From Monolithic Models to Agile Micromodels

Sebastian Copei[1], Christoph Eickhoff[1], Adam Malik[2], Natascha Nolte[1], Ulrich Norbisrath[3], Jonas Sorgalla[4], Jens H. Weber[5] and Albert Zündorf[1]

[1]*Kassel University, Germany*

[2]*Adam Malik Consulting, Germany*

[3]*Tartu University, Estonia*

[4]*Dortmund Uni. App., Germany*

[5]*University of Victoria, Canada*

Keywords: Modeling, Work Flow, Event Based Systems, Micromodels.

Abstract: This paper proposes a new modeling approach that allows to split classical monolithic class models into a system of small micromodels. The events of such a system are modeled through an additional Event Storming. This new approach facilitates evolution as Event Storming uses events with relatively simple structures and new events may be added easily. Similarly, sets of micromodels are more evolveable as changes remain local to only those micromodels that are concerned by the particular subset of changed events. Different micromodels communicate via a small set of dedicated “interface" events, which again aids evolution.

1 INTRODUCTION

In our modern world, software systems, e.g. in healthcare, are characterized by ever-increasing complexity. A trend whose end is not in sight. A classic approach taught today by universities around the globe to deal with this complexity is to use a holistic model in design, implementation, and communication of a software system (Brambilla et al., 2017). Another popular academic option is the use of a common domain model. This results in an architecture where multiple components collaborate on a common database that holds the whole model. Inspired by Richardson (Richardson, 2018), we call such big uniform models *monolithic models*. In our experience, such monolithic models quickly gain significant complexity and become a major obstacle to maintenance and evolution - curiously, these challenges led us originally to the usage of models in software engineering.

In this article, we propose to rethink the use of models in the software engineering process. In our vision, we move away from the demand of a holistic model to multiple *micromodels* that represent only a fragment of a complex system in the software engineering process. In detail, we propose Event Storming (Brandolini, 2013) to model the central workflows of our system based on a common *event model*, cf. Figure 1. This common event model may become pretty large (some thousand event types), thus we structure it into bounded contexts and group the events, accordingly, cf. Figure 3. Next, our process foresees to model the structure of the events, i.e. the data that each event transports cf. Figure 2. This event data model is then used as event schema for an event broker that is the core of the proposed event based architecture. Here, a generic event broker or messaging service, such as Kafka[1], may be used (cf. Figure 6). In addition, the grouping of events into bounded contexts may result in topics for the event broker. Event schema and topics facilitate a type safe communication.

Based on this event model and the bounded contexts, we derive *micromodels*. A micromodel uses event handlers to map events into an object oriented data model that allows to implement complex algorithmic tasks and e.g. supports special model queries. Based on these micromodels, we propose (web based) GUI components. The GUI components may use REST requests to retrieve relevant data from our micromodels and to present this data in dedicated views. Then the GUI components provide dialogues and interactions that enable their users to execute their tasks. Eventually, these tasks raise events to notify relevant

[1]https://kafka.apache.org/

Copei, S., Eickhoff, C., Malik, A., Nolte, N., Norbisrath, U., Sorgalla, J., Weber, J. and Zündorf, A.
From Monolithic Models to Agile Micromodels.
DOI: 10.5220/0010837200003119
In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2022)*, pages 227-233
ISBN: 978-989-758-550-0; ISSN: 2184-4348

micromodels and to trigger subsequent tasks.

The overall development process is iterative and agile, i.e. at any point in time, there may be multiple teams working on different features in different phases of the development lifecycle in parallel. Based on a continuous integration and deployment process, the whole system is constantly evolving.

The remainder of the paper is structured as follows. In Section 2 we present related work. Section 3 elaborates about the connection between Event Storming and micromodeling in detail. Section 4 especially addresses the concern for model evolution and, last, Section 5 concludes the paper.

