Evolution of the E-Assessment Framework JACK

Michael Striewe, Björn Zurmaar, Michael Goedicke
{michael.striewe,bjoern.zurmaar,michael.goedicke}@paluno.uni-due.de

paluno - The Ruhr Institute for Software Technology
University of Duisburg-Essen

Abstract

This paper discusses the architecture of the e-assessment system JACK as a case study for planned and unplanned architecture evolution. It presents the original architecture and provides examples for both kinds of evolution.

1 Introduction and System History

The system JACK is a framework for e-assessment of complex exercises like programming, modeling, and mathematics. Its development started in early 2006 with a first version going live in late 2006. A sketch of the architecture and first plans for evolution have been published as technical report in 2008 [2], just before a major revision of the system was performed. The aim of this revision was to make the architecture more evolution friendly. An overview on the resulting system has been published in [1].[1]

This paper aims to provide a case study for evolution by presenting three sections: Section 2 gives an overview on the system architecture and some key design decision. Section 3 discusses evolution scenarios that were taken into account during the design, while section 4 reports on experiences with unplanned cases of architecture evolution.

2 Framework Overview

JACK is a server application and basically divided into two major parts: The core system including user interfaces, data storage, and short-running marking processes (called synchronous checks), as well as the backend system for long-running marking processes (called asynchronous checks). Each part is subdivided into several components (see figure 1). Particularly, the backend may use different components to provide different actual marking techniques, such as static or dynamic checks on programming exercises.

The main decision driver for this design was performance. The connection between core and backend is designed following the master-worker-principle, where one or more backend instances can handle the marking tasks provided by the core system. Hence the core system is not generally slowed down by a long queue of jobs waiting for completion. However, this only works for asynchronous checks, while synchronous checks still have to be performed on the core system.

A second decision driver was interface flexibility. The core system was designed to be able to receive solution submissions on different ways, such as submission via browser or upload via a plug-in for the Eclipse-IDE. At the same time, the core system also should be able to connect to different services to authenticate users.

The system is implemented in Java, using EJB as component model for the core system and OSGi as component model for the backend system. The core system including the web-frontend is deployed on an JBOSS application server, while the backend is deployed as a standalone OSGI application.

[1]The system is not available as open source system, but the source code of a recent version can be provided for research purposes upon request.

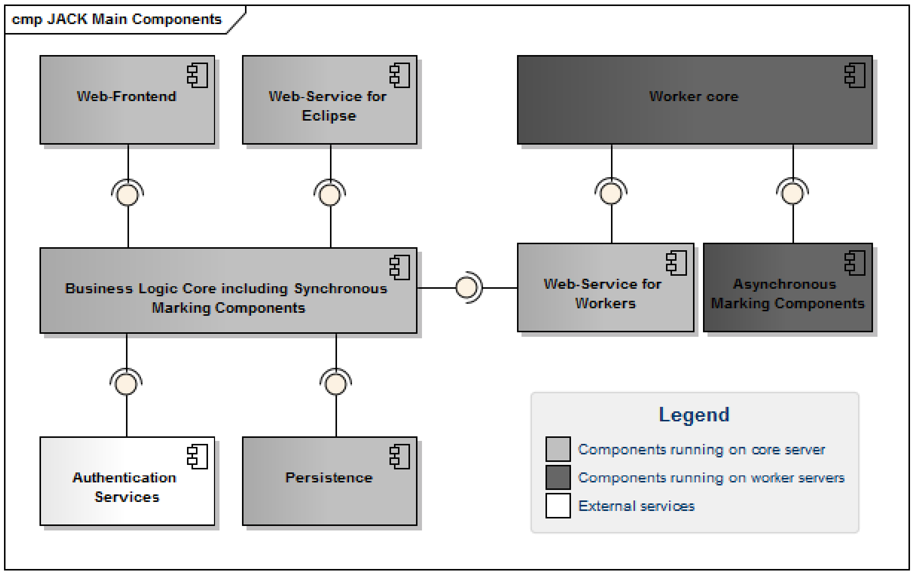

cmp JACK Main Components
Web-Frontend
Web-Service for Eclipse
Worker core
Business Logic Core including Synchronous Marking Components
Web-Service for Workers
Asynchronous Marking Components
Authentication Services
Persistence
Legend
Components running on core server
Components running on worker servers
External services

Figure 1: Component structure of JACK: The core system (light grey) is responsible for user interaction, data storage, and marking for exercises with immediate feedback (synchronous marking). The backend system (dark grey) is responsible for long-running marking processes (asynchronous marking) and can use an arbitrary number of actual marking components per backend system. In general, there is a many-to-many relationship between core systems and backend systems.

