UNIVERSITÄT
KOBLENZ · LANDAU
Institut für Softwaretechnik

FB 4
Informatik

The STOR Component System Interim Report

Kerstin Falkowski
Jürgen Ebert

Nr. 14/2009

Arbeitsberichte aus dem Fachbereich Informatik

Universität Koblenz-Landau • Campus Koblenz • Universitätsstraße 1 • 56070 Koblenz
Telefon +49 261 287-0 • http://www.uni-koblenz.de

Die Arbeitsberichte aus dem Fachbereich Informatik dienen der Darstellung vorläufiger Ergebnisse, die in der Regel noch für spätere Veröffentlichungen überarbeitet werden. Die Autoren sind deshalb für kritische Hinweise dankbar. Alle Rechte vorbehalten, insbesondere die der Übersetzung, des Nachdruckes, des Vortrags, der Entnahme von Abbildungen und Tabellen – auch bei nur auszugsweiser Verwertung.

The "Arbeitsberichte aus dem Fachbereich Informatik" comprise preliminary results which will usually be revised for subsequent publication. Critical comments are appreciated by the authors. All rights reserved. No part of this report may be reproduced by any means or translated.

Arbeitsberichte des Fachbereichs Informatik

ISSN (Print): 1864-0346
ISSN (Online): 1864-0850

Herausgeber / Edited by:

Der Dekan:
Prof. Dr. Zöbel

Die Professoren des Fachbereichs:
Prof. Dr. Bátori, Prof. Dr. Beckert, Prof. Dr. Burkhardt, Prof. Dr. Diller, Prof. Dr. Ebert, Prof. Dr. Furbach, Prof. Dr. Grimm, Prof. Dr. Hampe, Prof. Dr. Harbusch, Prof. Dr. Sure, Prof. Dr. Lämmel, Prof. Dr. Lautenbach, Prof. Dr. Müller, Prof. Dr. Oppermann, Prof. Dr. Paulus, Prof. Dr. Priese, Prof. Dr. Rosendahl, Prof. Dr. Schubert, Prof. Dr. Staab, Prof. Dr. Steigner, Prof. Dr. Troitzsch, Prof. Dr. von Kortzfleisch, Prof. Dr. Walsh, Prof. Dr. Wimmer, Prof. Dr. Zöbel

Kontaktdaten der Verfasser

Kerstin Falkowski, Jürgen Ebert
Institut für Softwaretechnik
Fachbereich Informatik
Universität Koblenz-Landau
Universitätsstraße 1
D-56070 Koblenz
EMail: falke@uni-koblenz.de, ebert@uni-koblenz.de

The STOR Component System Interim Report

Kerstin Falkowski, Jürgen Ebert

Contents

1 Introduction **2**

2 STOR foundations **3**
- 2.1 Java-related technologies . . . 3
- 2.2 Java-related component systems . . . 4
  - 2.2.1 JavaBeans . . . 5
  - 2.2.2 ConQAT . . . 7
- 2.3 Image processing libraries . . . 10
- 2.4 Graph processing API JGraLab . . . 10

3 STOR entities **11**
- 3.1 STOR Activities . . . 11
  - 3.1.1 Image processing . . . 12
  - 3.1.2 Feature processing . . . 12
  - 3.1.3 Model processing . . . 13
  - 3.1.4 Product-line view . . . 13
  - 3.1.5 Example: Domino tile recognition . . . 14
- 3.2 STOR Data Structures . . . 14
  - 3.2.1 Images . . . 17
  - 3.2.2 Features . . . 19
  - 3.2.3 Integrated models . . . 20
- 3.3 STOR Components . . . 21
  - 3.3.1 STOR components in general . . . 21
  - 3.3.2 STOR JavaBeans components . . . 23
  - 3.3.3 STOR ConQAT components . . . 25
  - 3.3.4 Domino tile recognition intermediary result images . . . 27
  - 3.3.5 Native library wrappers . . . 27

4 Conclusion and future work **29**

A STOR development environment **30**
- A.1 Folder structure . . . 30
- A.2 Build environment . . . 30

Abstract. The STOR project aims at the development of a scientific component system employing models and knowledge for object recognition in images. This interim report elaborates on the requirements for such a component system, structures the application area by identifying a large set of basic operations, and shows how a set of appropriate data structures and components can be derived. A small case studies exemplifies the approach.

1 Introduction

This paper reports on the interim results of the *software engineering part* of the project *Software techniques for object recognition (STOR)*[1] [2] whose goal is the development of a *Component-based Concept Employing Models and Knowledge for Object Recognition in Images and Image Sequences*. STOR is joint work with the Work Group Active Vision[3] of the University of Koblenz-Landau.