2 RELATED WORK

Classical modeling approaches opt for a common enterprise wide or domain wide model. We call this a monolithic model. As example, the medical domain has a standardized domain model, the *Reference Information Model* (RIM) that is provided by the Health Level 7 (HL7) standard in version 3 (HL7,). The RIM is the basis for deriving even more complex models for processing and exchanging medical information. This complexity has been difficult and costly to implement. Thus, in version 4 of the HL7 standard, the RIM and its derived holistic models have been replaced by multiple smaller models, so called Fast Healthcare Interoperability Resources (FHIR). This relates closely to our idea of micromodels. However, HL7 is not yet using Event Storming as a means for modeling the main workflows and as start for the modeling and development process.

In industry, the derivation of suitable software architectures using Event Storming is an established practice. For example, Junker (Junker, 2021) describes a corresponding approach using Event Storming to derive service boundaries in a microservice architecture in the context of platforms in the healthcare sector. However, Junker does not refer to an event based architecture but her microservices exchange more complex documents which results in a more strong coupling of microservices.

Generally, the idea of using partitioned smaller models to address the problem of growing model complexity is a well established method. For example Scheidgen et al. (Scheidgen et al., 2012) describe a process to split models into smaller fragments to cope with transportation and persistence of large models. Also, the area of Collaborative Model-Driven Software Engineering (Di Ruscio et al., 2018) provides concepts to create smaller models to describe a larger complex software system. We support these efforts, but go a step further in our vision. Instead of reassembling smaller models, we think that within the software engineering process of modern applications it is not necessary to have a unified, large model of the system by using the proposed micromodel approach. In addition, the combination of micromodels with an Event Storming model is new.

3 EVENT STORMING AND MICROMODELING

In the following, we illustrate our approach using the example of a software application for a family doctor. As proposed, the modeling starts with an Event Storming workshop, cf. (Brandolini, 2013). During the Event Storming workshop a large group of domain experts and future users collects *domain events* using orange sticky notes that name relevant events in past tense, e.g. *patient registered*. A detailed Event Storming may contain additional sticky notes (using other colors) that may model *commands*, *policies*, *read models*, etc. cf. (Brandolini, 2013). cf. Figure 1.

Figure 2 shows a small cutout of an Event Storming for our medical example. We start with a *disease registered* event that adds a disease to our system. Then *test registered* events add tests that allow to identify symptoms. A *patient registered* event adds a patient and a *consultation registered* event protocols a consultation where the doctor does tests to identify symptoms that result in the diagnosis of a disease and a treatment.

Compared to business process models (BPM) (Recker et al., 2009), Event Storming models workflows in a very informal way. According to Brandolini the informal approach is a major advantage of the Event Storming model as it allows to involve domain experts and future users very easily. However, to turn an Event Storming into software we need to add more details. This shall be done by a requirements engineer or a software engineer. For the medical example some engineer refines the initial Event Storming notes by adding example data. We call this step *event modeling*, cf. Figure 9. The event model specifies the structure of the events of our system. It is important input for the implementation or configuration of the event broker used in our approach, cf. Figure 6. In Figure 2 we use a simplified YAML notation for our event modeling data. For example, the *disease registered* event shows a *name* attribute and a *symptoms* attribute that consists of a list of symptom names. Similarly, the *test registered* event has attributes for the *type* and for the symptom that the test clarifies and for the *price* of the test. Note, the data model for our events has a relatively simple overall structure: there are event types with primitive

Figure 1: Event Storming Big Picture (general example taken from (Brandolini, 2013)).

Figure 2: Event Storming Model for the medical domain (small cutout).

attributes only. Complex relations are not represented by associations between different event types. Instead our approach uses unique keys like a patient's name or some patient id, or the names of diseases and symptoms. While such keys create hidden relationships, Event Storming is tuned to deal with such hidden complexity by using a *ubiquitous language*, i.e. consistent naming of attributes throughout all events or at least throughout a *bounded context*, cf. (Brandolini, 2013), (Evans, 2004) and Figure 3. Still, Event Storming models may easily comprise several thousand events and become very large. Event Storming makes its overall model size manageable by grouping related events into so-called *bounded contexts* (Evans, 2004). According to Conway's law (Conway, 1968), in a larger enterprise a bounded context typically covers workflow parts that are internal e.g. to a certain department or organizational unit. By focusing on smaller organizational units, it becomes easier to come up with an appropriate Event Storming model and with consistent event data models that stick to a consistent ubiquitous language. Later on, most micromodels will only focus on a single or a few bounded contexts and thus deal with limited complexity. Domain events that hand over a workflow from one organizational unit to the next are exchanged between multiple bounded contexts. These higher level events need to be discussed at least between the participating organizational units and also between the teams in charge of the corresponding micromodels. In the implementation we also propose to use the bounded contexts as topics for the event broker. Such topics give structure to the event model and facilitate to identify the events relevant for a micromodel.

