, fulfilling requirement 5 by providing acccess to the CM application for external clients and systems.
7. ***Persistence*** is based on JPA. Persistence contexts are used by the stateless services of (2). They cover *Create, Update and Delete* (CRUD) operations, e.g. to persist case blueprints and to retrieve and manipulate resulting instances in their context. Together with the persistent storage used to persist and retrieve case instances, this component fulfills requirement 9. Data storage has to be compatible with JPA.

4.2 Main Building Blocks for a CM Application

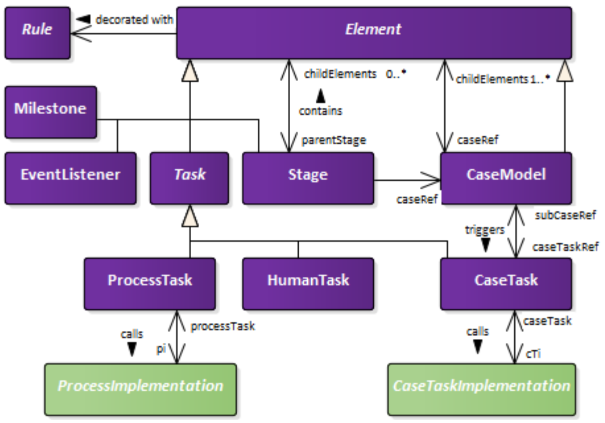


Fig. 3. Base CM Elements

This section highlights the class models of the previously discussed layers **Case Specific Implementations** (4) (light green) and **Case Management Core** (5) (purple).

Discussed are the *Element*, role, data and *Sentry* structures provided by the framework and how they are connected to each other. These structures constitute the layer **Case Blueprints and Instances** (3).

Fig. 3 shows classes representing CMMN elements. Central is the abstract class *Element* from which specialisations inherit base attributes. These include an id for persistence, the current state and *cmId* as a CM-related identifier. The *CaseModel* reference *caseRef* provides context to case instances and can be used in queries.

Internally, factory methods are used to instantiate custom implementations of layer (4) during runtime. The correct context is provided by the reference to *Elements* and *CaseModels*.

Classes *CaseModel* and *Stage* serve as containers for child elements like *Task*, *MileStone* and *EventListener*. Specialisation of abstract class *Task* further include *ProcessTask* and *CaseTask*. Both are associated with abstract classes used for case specific implementations located in layer (4).

ProcessImplementations can be used to execute algorithms, e.g. to send an e-mail or communicate with external systems such as a BPM platform, while *CaseTaskImplementation* is used to start and as a link to a nested sub-case. For this a method is overridden.

Where permitted as per CMMN specification, *Elements* can be associated with *Rules* representing CMMN decorators. Specialisations are shown in Fig. 5. Fig. 4 shows a

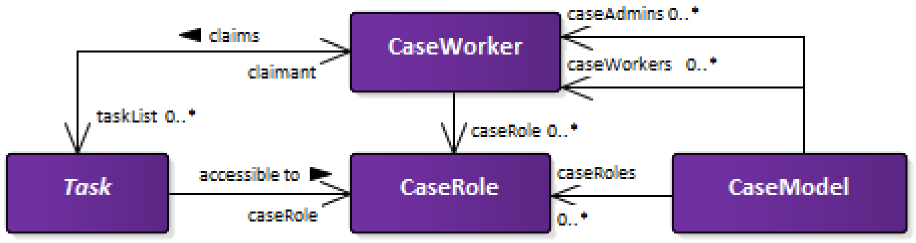


Fig. 4. Simple Role System

simple role system based on classes representing *CaseWorkers* (both as case admins and regular workers) and *CaseRoles* in the context of a *CaseModel*. The structure can be used to restrict access to *Tasks*. *CaseWorkers* can claim *Tasks* which are then included in their individual task list.