On the software engineering side, there are two main research areas tackled in STOR. The first one is the development of an adequate *component concept* for the area of object recognition. The second one is the development of an integrated *modeling approach* for this domain.

The work done in STOR is triggered by two different case studies. The first case study deals with the *recognition of domino tiles* in images. The second one concentrates on the *generation of urban object models* from images. The second case study has already been described in two publications focussing on schema development for integrated urban object models and on the generation of such models from images [11, 10]. This paper focusses on the component concept and uses the domino recognition case study for explanation.

Example: Domino tile recognition. The scenario addressed in this case study is the recognition of domino tiles in twodimensional images. All kinds of intensity images are given as input. The problem to be solved is to identify correctly as many tiles as precisely as possible.

This small case study gives rise to a large number of problems in object recognition including distorted and overlapping objects and calls for a broad range of experiments - using recognition algorithms ranging from concrete special purpose deterministic algorithms to general AI-based methods.

This report has two goals: firstly the interim results shall be recorded as a snapshot of the project status, and secondly it should serve as a short introduction to the software engineering part of the STOR project for new members of team.

Chapter 2 shortly introduces the *basic concepts and foundational work used* in STOR. Then, chapter 3 describes the different *entities developed in STOR* forming the base for a component concept (namely activities, data structures, components and native library wrappers). Finally, appendix A describes the *current development environment used*.

[1] This work is funded by Deutsche Forschungsgemeinschaft under grants PA 599/8-1 and EB 119/3-1.

[2] `http://er.uni-koblenz.de`

[3] `http://www.uni-koblenz-landau.de/koblenz/fb4/institute/icv/agpaulus`

2 STOR foundations

STOR aims at a *scientific component system* that can be used to experiment with different techniques and data structures for object recognition. A scientific component system supplies an experimental environment for a research area that can be employed to easily reuse and combine existing assets (components and/or data structures) to construct, analyze and evaluate solutions for exemplary problems. In the case of STOR, object recognition is the area to be explored.

Since the *Java language* supports many concepts needed for such an environment, it was decided to use Java for the front-end and some parts of the back-end and C++ for the image processing part of the back-end of the component system. Though such a two-language approach is quite unusual in the image recognition community, it has been chosen here since Java offers several important features (reflection, serializability, multi-threading) that would have to be implemented explicitly in C++ environments. The probable loss in efficiency seems to be tolerable, since (i) for image processing activities only the main coordinating code on a coarse-grained level is supposed to be written in Java while the image processing algorithms stay in C++, and (ii) real-time requirements are less urgent in experimental environments since quality of the results is primarily in focus.

Section 2.1 shortly cites the *Java-related technologies* that lead to this decision and are used in STOR. Section 2.2 introduces the background and terminology from the area of *component concepts*, and section 2.3 comments on the state of the art of *image processing libraries* for STOR. Section 2.4 contains pointers to the JGraLab API that is used for graph-based modeling and knowledge representation in STOR.

2.1 Java-related technologies

Java and *Java-related technologies* are used for the realization of the STOR component system. Those language features that have been added to Java are shortly introduced in the following for the reader's convenience.

Java assertions. An *assertion* [14] is a Java statement that enables testing of assumptions about programs at runtime. An assertion contains a boolean expression that is expected to evaluate to true during program execution. If it is false, the runtime system throws an error and terminates the program. Assertions are disabled by default, but they can globally be enabled by configuration of the Java runtime environment.

Java annotations. An *annotation* [14] is a Java language element that enables the integration of metadata in Java programs. Annotations can be interpreted during compile time by an annotation processor or at runtime via reflection. There are seven predefined annotation types, but one can also create domain-specific annotation types.

Java covariant return types. *Covariant overriding* is a Java language feature that helps to increase its type safety. Java prohibits to override a method by a new method in a derived class which possesses identical parameter and return types, but it allows to override a method by a new method in a derived class whose return type is proper a subtype of the overridden method's return type.

Java reflection. *Java reflection* [12] is a Java core concept and an appropriate API, that enables the access to and the restricted modification of class instances stored in the Java Virtual Machine (JVM) memory as well as the creation of new class instances at runtime. It is used in meta programming techniques like JavaBeans (Section 2.2.1), for example, if the existence and/or specification of a class is not known at compile time. In contrast to static Java programs, Java programs using reflection can be unsave, because there is no static type checking for class instances modified or created at runtime.