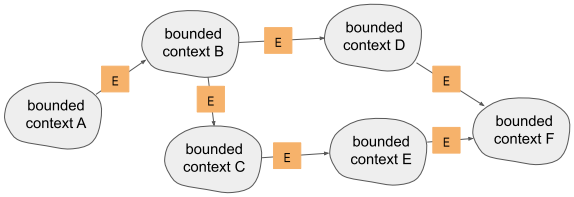

Figure 3: Bounded Contexts Model.

While Event Storming is effective for modeling workflows, such a workflow model is not well suited for certain complex algorithmic tasks. Thus, for complex algorithmic tasks or for complex data queries we propose e.g. object oriented micromodels. In the family doctor example, we use a *medical micromodel* for tests and symptoms and diseases, cf. Figure 4. The medical micromodel is designed to help the doctor during a consultation to identify the tests that may be used to clarify certain symptoms that indicate certain diseases. Figure 5 shows a cutout of an object diagram for our accounting micromodel.

Our approach uses an event based architecture, cf. (Richardson, 2018), cf. Figure 6. This means, the micromodels subscribe with the event broker as listener for the event types or topics they are interested in. For each interesting event type, the micromodel provides an *event handler*. On event arrival, the responsible event handler maps the event to the corresponding object structures (e.g. within a local database) that then may be queried e.g. by a GUI to show a list of tests or diseases to the family doctor. Similarly, an event may be raised by a GUI e.g. when the family doctor enters data during a consultation cf. Figure 6. Such user events may be sent to a local micromodel first and then the micromodel may forward these events to an event broker which forwards the event to other interested (i.e. subscribed) micromodels. For example our *disease registered* event created the *flu:Disease* object and the *Symptom* objects at the top of Figure 4. However, *disease registered* events are ignored by our accounting micromodel. The *test registered* event resulted in the *t101:Test* object in our medical model (at the top right of Figure 4) and in the *temperature:PriceItem* object within our accounting micromodel (in the middle of Figure 5). The *patient registered* event results in a *p42:Patient* object within both micromodels. However, note that the two micromodels use the *state* attribute in different manners. Within the medical micromodel, the *consultation registered* event results in a *c1337:Consultation* object that identifies a *medium flu* as diagnosis. In addition, the *state* attribute of the patient is set to *see again* as the doctor wants another consultation within some days. (We use the term “medium flu" in lieu of a precise clinical code to increase readability for the reader.)

Within the accounting micromodel the same event creates a *c1337:Invoice* object, that lists the prices for the used tests and the consultation and the diagnosis itself. In addition the accounting model sets the state of its *Patient* to *pending* as the invoice is not yet paid.

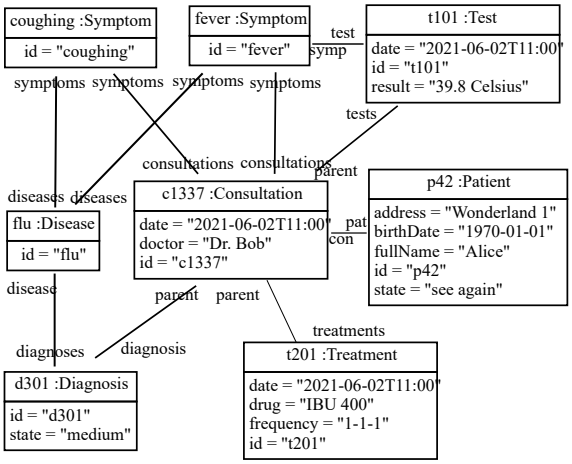

Figure 4: Medical Model.