3 Planned Evolution

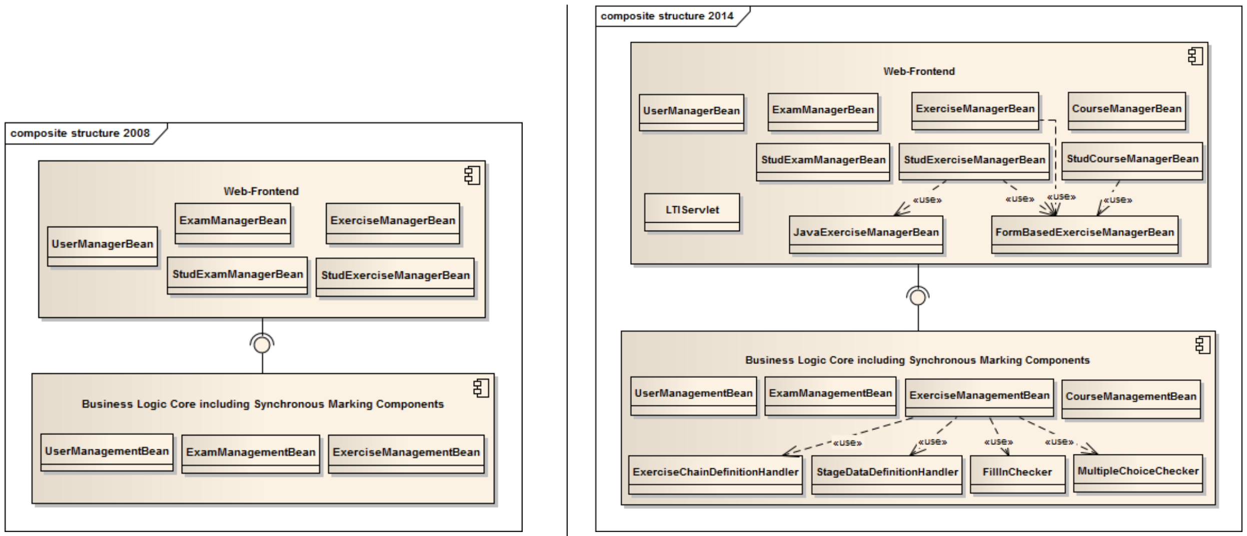
As the backend was designed to host different components for different types of marking techniques, it was planned that additional components will be developed later and have to be integrated into the system. This integration worked as expected and also allowed to integrate external systems into JACK by writing an appropriate component that serves as a broker. Since there was a clearly defined interface both between core system and backend system, and between backend system and marking components, it was also possible to extend these interfaces systematically to allow for more data to be transferred.

The original design included no actual components for synchronous marking, but only the concept of integrating such components into the core system. Work on the first component of this type was started in 2011 and the architectural concept turned out to be insufficient. Particularly, it was not possible to use the same interface as the one running on the backend system, as the latter uses some data types specific for the backend system. Hence it can be concluded that planning has failed with respect to this aspect, or that implementations have not realized all ideas from the original plans correctly.

4 Unplanned Evolution

While the integration of components for synchronous checks was planned from the beginning, it also brought some unplanned evolution with it. The original concept assumed that there is only few data that actually needs be handled by the core system, as marking happens in dedicated marking components. Consequently, the system employed one component for all exercise related low level tasks (`ExerciseManagementBean` in figure 2(a)) and two distinct components for the teacher oriented and student oriented user interface, respectively (`ExerciseManagerBean` and `StudExerciseManagerBean` in figure 2(a)). A similar set of sub-components was used for tasks related to exams and user management. It was supposed that this is a good and sustainable way to divide the system into sub-components.

However, soon after working with exercises with synchronous feedback we started to develop exercise types with flexible content that had to be generated by the core system. Thus it turned out that here is a strong need for several kinds of utilities for the internal representation of exercise content for some exercise types (`ExerciseChainDefinitionHandler` and `StageDataDefinitionHandler` in figure 2(b)). The new exercise types also influenced the design of the web-frontend in the way that additional components specialised to the needs of a particular exercise type were created (`JavaExerciseManagerBean` and `FormBasedExerciseManagerBean` in



(a) Sub-components structure of business logic and web-frontend as originally created in 2008.

(b) Sub-components structure of business logic and web-frontend after six years of software evolution.

Figure 2: Comparison of the original component structure with the one after six years of evolution. The original version shows a clear separation between components according to their scope (user-centric, exercises-centric, exam-centric) as well as a separation between the teacher-oriented and student-oriented part of the web-frontend. After six years, a third separation according to exercise types has been introduced which got more important than the others.

figure 2(b)). This was also used to avoid duplicated code in the teacher and student oriented parts of the web-frontend, as the separation between these parts in the component structure turned out to be less helpful. Finally, an additional way of handling collections of exercises was introduced, resulting in “courses" as an alternative way to group exercises in contrast to the already existing “exams". Hence another set of components had to be introduced into the web-frontend and the business logic core to handle that concept. As can be seen from the figures, the resulting architecture got populated with many components and there is no longer a clear rule what is grouped into one sub-component and what is placed in another one.

As a consequence we decided that we can achieve much better design by having one sub-component in the business logic core per exercise type, which handles all specific task for this type and is accompanied by one web-frontend component per exercise type both for the teacher and the student interface. This change will result in a fundamentally different component structure in the next major revision of the whole system.[2]

Another case of unplanned evolution was encountered with respect to authentication interfaces, although interface flexibility was one of the key factors in system design: In 2012, we were required to implement the IMS-LTI standard[3] to allow integration of JACK into e-learning platforms like Moodle. The solution involved creation of a new component to accept specific HTTP-requests (`LTIServlet` in figure 2(b)), integration of an OAuth library to decode authentication requests, and internal changes in the login module (`UserManagementBean`). While this were quite a lot of changes touching several parts of the architecture, they did not affect the architecture in any negative way. However, the consequence for future versions is to consider interfaces for HTTP-request right from the beginning for all places in which interaction with users or other systems can happen.

References

[1] Michael Striewe, Moritz Balz, and Michael Goedicke. A Flexible and Modular Software Architecture for Computer Aided Assessments and Automated Marking. In *Proceedings of the First International Conference on Computer Supported Education (CSEDU), 23 - 26 March 2009, Lisboa, Portugal*, volume 2, pages 54–61. INSTICC, 2009.

[2] Michael Striewe, Michael Goedicke, and Moritz Balz. Computer Aided Assessments and Programming Exercises with JACK. Technical Report 28, ICB, University of Duisburg-Essen, 2008.

[2]The component architecture planned for the next evolution step is available upon request for research.
[3]`http://www.imsglobal.org/lti/index.html`