Java Native Interface. The *Java Native Interface (JNI)* [19] is a standardized concept and an appropriate API, that enables the communication between Java programs and native libaries written in other languages. It enables the use of native (e.g. C++) methods by a Java program as well as the execution of a full Java virtual machine or the use of Java methods via reflection from a native program. In contrast to plain Java programs, Java programs using native libraries via JNI are not necessarily platform independent anymore.

2.2 Java-related component systems

The STOR component concept will be based on existing component concepts or at least will work in accordance with them. The two most interesting Java-based component concepts for STOR and their related software are introduced in the following.

Three main phases can be identified for component-based development

- *component development*
  - consisting of component specification and component implementation - is the phase of identifying the relevant activities and constructing the corresponding components as building blocks;
- *component assembly*
  - consisting of component registration, component instantiation, component adaptation, and component composition - is the phase of constructing a solution from these building blocks in an assembly environment;
- *(composed) system execution*
  is the phase of executing the composed solution on sample data.

Scientific component systems shall support all three phases in a seamless manner and facilitate all kinds of analysis, profiling, and tracing.

Currently, STOR uses a *JavaBeans*-like approach which is incrementally refined and extended according to the needs of the project. For the first experiments, also the environment *ConQAT* was used. Both component approaches are shortly explained in the following and their terms are compared in Table 1.

General component concept terms	JavaBeans terms	ConQAT terms
component	bean	analysis
atomic component	bean	processor / analysis (step)
composed component (composit)	design	block / (composed) analysis / configuration
component without context	simple bean	/
component with context	nested/participant bean	/
component container	.jar-archive	bundle
program	design	(composed) analysis / configuration
input datum: parameter	property	input datum
input datum: argument	/	input datum
output datum	property	output datum
service	any public method	process-method
event	event	/
assembly environment	builder tool	editor

Table 1: Component concept term comparison.

2.2.1 JavaBeans

JavaBeans[4] is a Java-based component concept and an appropriate API of Sun Microsystems. The JavaBeans specification [16, 4, 3, 5] describes requirements to beans and their development as well as JavaBeans-specific functionality. Most of this functionality is part of the Java core SDK (packages `java.beans` and `java.beans.beancontext`).[5]

A *bean* is defined as a "reusable software component that can be visually manipulated in builder tools" [16]. Although such builder tools, called *assembly environments* in this report, are not part of the Java programming language, there are reference implementations from Sun Microsystems as well as further proprietary tools for that purpose. Unfortunately, most of them are only "visual" in the sense that they visually support the adaptation of beans. A visual support for composition is not supplied or can only be achieved with some additional (manually programmed) connection code.

The bean concept was originally developed for the simplification of graphical user interface (GUI) development. In contrast to the received opinion, a JavaBean does not need to have a GUI representation. According to the specification a bean can be either *visible*, *invisible* or *both* [16]. But most JavaBeans applications use visible beans.[6]

JavaBean components. Structure and behavior of a bean are described by the set of *properties* it possesses, the set of *services* it offers, and the set of *events* it fires. Technically a bean is represented by a single Java class fulfilling some conventions, so that structure and behavior can automatically be examined and used via reflection (Section 2.1).

A bean class must offer a public parameterless *constructor* so that any bean can automatically be instantiated at runtime. Furthermore it must implement the *interface* `java.io.Serializable` so that the internal state of any instantiated bean can be (de-)serialized at runtime.

A *property* is a private attribute that is accessible via a public get-method and potentially a public set-method if the property is changeable; both methods have to fulfill some *naming conventions*. If a bean is able to register other beans for receiving its *events*, the methods for (de-)registration also have to fulfill some naming conventions. All *services* of a bean are public methods, including those for property and event handling.

Beans that shall have a GUI representation have to *extend* `java.awt.Component` directly or via inheritance. To be usable by an assembly environment, the `.class`-file of a bean has to be packed into a *`.jar`-archive*. A `.jar`-archive can contain more than one bean and therefore it must include a *`.xml`-manifest-file* that explicitly identifies all contained beans.

Bean-API. The *Bean-API* (package `java.beans`) provides additional functionality for JavaBeans. Firstly, it contains a special *introspector* (`Introspector`) for bean examination via reflection (Section 2.1). Secondly, it offers classes providing further information about bean properties and their possible adaptation, namely *bean info classes* (`BeanInfo`), *property editors* (`PropertyEditor`) and *customizers* (`Customizer`). Moreover it offers an interface for the more specific definition of a bean's *visibility* (`Visibility`).