Ideally, each micromodel deals only with a small set of related events from some bounded context(s). Each micromodel focuses on a certain logical aspect e.g. to support a certain user for a small number of workflow steps. Therefore, micromodels remain relatively simple and easy to maintain.

4 EVOLUTION

As discussed, we propose an agile and evolutionary modeling approach. The activities are grouped into three major workflows for events, micromodels, and GUIs, cf. Figure 9. Evolution usually starts within Event Storming activities. The developer may change some workflow part by adding new events, by extending existing events, or by deprecating events. While ensuring safe and consistent workflows is hard, the technical steps at the event level are quite simple. Within the event modeling activity the developer adds a description of the events’ data fields and extends the event schema of the event broker, accordingly.

Eventually, the new events need to be raised and handled by some micromodel(s) which requires to add e.g. new GUI dialogues and appropriate event handlers to the implementation of these micromodels. Adding a new event handler is relatively simple, as this is a quite local change that usually does not require to refactor large parts of the corresponding micromodel. As soon as everything is implemented, the application may start to raise the new events.

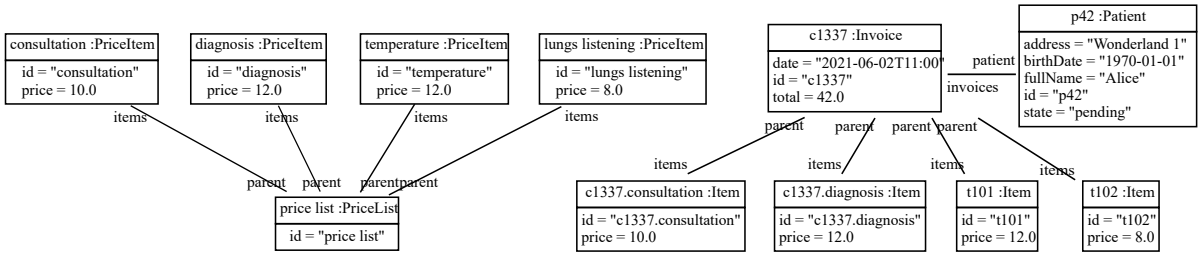

Figure 5: Accounting Model.

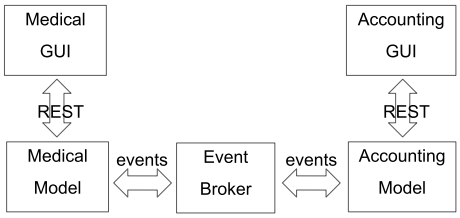

Figure 6: Micromodel Architecture.

For event deprecation, the workflows may stop to use the deprecated event as soon as the affected micromodels have implemented the GUI and the handlers for the new workflow events. As soon as the old events are no longer raised, the developer may remove the corresponding event handlers from the micromodels. However, the application may contain some micromodels that e.g. do statistics on the event history, e.g. mean waiting time, etc. Such parts may deal with old events for a longer time. Changing the data model of an already used event e.g. by adding additional attributes would require the event handlers to be updated, accordingly. However, there may be old micromodels that either still raise the (old) event or handle only the old events. All these micromodels need to be adapted to the new event structure. This will be done by different persons or teams at different speeds. Thus, during a certain period, we may need to deal with old and new versions of events in parallel. This may be achieved by adding version information to the event which eventually results in a new event type. Thus, we propose to handle changes to the data model of an event by creating a new event type (e.g. by adding a version post-fix to the event type like *disease registered v2* and by deprecating the old event type).