Fig. 5 shows classes representing the CMMN *CaseFileItems* associated with a *CaseFile* of a *CaseModel*. *SimpleProperties* are for primitive data types to structure data, while *CaseFileItemAttachment* is added to reference unstructured data such as document files. The figure also shows how *CaseFileItems* are referenced by the three *Rule* specialisations representing CMMN decorators: *RequiredRule*, *RepetitionRule* and *ManualActivationRule*. A case specific implementation, *RuleExpression*, is used by a *Rule* to evaluate a referenced *CaseFileItem*. For this a method is overridden.

Fig. 6 shows classes to build *Sentry* structures. Not displayed are specialisations *EntrySentry* and *ExitSentry*. Where permissible, these can be attached to *Elements*. For example, a *CaseModel* can only be associated with *ExitSentries*, while most of the other elements can have both types of *Sentry*. *ElementOnPart* and *CaseFileItemOnPart* are

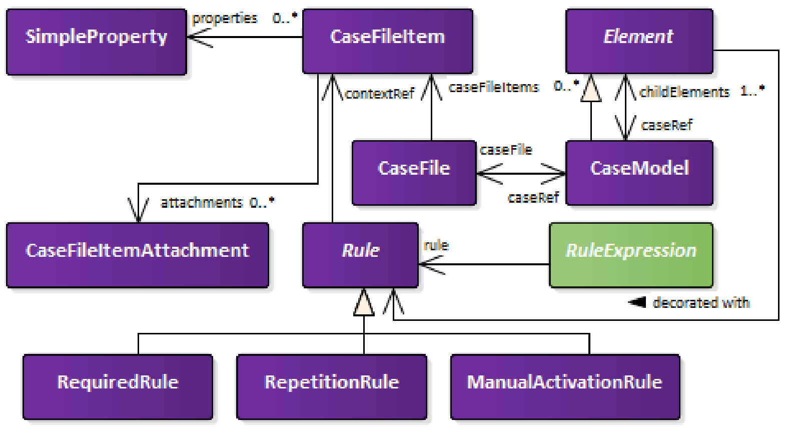


Fig. 5. Data Structure

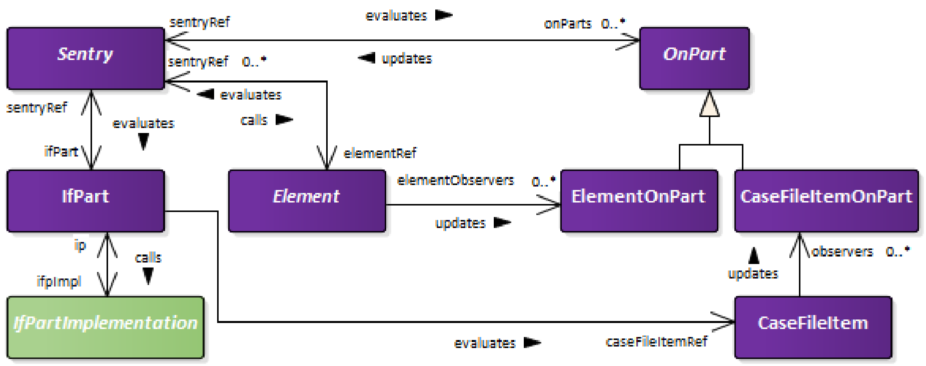


Fig. 6. Sentry Structure

specialisations of abstract class *OnPart*, used to observe life cycle transitions. A *Sentry* can have an *IfPart*, which like *Rules* references a *CaseFileItem* for evaluation.

Names of the directed associations highlight the execution semantics of *Sentries*: When an *Element* or *CaseFileItem* transitions to another state, the observing *ElementOnPart* or *CaseFileItemOnPart* updates their *Sentry*. The notified *Sentry* then evaluates its (other) *OnParts* and *IfPart* if it exists. If the *Sentry* conditions are fulfilled, the *Element* it is attached to is called and its state transitioned.
How these elements are assembled into a *CaseModel* blueprint of layer **Case Blueprints and Instances** (3) is shown in Ch. 5.