[4] `http://java.sun.com/beans`

[5] Only the *JavaBeans Activation Framework (JAF)* is a standard extension.

[6] For example the *Eclipse Visual Editor Project* `http://www.eclipse.org/vep` or *Matisse4MyEclipse* `http://www.myeclipseide.com/module-htmlpages-display-pid-5.html`.

BeanContext-API. The *BeanContext-API* (package `java.beans.beancontext`) provides additional functionality for JavaBeans packed in a `.jar`-archive. A `.jar`-archive can contain a specific *bean context* (`BeanContext`), that provides specific services for its contained beans (for example a printing or a debugging service). A bean can use a bean context, if it is a bean context child via implementing the interface `BeanContextChild`. A bean whose `.jar`-archive has no bean context or that even does not use this bean context is called *simple bean*; a bean that uses the context of its `.jar`-archive is called *nested bean* (or participant bean).

Bean assembly. Beans can be *adapted* by changing their properties and *composed* by connecting them via events. Since an event may contain additional data, this mechanism offers the possibility to pass data between beans. Both steps of bean assembly can be visually supported by an assembly environment or manually by a programmer.

Most existing assembly environments offer *visually supported adaptation* using the Bean-API functionality. But there seem to be different ideas about what *visually supported bean composition* means.

Normally event-based composition should be the *registration* of the target bean at the source bean. This can only be done, if the target bean is an event listener for a kind of event, that the source bean is able to fire and for which the source bean offers (de-)registration methods conforming to the *listener pattern*. In this case the connection can be done without additional connection code.

An assembly environment can examine properties, services and events of source and target beans via reflection in order to explore which event-based connections are possible. A mature visually supported bean composition environment should offer the possibility to connect beans by drawing an arc between source and target bean.

In this context, the ability of components to deliver all kinds of information about themselves is the key feature for the capability of the assembly environment with respect to analysis, profiling, and tracing.

Assembly environments. The *Bean Development Kit* (*BDK*, also called *BeanBox*) [26, 18] and its successor the *BeanBuilder*[7] are reference implementations for assembly environments. They are supplied by Sun Microsystems providing simple "test containers" for JavaBeans. Their official successor is *NetBeans*[8], an integrated development environment (IDE) – similar to Eclipse, that (among other facilities) can work as an assembly environment. *JBeanStudio*[9] [10] [24, 25] is a proprietary assembly environment from the Visualisation and High-Performance Computing Laboratory (ViSLAB), University of Sydney, hosted at Sourceforge.

NetBeans seems to be the only tool up to date, and therefore not all information could be detected for the other assembly environments. It was not possible to get running versions of BDK and BeanBuilder, but for both one can still find some documentation. Although there is a running version of JBeanStudio, there is no tool documentation at all.

All mentioned tools are *Java-based*. From the mentioned assembly environments only NetBeans can be uses for *bean development*. All of them offer the *registration* of existing beans (.jar-archives), the *instantiation* of registered beans via drag-and-drop into a GUI (an instance of `java.awt.Container`) and the visually supported *adaptation* of selected instantiated beans. The tools differ in their possibilities for visually supported bean *composition*, *system execution*, *persistent storage* of composed beans and reuse of stored systems as components. Except BDK all other assembly environments are able to handle invisible beans.

[7] `http://bean-builder.dev.java.net`
[8] `http://www.netbeans.org`
[9] `http://www.vislab.usyd.edu.au/moinwiki/JBeanStudio`
[10] `http://sourceforge.net/projects/jbeanstudio`

Table 2 compares the mentioned assembly environments and CqEdit (below). Information for the BDK and the BeanBuilder that could not be detected is marked with a questionmark. A screenshot of JBeanStudio can be seen in Figure 16.

assembly environment	BDK	BeanBuilder	NetBeans	JBeanStudio	CqEdit
currently up to date (version)	no	no	yes (6.5.1)	no	yes (2.0)
Java-based	yes	yes	yes	yes	yes
component development	no	no	yes	no	yes
visual component registration	yes	yes	yes	yes	yes
visual component instantiation	yes	yes	yes	yes	yes
visual component adaptation	yes	yes	yes	yes	yes
visual component composition (- programming)	?	?	no	yes	yes
visual component composition (+ programming)	?	?	yes	no	no
system execution	?	?	yes	yes	yes
persistent system storage	?	?	yes	yes	yes
reuse of stored system as component	?	?	no	no	yes
invisible bean handling	no	yes	yes	yes	/

Table 2: Assembly environment comparison.