Another kind of evolution is a major change to the algorithmic requirements or to the model queries for one of our micromodels. Such a major change probably requires a refactoring or new modeling of the current data model and the adaption of all event handlers that contribute to the data model. In addition, a migration of existing data may be required before the new micromodel is deployed. For this kind of evolution, all the existing model evolution techniques may be used. As our architecture is event based, developers may also build (parts of) the new data model by re-executing some old events with the help of the new event handlers. If necessary, an explicit migration strategy can be added to the Event Storming / workflow description of the application, followed by the implementation, deployment, and execution of these workflow parts. This allows incorporation of some manual steps (e.g. re-enter old data) into the migration of your micromodel. This idea has e.g. been proposed by (Jacobson et al., 2013). Eventually, we claim that it is easier to migrate a small micromodel than to migrate a similar change within a complex monolithic model.

As an alternative for refactoring an old micromodel and for handling new events within existing micromodels, we can evolve our system by introducing a new micromodel that handles the new events and supports the new query model. Adding a new micromodel has the advantage that we do not need to deal with old code but we almost have a green field project. Actually, the new micromodel will also need to listen to some old events in order to incorporate information of old system parts. Identifying and incorporating relevant old events is facilitated e.g. through the bounded contexts of our Event Storming. In addition, the events are relatively simple, thus subscribing to some relevant event will hopefully not require to incorporate a large number of additional events that the relevant event depends on.

As an example consider that the family doctor wants to offer Covid-19 vaccinations in Germany (early in 2021) and thus the family doctor needs some software computing the priorities for patients that ask for a vaccination. The computation of a Covid-19 priority needs the age of the patient and some extra priority e.g. for people that work in health care jobs and some priority for people with certain diseases. To address these issues, in the example we introduce a *Covid-19 risk registered* event, that e.g. adds 2 points to the priority of patients with a flu, cf. Figure 7. In addition, we introduce a *Covid-19 vaccination requested* event that records the patient and optionally a job related extra priority, cf. Figure 7.

As shown in Figure 8, the new Covid-19 micromodel first handles *Covid-19 risk registered* events and

adds corresponding *Disease* objects to our Covid-19 micromodel. Similarly, an event handler for *Covid-19 vaccination requested* events adds patients to our Covid-19 model. As we need the age of patients, the Covid-19 micromodel also subscribes to *patient registered* events. To identify interested patients, we use a filter on *patient registered* events that returns patients that have requested a vaccination. Finally, we need the *consultation registered* events of interested patients to identify relevant diseases of these patients. This allows us to compute e.g. the priority of Alice as 51 years old plus 40 points for working in health care plus 2 points for having a flu resulting in 93 points.

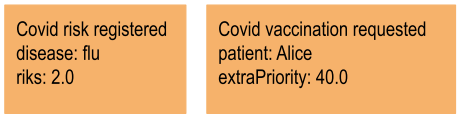

Figure 7: Covid Events.

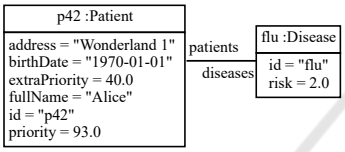

Figure 8: Covid Events and Model.

The important design decision here is that the Covid-19 micromodel relies only on local events. We might have been tempted to retrieve the diseases of a patient from the medical model. However, when micromodels use each other directly, we soon end up in a complex network of micromodels that is as hard to maintain and evolve as a monolithic model. Having the Covid-19 micromodel directly accessing the medical micromodel would create a dependency between these two models. Whenever we wanted to restructure the medical model, the Covid-19 micromodel would be affected. By basing the Covid-19 micromodel on events only, such dependencies are avoided. On the other hand, the Covid-19 micromodel now needs to identify the *consultation registered* event as the event relevant for the retrieval of diseases and to some extend the handler for *consultation registered* events repeats work already done in the medical micromodel. In order to identify the relevant events, the bounded contexts of the Event Storming provides us with guidance for our search. Similarly, the Covid-19 micromodel is interested only in the diagnosis of a *Consultation Registered* event, thus only little work is repeated.