4.3 Case Management Services

The framework provides basic services for CRUD operations, covering all elements making up a case, such as *CaseModel*, *Tasks* or *Milestones*. A central *CaseService* provides methods to persist, retrieve and transition (specific) case instances. A *CaseFileService* provides methods to manipulate case data, i.e. retrieve *CaseFileItems*, transition them from one state to another or to manipulate their *Properties*. Its functions can further be used for monitoring and reporting purposes.

A *TaskService* can be used to retrieve a list of (available) *(Human)Tasks* for a specific case instance, across all cases, or by role restriction. *HumanTasks* claimed by a *CaseWorker* can also be queried. The service also manages transitions of *Task* states. Repetition decorators are also managed by the service, i.e. it creates a new instance of a given task to be repeated.

Working closely with the *TaskService* is the *CaseWorkerService*, which covers operations to manage case workers, i.e. to query either for all, by id or by login credentials, as well as to create, update and delete them. A *CaseWorker* object is used in the *TaskService* to associate an available *HumanTask* to a specific worker. Other services are a *MilestoneService* and an *EventListenerService*, mainly used to retrieve information on their states or in the latter case to trigger an event.

5 Approach to Implementing Case Management Applications

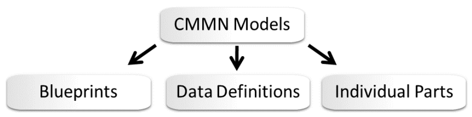
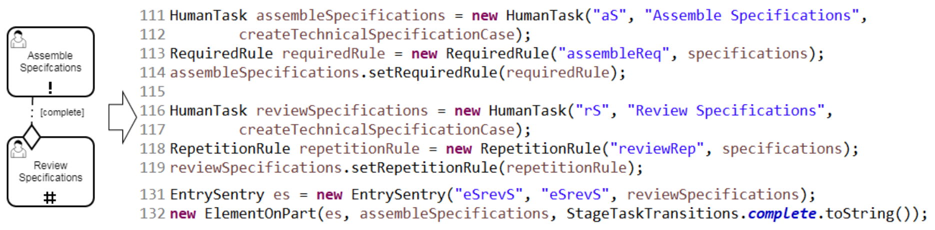


Fig. 7. CMMN Model to CM Application with the Framework

In this section, we show how the elements of a CMMN model are manually translated to code using the framework. Our aim is not to parse and execute a CMMN mark up with all the intricate details and abstraction levels of the specification in a generic environment. We want to enable developers to systematically build and implement case management applications on top of light-weight core building blocks including basic services.

Fig. 7 shows three main components resulting from CMMN models: case model blueprints, data definitions and individual parts. These make up the layers 1-4 described in Sec. 4.1 and build on core classes located in layer 5.



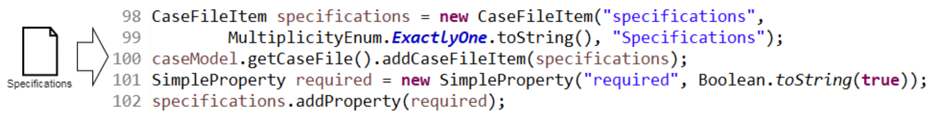
```
HumanTask assembleSpecifications = new HumanTask("aS", "Assemble Specifications",
        createTechnicalSpecificationCase);
RequiredRule requiredRule = new RequiredRule("assembleReq", specifications);
assembleSpecifications.setRequiredRule(requiredRule);

HumanTask reviewSpecifications = new HumanTask("rS", "Review Specifications",
        createTechnicalSpecificationCase);
RepetitionRule repetitionRule = new RepetitionRule("reviewRep", specifications);
reviewSpecifications.setRepetitionRule(repetitionRule);

EntrySentry es = new EntrySentry("eSrevS", "eSrevS", reviewSpecifications);
new ElementOnPart(es, assembleSpecifications, StageTaskTransitions.complete.toString());
```

Fig. 8. Model To Code: HumanTasks

Framework building blocks are assembled into blueprints which are used to instantiate case instances. They contain the core structures of a CMMN model. For a complete example of a blueprint structure see Fig. 11 in Ch. 6. Each element in the model is represented by a framework class.