2.2.2 ConQAT

The *Continuous Quality Assessment Toolkit (ConQAT)*[11] builds on a Java-based open source component concept and an appropriate IDE developed at of the Competence Center Software Maintenance, Technical University of Munich. ConQAT was originally developed as an *academic research prototype for a quality assessment tool* [8, 7]. It provides an environment for the development of quality analysis procedures and their composition (e.g. clone detection).

ConQAT is quite generic in the sense, that its component support can also be used for other application domains.

ConQAT has a *plugin architecture*, consisting of the *ConQAT core*, some *plugins* and the assembly environment *CqEdit*. There are different variants of ConQAT; this description is based on the current version 2.0 of the platform-independent Eclipse-based variant.

ConQAT components. ConQAT supports three kinds of active entities: *processors*, *blocks* and *bundles*.

A *ConQAT processor* is an atomic component. The authors claim that it is working similar to a mathematical function, that accepts an arbitrary number of inputs and produces a single output. Technically it is a public Java class implementing the interface `IConQATProcessor`, from which it inherits the *methods* `process` and `init` being annotated with the annotation `@AConQATProcessor` that provides a parameter `description`, and offering a parameterless public constructor. The `process`-method defines the output type of the processor. The method is also parameterless and its return type is originally `java.lang.Object`, but can be refined by any of its subclasses (covariant return type). The `init`-method has no return type but a parameter of type `IConQATProcessorInfo` that can provide information for a processor instance about itself (e.g. its instance name) and its environment (e.g. the current driver version) at runtime.

A processor can have *any number of input data* changeable from the outside, but there is no distinction between input data that affect the procedure of a component and those being processed by a component. For all input data a processor has to offer a *set-method*. A set-method has no return type and offers the changeable input data as its own parameters. It can change more than one input datum. A set-method must be annotated with the annotation `AConQATParameter` that provides the parameters `description` and `name` and every parameter must be annotated with the

[11]`http://conqat.cs.tum.edu`

annotation `AConQATAttribute` that provides the parameters `description`, `name` and optionally `defaultValue`. There are no ConQAT specific naming conventions for set-methods. Input data can be of arbitrary type (`java.lang.Object`). An input datum that is changed in-place can be annotated with the annotation `@APipelineSource`.

A *ConQAT block* is a component composed of at least one processor/block. The authors claim that it works similar to a mathematical function, but is able to produce *more than one output*. Technically it is a file in a special XML-format (suffix "`.cqb`") describing the contained processors/blocks and their interconnection in a declarative manner. A processor's implementation is referenced by the name of its Java `.class` file. The output of a source processor/block is referenced by its name with an "@" as prefix.

A *ConQAT bundle* is a container for processors/blocks that can be opened as *ConQAT bundle projects* in CqEdit or are referenced from other bundle projects. Technically it is a folder in the file system. The folder name is also the identifier used to reference the bundle, by convention it is the name of the top package used by the Java code it contains. A bundle folder has the following structure and contents:[12]

- `blocks`: subfolders and blocks
- `bundle.html`: bundle documentation main page
- `bundle.xml`: bundle description
- `classes`: subfolders and processors (`.class`)
- `lib`: libs used by processors
- `resources`: any resources
- `src`: subfolders and processors (`.java`)
- `test-data`: any test data
- `test-src`: any test sources

The file *bundle.xml* contains information about the bundle's name, its provider, a bundle description, the current bundle version, the required ConQAT version and the bundles, this bundle relies on. A bundle can have a *bundle context* (a class inherited from `BundleContextBase`), providing information about the bundle itself and the access to its resource folder. A bundle's context has to be placed in the top-level package of the bundle and is typically a singleton.

A *minimal executable ConQAT component* is a block that contains one processor where both are stored in the same bundle. The bundle can be loaded as a ConQAT bundle project in CqEdit where the block can be executed calling its processor instance internally.

Component assembly. Using ConQAT, *component assembly* is the equivalent to block development. It can be done *manually* in a `.cqb`-file or *visually supported* using the assembly environment CqEdit.

Processors and/or blocks can be *composed to a new block* in a pipes-and-filter style that determines the program data flow. They can be interconnected by taking an output datum of a source processor/block as input datum for a target processor/blocks.

Assembly environment. *CqEdit* is the Eclipse-based assembly environment of the ConQAT IDE, offering support for all important component life cycle activities. Table 2 lists these activities and compares CqEdit to the JavaBean assembly environments. A screenshot of CqEdit can be seen in Figure 18.