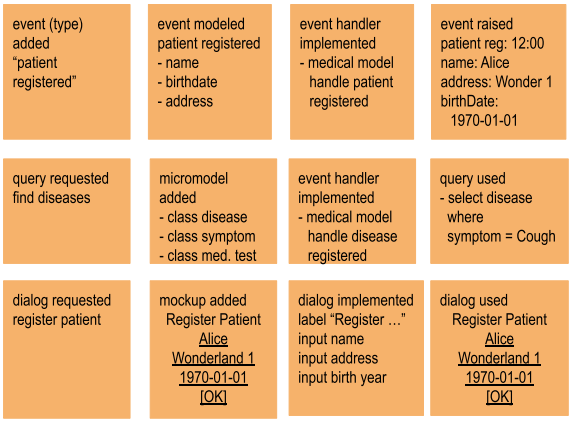

Figure 9: Modeling and Development Process.

5 CONCLUSIONS

Classical modeling uses e.g. a domain model that covers almost all aspects of a certain domain. Such a monolithic model soon becomes complex and hard to maintain. Splitting a large monolithic model into multiple event based micromodels results in a separation of concerns that facilitates modeling and evolution. For example, in a monolithic model, our Covid-19 priority computation would touch the single global model with all the details of symptoms, tests, consultations, and invoices. It only needs to know, whether the current patient has a common cold or the flu. It would then add a *priority* attribute to our global *patient* class which would likely collide with other priority attributes and would add more complexity to all other system parts and may require complex data migration.

Micromodels are dedicated for small purposes and thus avoid complexity and achieve flexibility. Note, micromodels frequently contain overlapping data, for example each of our micromodels has a *Patient* class. However, while these different patient classes have some common attributes, each patient class has micromodel-specific attributes that serve the needs of its respective micromodel. Within a micromodel such special attributes are introduced, easily, without cluttering a common monolithic model.

The basis for our micromodeling approach is Event Storming. It provides us with bounded contexts and a ubiquitous language, e.g. for events and their attributes. Event Storming also assigns clear ownership for events and the resulting model changes. This avoids a lot of merge conflicts usually related to distributed applications. Finally, our micromodels communicate via fine grained events rather than exchanging complex documents. The medical model raises *consultation registered* events instead of sending a complete list of

diseases, symptoms, tests, patients, and this week's consultations to our accounting service. This allows for agile development and evolution of event handlers.

Our co-author Adam Malik currently evaluates the Event Storming and micromodeling approach in an industrial project. Adam is the lead architect of a project in the domain of billing of health clinics in Germany. Using Event Storming helped indeed to capture institutional knowledge as well as deriving bounded contexts and identifying respective micromodels. So far, the micromodels enabled fast iterative development. We will report, whether the project keeps agile while it grows.

REFERENCES

Brambilla, M., Cabot, J., and Wimmer, M. (2017). Model-driven software engineering in practice, second edition. *Synthesis Lectures on Software Engineering*, 3(1):1–207.

Brandolini, A. (2013). Introducing event storming. *Available at: goo. gl/GMzzDv [Last accessed: 8 July 2017]*.

Conway, M. E. (1968). How do committees invent. *Datamation*, 14(4):28–31.

Di Ruscio, D., Franzago, M., Malavolta, I., and Muccini, H. (2018). Collaborative model-driven software engineering: A classification framework and a research map [extended abstract]. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 535–535.

Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

HL7. Health level 7 standard. http://www.hl7.org/.

Jacobson, I., Ng, P.-W., McMahon, P. E., Spence, I., and Lidman, S. (2013). *The essence of software Engineering: applying the SEMAT kernel*. Addison-Wesley.

Junker, A. (2021). Microservices as architectural style. In Glauner, P., Plugmann, P., and Lerzynski, G., editors, *Digitalization in Healthcare: Implementing Innovation and Artificial Intelligence*, Future of Business and Finance, pages 177–190. Springer, Cham.

Recker, J., Rosemann, M., Indulska, M., and Green, P. (2009). Business process modeling-a comparative analysis. *Journal of the association for information systems*, 10(4):1.

Richardson, C. (2018). *Microservices patterns*. Manning Publications Company,.

Scheidgen, M., Zubow, A., Fischer, J., and Kolbe, T. H. (2012). Automated and transparent model fragmentation for persisting large models. In *International Conference on Model Driven Engineering Languages and Systems*, pages 102–118. Springer.