Fig. 8 shows a code example of two *HumanTasks* and the resulting code. First, two *HumanTask* are created (lines 111 and 116) and references to a *RequiredRule* (line 113f.) and *RepetitionRule* (line 118f.) are added. Their given names will be referenced later in a factory method to return the correct rule implementation.

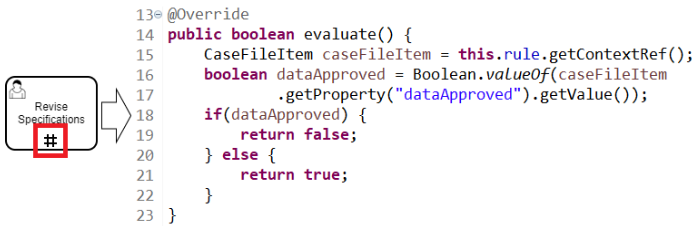
The *EntrySentry* attached to the *HumanTask* **Review Specifications** is created and an *ElementOnPart* added in line 131f. with the transition seen in the CMMN model as the connector labelled *[complete]*.



```
 98 CaseFileItem specifications = new CaseFileItem("specifications",
 99         MultiplicityEnum.ExactlyOne.toString(), "Specifications");
100 caseModel.getCaseFile().addCaseFileItem(specifications);
101 SimpleProperty required = new SimpleProperty("required", Boolean.toString(true));
102 specifications.addProperty(required);
```

Fig. 9. Model To Code: CaseFileItem

Data definitions consist of *CaseFileItems* and their *Properties*. These can be included in blueprints, or added later via services. A reference to the *CaseFileItem* itself is required in the blueprint for *Sentry* references. Fig. 9 shows a code example for a basic definition. The created *CaseFileItem* in line 98f. is added to the *CaseFile* of the *CaseModel* in line 100. The added Property 'required' seen in lines 101f. is used to evaluate the decorator on the *HumanTask* seen above on the left hand side in Fig. 8.



```
13 @Override
14 public boolean evaluate() {
15     CaseFileItem caseFileItem = this.rule.getContextRef();
16     boolean dataApproved = Boolean.valueOf(caseFileItem
17             .getProperty("dataApproved").getValue());
18     if(dataApproved) {
19         return false;
20     } else {
21         return true;
22     }
23 }
```

Fig. 10. Model To Code: Repetition Rule

Individual parts include graphical user interfaces (GUI), *IfParts* associated with *Sentries*, decorator rules and implementations for *Process*- and *CaseTasks*. GUI are not further discussed but can use services provided by the framework to access all case instance elements, e.g. a GUI of *HumanTask* **Assemble Specification** could access the *CaseFileService* to upload specification documents.

Fig. 10 shows the code needed to implement the repetition decorator on the shown *HumanTask* (note the highlight). The correct instance of the *CaseFileItem* to be evaluated is provided by the reference from line 15. Its *Property* 'dataApproved' is accessed and a boolean value is returned.