CqEdit supports the *import* of an existing bundle project or the *creation* of a new bundle project [New → Other → cq.edit → Create Bundle Project]. Opened bundles are listed in the `Project Explorer` view. Inside an opened bundle one can *develop* new processors/blocks. For *processor development* a *JRE* and the archive `conqat.jar` have to be included as libs into the bundle-projects buildpath.[13].

[12]The `src`, `test-data` and `test-src` folders may only be required during processor development.
[13]The archive is in the Eclipse folder: `/plugins/edu.tum.cs.cqedit.core_2.0.0/conqat.jar`

Then a processor class with the above described properties can be created. As mentioned before *block development* complies with component assembly. One can *create* a new empty block [New → Other → cq.edit → New ConQAT Block][14], *add* instances of integrated processors/blocks to it, and *adapt and compose* them.

When the first block is opened with the `Block editor` after opening a bundle project, the *ConQAT driver* loads instances of all processor classes via reflection (Section 2.1) and lists all integrated processors and blocks in the `ConQAT Library` view. Then CqEdit offers the (multiple) instantiation of processors/blocks by dragging them from the `ConQAT Library` view to a block opened in the `ConQAT block editor`. There, instantiated processors/block are visualized as rectangles with rounded corners and ports for input and output data. After its instantiation a processor has one output value port and no input value port.

CqEdit offers the *adaptation* of focussed instantiated processors/blocks by changing their names as well as by adding, changing and deleting their input data ports in the `Properties` view. An input datum can be instantiated multiple times for the same processor. Moreover it offers the visually supported *composition* of instantiated processors/blocks in the block editor by drawing an arc from the output port of a source processor/block to an input port of a target processor/block, provided the ports have matching types. This type checking is done using the loaded processor classes mentioned before based on the Java type system.

Lastly, CqEdit offers the *execution* of a focussed block via the (context) menu [Run as → ConQAT Analysis]. During runtime the *ConQAT driver* computes a topological execution order for the processors, calls them one after another in this order and passes the data between them. If an output datum is used more than once, the driver is responsible for cloning it. For every processor it collects all required input values, instantiates the processor using the parameterless constructor, calls the `init`-method, calls the `set`-methods, calls the `process`-method and stores the result.

JavaBeans versus ConQAT

In this section the *similarities and differences* between the two component concepts JavaBeans and ConQAT are shortly compared.

Both component concepts have special requirements, that Java classes have to observe to become a component. A JavaBeans component has to *inherit the interface Serializable* and has to *fullfill a lot of naming conventions*. A ConQAT component has to *inherit a component base class* and has to *use annotations*. ConQATs requirements seem to be tougher than those from JavaBeans.

The JavaBeans component concept is `pure Java-based` but there seems to be no possibility to store a composed component so that it can be used as atomic component later on. Using ConQAT it is possible to use a composed component as atomic component, but this requires the *use of XML* beside Java to create and store composed components.

For JavaBeans there are *some assembly environments*, but there seems to be a big interpretation scope for the JavaBeans requirements and only the JBeanStudio supports nearly all component life cycle activities. For ConQAT there is *only the one assembly environment CqEdit*, but this one supports all important component life cycle activities.

Both component concepts allow the assembly of components using a *data flow metapher*. Using JavaBeans one has to implement the data transfer *indirectly via registration possibilities and events*. Using ConQAT one can *directly connect the output port of a component with the input port of another component* and the ConQAT core realizes the data exchange.

[14]Block creation functions only, if the focus is on the block folder of an opened *ConQAT bundle project*.

2.3 Image processing libraries

Since the objective of STOR-components is object recognition in images, there is definitely a need for image processing. There are a lot of *image processing libraries* available, two of them being developed at the University of Koblenz-Landau.

Haas [15] explored the state of the art for image processing libraries in the context of STOR. She inspected fifteen different libraries, selected the four most suitable ones, described them in detail from a software engineering point of view and evaluated them using the domino tile recognition case study. Three of them are shortly described in the following.

Haas concluded that all good image processing libraries are written using C/C++. In STOR they are integrated into a Java-based component concept using JNI (Section 2.1 and 3.3.5). According to Haas, OpenCV (Section 2.3) seems to be the image processing library best-suited for STOR. Therefore, at present OpenCV is the default library for STOR image processing activity components (Section 3.1.1).

OpenCV. The *Open Computer Vision Libary (OpenCV)*[15] [2, 15] is a widely-used API offering a comprehensive collection of algorithms, data structures and example programs for realtime image processing and computer vision. It is developed in C/C++ and available for Mac OS X, Linux and Windows, but only for 32 bit systems. OpenCV was foremost developed by Intel. Since 2008 it is an open source project supported by Willow Garage and hosted at SourceForge. The following descriptions and developments are based on the OpenCV pre-version 1.1 of October 2008.