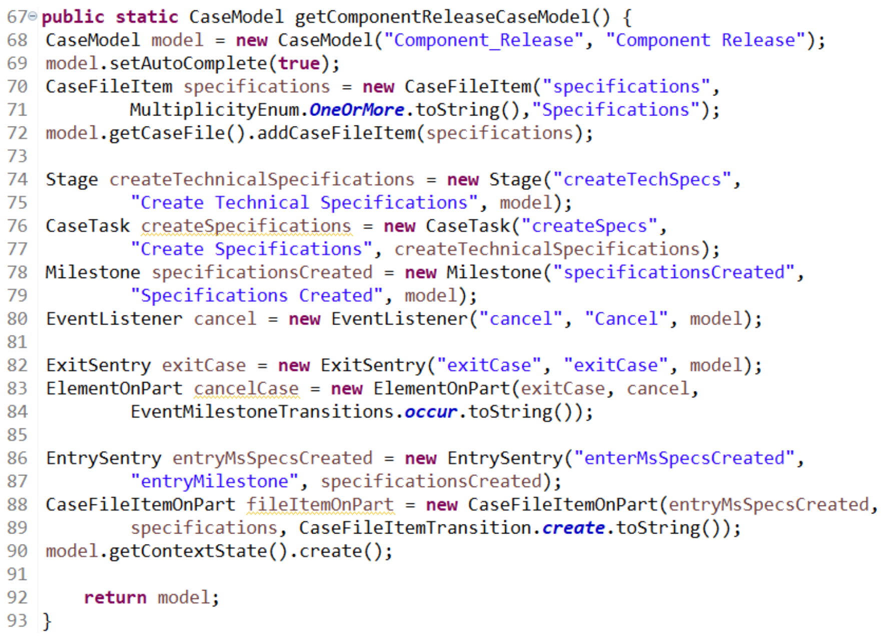
Similarly, methods of abstract classes used for *IfParts* and *Process*- and *CaseTasks* are overridden. A factory returns the individual implementation at runtime which is then used by the framework.

Factory methods called by the framework which are used to return blueprints and individual parts need to be adjusted. Elements and their contained references, such as a *HumanTask* and its reference to its *CaseModel*, provide the correct context.

6 Case Study

Using the approach presented in the previous section, we have implemented a CM application with the framework[1], deployed on a Tomcat 8.5 TomEE PluME application server with a JTA managed MySQL data source. The presentation layer is implemented with Vaadin 8 [24] and its CDI plug-in using a *HumanTask cmId* attribute. Previously shown Fig. 1 shows the CMMN models used to assemble *CaseModel* blueprints and implement required implementations. They capture a small simplified part of a previous case study in [26].

Fig. 11 shows the blueprint for the upper case *Component Release* (see Fig. 1): Here, the CMMN model is directly translated into a Java structure using framework elements such as *CaseModel*, *CaseFile*, *Stage*, *CaseTask* or *Milestone*. First, in line 68, a new *CaseModel* is created. Then in line 70ff., a *CaseFileItem* is created and added to the *CaseModel*. Given to the constructors of *Elements* is a reference to the parent: the *Stage* in line 74 is given a reference to the *CaseModel*, the *CaseTask* is constructed with a reference to that *Stage*, while the other elements such as the *Milestone* are added directly to the *CaseModel*. In the following lines, the whole structure is translated. Of note is line 90, which transitions the *CaseModel* from an initial state to state available.

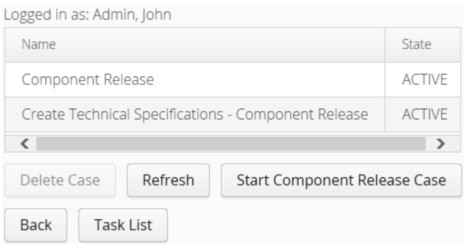


```
67 public static CaseModel getComponentReleaseCaseModel() {
68  CaseModel model = new CaseModel("Component_Release", "Component Release");
69  model.setAutoComplete(true);
70  CaseFileItem specifications = new CaseFileItem("specifications",
71          MultiplicityEnum.OneOrMore.toString(),"Specifications");
72  model.getCaseFile().addCaseFileItem(specifications);
73
74  Stage createTechnicalSpecifications = new Stage("createTechSpecs",
75          "Create Technical Specifications", model);
76  CaseTask createSpecifications = new CaseTask("createSpecs",
77          "Create Specifications", createTechnicalSpecifications);
78  Milestone specificationsCreated = new Milestone("specificationsCreated",
79          "Specifications Created", model);
80  EventListener cancel = new EventListener("cancel", "Cancel", model);
81
82  ExitSentry exitCase = new ExitSentry("exitCase", "exitCase", model);
83  ElementOnPart cancelCase = new ElementOnPart(exitCase, cancel,
84          EventMilestoneTransitions.occur.toString());
85
86  EntrySentry entryMsSpecsCreated = new EntrySentry("enterMsSpecsCreated",
87          "entryMilestone", specificationsCreated);
88  CaseFileItemOnPart fileItemOnPart = new CaseFileItemOnPart(entryMsSpecsCreated,
89          specifications, CaseFileItemTransition.create.toString());
90  model.getContextState().create();
91
92      return model;
93 }
```

Fig. 11. Blueprint for sample process (see Fig. 1)

Individual implementations are required for several elements. Both models are instantiated via factory method calls to get their blueprints. They are then persisted with the help of a *CaseService*. The *EventListener Cancel* can be triggered by a graphical user interface making use of an *EventListenerService* in the service layer of the framework.

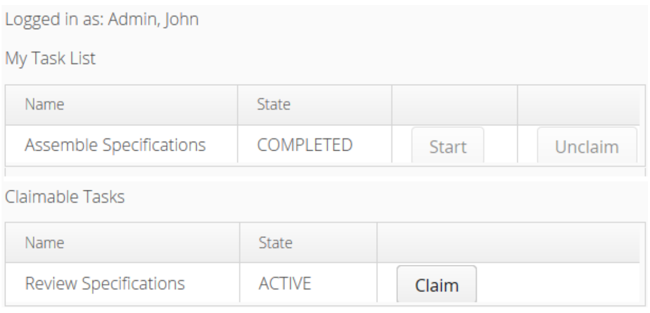
[1] The framework and examples are maintained at https://github.com/fhbielefeldagpm


Logged in as: Admin, John

Name	State
Component Release	ACTIVE
Create Technical Specifications - Component Release	ACTIVE

Fig. 12. Case List View

Case workers instantiate the parent case from an interface which uses a *CaseService* provided by the framework. The bottom case is instantiated automatically via activation of the included *CaseTask*. It calls the blueprint of the referenced child case at runtime, which is then persisted and started. Fig. 12 shows the parent and child case in a simple view listing cases. This view is provided by the framework as part of the presentation layer based on Vaadin. The referenced *CaseTask* is notified and completes once the child case automatically completes after the required *ProcessTask* is completed. While the repetition decorators require rule implementations, the *AutoComplete* decorators (both cases) do not need individual implementations. Workers have the option to claim *HumanTasks*. Fig. 13 shows claimed and completed tasks of a case worker. This view is also provided by the framework as part of the presentation layer based on Vaadin.


Logged in as: Admin, John

My Task List

Name	State		
Assemble Specifications	COMPLETED	Start	Unclaim

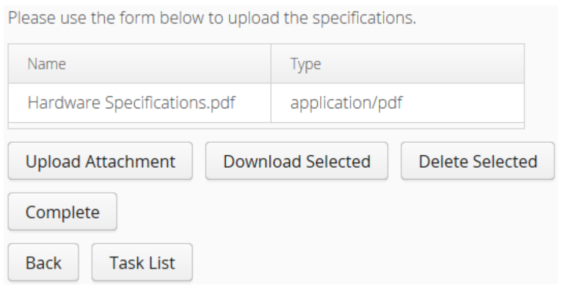
Claimable Tasks

Name	State	
Review Specifications	ACTIVE	Claim

Fig. 13. Task List View

The structure of *CaseFileItem Specifications* needs to be defined in the blueprint. A *SimpleProperty* with a boolean value expressing its approval state is used to evaluate *RequiredRules* and *RepetitionRules*. The required files are either uploaded via completing *HumanTask Assemble Specifications* or via a data viewer component. Both make use of a *CaseFileService*.

Fig. 14 shows a PDF file was added as a *CaseFileItemAttachments* with the help of a *CaseFileService*. It is also used to update and transition the *CaseFileItem* to activate the *HumanTask* to review the specifications. *IfPart* implementations are needed to evaluate whether a revision and thus repetition of the respective *HumanTask* is necessary, or if the last update transition of the data means that it is ready for the *ProcessTask* to upload it to the parent case.


Please use the form below to upload the specifications.