PUMA. The *Programmierumgebung für die Musteranalyse (PUMA)*[16] [21, 22, 23, 15] is an internal API of the *Work Group Active Vision*, University of Koblenz-Landau, offering algorithms, data structures and small executable programs for pattern recognition, image processing and computer vision. It is developed in C/C++, based on *Koblenz-NIHCL (KONIHCL)* a variant of the *National Institute of Health Class Library (NIHCL)* [13] and available for Mac OS X, Linux and Windows. PUMA was mainly developed by Dietrich Paulus and some staff and students since 1992 at the University of Erlangen-Nürnberg and since 2002 at the University of Koblenz-Landau.

KIPL. The *Koblenzer Image Processing Library (KIPL)* [23, 15] is an internal API of the *Image Recognition Laboratory*, University of Koblenz-Landau, offering algorithms and data structures for image processing and object recognition. It is developed in C/C++ and is available for Mac OS X, Linux and Windows. KIPL was developed by Frank Schmitt and Patrick Sturm and some students under the direction of Lutz Priese. KIPL is used in a `Plugin-Framework` [1] developed by Dirk Balthasar, that encapsulates KIPL's algorithms so that they can be activated in several frontends, for example an own plugin environment or the open source software GIMP[17].

2.4 Graph processing API JGraLab

The *Java Graph Laboratory (JGraLab)*[18] is a *Java class library* developed by the *Work Group Software Technology*, University of Koblenz-Landau, offering a highly efficient API for the processing of TGraphs. *TGraphs* are directed graphs whose vertices and edges are typed, ordered and attributed. TGraphs are supported by JGraLab in combination with a *graph query language* (GReQL) and a corresponding

[15] `http://opencv.willowgarage.com` and `http://sourceforge.net/projects/opencvlibrary`
[16] English: "Programming environment for pattern recognition", `http://www.uni-koblenz-landau.de/koblenz/fb4/institute/icv/agpaulus/agas-projekte/puma`
[17] `http://www.gimp.org`
[18] http://jgralab.uni-koblenz.de

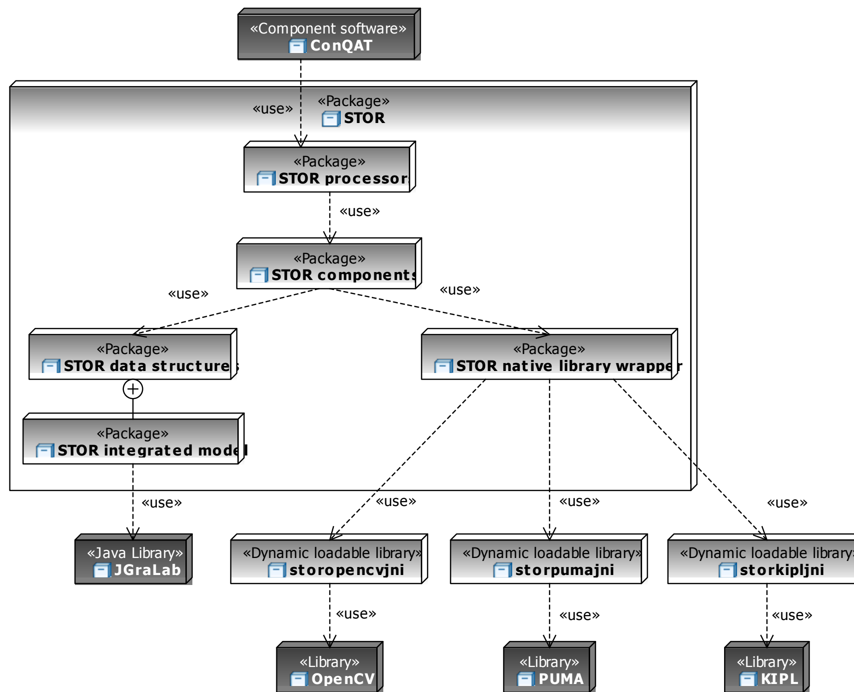


Figure 1: STOR entities. Internal entities are colored gray with a black font, external entities are colored dark gray with a white font.