Name	Type
Hardware Specifications.pdf	application/pdf

Fig. 14. Attachment Upload

To give a sense of the amount of work needed to implement the case study application, Table 1 lists the lines of code needed to create working blueprints based on the framework. These include an implementation to execute the *CaseTask*, two *RepetitionRules*, one implementation for the execution of the *ProcessTask* and one for the *IfPart*. They do not include lines of code for user interfaces, but these range from 100 to 200 lines per Vaadin view, including controller logic.

Artifact	**Lines of Code**
Assembled Blueprints	60
Implemented Decorator Rules	20
Implementation of CaseTask	4
Implementation of ProcessTask	10
Implementation of IfPart	10
Total	**114**

Table 1. Lines of Code required to provide working *CaseModel*s

The framework has about 4.700 lines of code organised in about 130 classes (including specialisations) in 15 packages. Graphical user interfaces were built on the framework services. These include a case list overview to instantiate and access cases, task lists to view and claim tasks, as well as GUIs for *HumanTasks*. Provided services are used to access elements such as *CaseModel* and *Task* instances, trigger transitions of *Elements*, or to retrieve and manipulate data in *CaseFiles*.

The example shows how flexible work flows can be designed with CMMN: paths can react to data changes as well as state changes of elements without being bound to an imperative sequence flow. The implementation is supported by the framework to realize CM applications based on the models.

7 Related Work

Most research on case management with CMMN focusses on modeling aspects. Research on CMMN based implementations is available to a lesser extent.

The Darwin Wiki [13] uses a subset of CMMN in an extended wiki platform implementation to empower non-expert end-users to structure processes for knowledge work.

It integrates a graphical editor using a sub-set of CMMN elements to model case work in the wiki.

A reference architecture for model-based collaborative information systems is presented in research related to the Connecare project [18]. It integrates process and data modelling in a fully model-based system enabling non-technical end-users to create case-based processes. It includes process models and a case execution engine based on an extended CMMN sub-set, though it does not further detail how.

An approach to utilize a Content Management Interoperability System (CMIS) to implement an information model based on CMMN is presented in [17]. Its focus is on using a CMIS folder as the CMMN *CaseFile* containing the case instance data, linking CMMN data concepts to existing document management systems.

Camunda [2] and flowable [11] partially support CMMN. In contrast to our framework, which focusses on creating light-weight CM applications, their main focus is on a BPMN engine and platform.

Another approach to implementing CM concepts is FLOWer, but unlike our framework it is not based in CMMN. [25] shows a simplified internal structure of FLOWer, representing artefacts such as case and activity. Activities are directly associated with data objects, forms and roles.

A fragment-based CM (fCM) case engine is presented in the Chimera project [12]. Based on the PCM approach to CM, a process is split into smaller fragments and combined with domain models, object life cycles for data objects as well as goal states. Fragments are modeled with a separate modeler ('Gryphon'). Unlike our framework, it uses elements based on a BPMN sub-set. The fragments are dynamically combined and executed depending on data states.

A data-centric business process management approach similar to but not based in CMMN is presented in research related to the PHILharmonicFlows project [16]. Users can define case-like structures and propagate ad-hoc changes of these and data to already running instances.

8 Conclusion and Future Work

Designing and implementing case management applications is a challenging and complex task. In this paper, we have presented a case management implementation framework based on Java which allows the rapid realization of case management applications based on widely available Java and open source technologies. The framework makes use of Java EE technologies and includes support for major CMMN elements. A first evaluation of our framework has demonstrated that the effort for creating a CM application using our framework is then concentrated on designing user interfaces and implementing the logic of the CMMN model. Other aspects such as case instance management and execution semantics are provided for by the framework.

Future work includes the extension of our framework to cover further elements of CMMN such as discretionary items to support an ACM approach. Other work includes the automatic generation of blueprint skeletons and specialisation stubs, the implementation of fine grained security and user management layers as well as integration with existing systems.

References

1. OMG. Business Process Model and Notation (BPMN) Specification, 2011. Version 2.0.
2. Camunda. Workflow and decision automation platform. `https://camunda.com`.
3. C. Di Ciccio, A.Marrella, and A. Russo. Knowledge-intensive processes: Characteristics, requirements and analysis of contemporary approaches. *JoDS*, 4(1):29–57, 2015.
4. OMG. Case Model Management and Notation (CMMN) Specification, 2016. Version 1.1.
5. A. Meyer et al. Implementation framework for production case management: Modeling and execution. In *IEEE 18th EDOC*, pages 190–199. IEEE, 2014.
6. M. Kurz et al. Leveraging cmmn for acm: examining the applicability of a new omg standard for adaptive case management. In *S-BPM ONE*, 2015.
7. M.A. Marin et al. Case management: An evaluation of existing approaches for knowledge-intensive processes. In M. Reichert and H.A. Reijers, editors, *BPM Workshops*, LNBIP 256, pages 5–16. Springer, 2016.
8. R. Hull et al. Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In M. Bravetti and T. Bultan, editors, *Web Services and Formal Methods*, LNCS 6551, pages 1–24. Springer, 2011.
9. T.T.K. Tran et al. Setup and maintenance factors of acm systems. In D. Hutchison et al., editor, *OTM Workshops*, LNCS 8186, pages 172–177. Springer, 2013.
10. L. Fischer and T. Koulopoulos. *Taming the unpredictable: Real world adaptive case management: case studies and practical guidance*. Future Strategies, 2011.
11. flowable. flowable java business process engines. `https://www.flowable.org/`.
12. BPT group. Chimera. `https://bpt.hpi.uni-potsdam.de/Chimera`.
13. M. Hauder, R. Kazman, and F. Matthes. Empowering end-users to collaboratively structure processes for knowledge work. In *BIS*, LNBIP 208. Springer, 2015.
14. N. Herzberg, K. Kirchner, and M. Weske. Modeling and monitoring variability in hospital treatments: A scenario using cmmn. In F. Fournier and J. Mendling, editors, *BPM Workshops*, LNBIP 202, pages 3–15. Springer, 2015.
15. IBM. Case management overview. `https://www.ibm.com/support/knowledgecenter/SSCTJ4_5.3.2/com.ibm.casemgmt.installing.doc/acmov000.htm`. [Online; accessed 25-January-2019].
16. V. Künzle and M. Reichert. Philharmonicflows: Towards a framework for object-aware process management. *J SOFTW MAINT EVOL-R*, 23:205–244, 2011.
17. M.A. Marin and J.A. Brown. Implementing a case management modeling and notation (CMMN) system using a content management interoperability services (CMIS) compliant repository. *CoRR*, abs/1504.06778, 2015.
18. F. Michel and F. Matthes. A holistic model-based adaptive case management approach for healthcare. In *IEEE 22nd EDOCW*, pages 17–26. IEEE, 2018.
19. H.R. Motahari-Nezhad and K.D. Swenson. Adaptive case management: Overview and research challenges. In *IEEE 15th CBI*, pages 264–269. IEEE, 2013.
20. Oracle. Java EE 7 Technologies. `https://www.oracle.com/technetwork/java/javaee/tech/index-jsp-142185.html`. [Online; accessed 25-January-2019].
21. D. Schuerman, K. Schwarz, and B. Williams. *Dynamic Case Management for Dummies: Pega Special Edition*. John Wiley & Sons, Inc., 2014.
22. Papyrus Software. Adaptive case management. `https://www.isis-papyrus.com/`, 2019.
23. K.D. Swenson and L. Fischer. *How Knowledge Workers Get Things Done: Real-World Adaptive Case Management*. Future Strategies, 2012.
24. vaadin. Vaadin framework 8. `https://vaadin.com/`, 2019.
25. W. M. P. van der Aalst and M. Weske. Case handling: A new paradigm for business process support. volume 53(2) of *Data Knowl. Eng.*, pages 129–162. Elsevier, 2005.
26. A. Zensen and J. Küster. A comparison of flexible bpmn and cmmn in practice: A case study on component release processes. In *IEEE 22nd EDOC*, pages 105–114. IEEE, 2018.