UML-based *metamodeling approach* (grUML). grUML is a subset of UML class diagrams which allows the specification of classes of TGraphs on the schema level. [9]

3 STOR entities

This chapter describes different *STOR entities* forming the base for the STOR component concept. Section 3.1 gives an overview on the main activities that have to be carried out for model-based object recognition in images and thereby introduces the field. These activities form the development base for the components. Section 3.2 describes the data structures used by these activities, and Section 3.3 introduces the components derived from the activities and processing the data structures. Subsection 3.3.5 introduces the wrapper concept used to integrate native library functionality from image processing libraries into the components. Figure 1 gives an overview on the developed entities and their relationships to each other as well as to some of the used libraries.

3.1 STOR Activities

This section gives an overview on the *activities* needed for model-based object recognition in images. It focusses on the *domino tile case study*. The activities described here are candidates for components that may be assembled to form a solution for a given problem (Section 3.3). The recognition process is explained top down and visualized using *UML 2.0 activity diagrams*, which describe the dataflow between activities. The basic activities appear as *actions* in these diagrams. The types of the input and output data are appended to the ports of the actions, extended by a list of state constants that give even more detailed information about the data. These states are written in brackets, if there is more than on possible state at one time, the alternatives are separated using a comma.

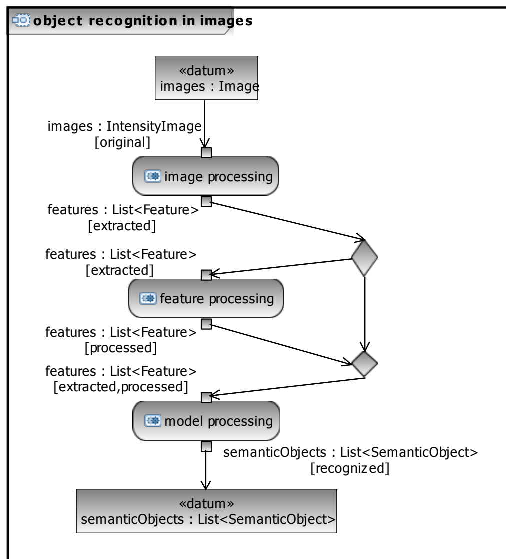


Figure 2: Object recognition in images.

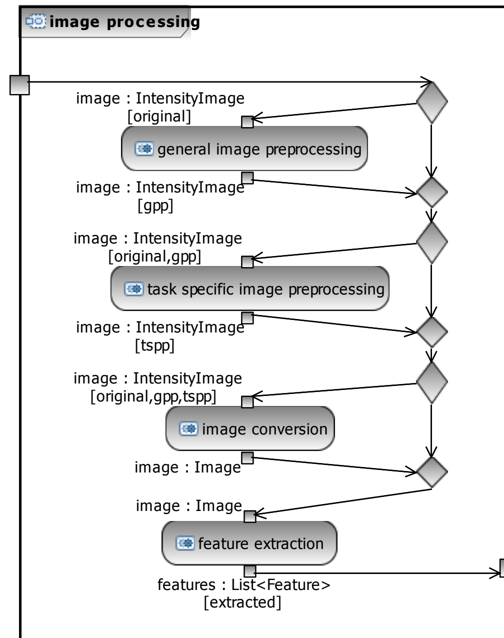


Figure 3: Image processing.

Model-based object recognition in images comprises *three phases* (Figure 2). Firstly, one has to *process a given input image* to extract relevant features from it. Then, one can *process the features* to prepare them for insertion into a model. Lastly, one has to *insert the features into the model* such that recognition of objects is possible by processing the model. This leads to three main activities which are further explained in the following: *image processing* (Section 3.1.1), *feature processing* (Section 3.1.2) and *model processing* (Section 3.1.3).

3.1.1 Image processing

Image processing deals with all activities performed on images (Section 3.2.1). It can be split into four sub-activities (Figure 3). First of all one can *generally preprocess the image* (gpp) to correct errors typically evolving during image creation and/or loading. Furthermore one can *specifically preprocess the image with respect to the actual task* (tspp). Then one can *convert the image* into a form suitable for the specific task, and at last one has to *extract the relevant features* from the image.

In the domino tile case study the specific task is *extraction* of rectangles and circles. Typical preprocessing tasks for feature extraction are *noise reduction* and *edge reinforcement*, typical image types for feature extraction are *binary images* and/or *edge images*, but any other image type could be used as well.

Besides the mentioned activities there are further *management activities* concerning images (for simplification not visualized in the diagram). Before an image can be used, it has to be *loaded* from a file and *converted* into an internal format, or it might even be *generated* from other sources. Moreover an (intermediary) result image might be *visualized* and/or *stored persistently*.

3.1.2 Feature processing

A `feature` (Section 3.2.2) is a piece of information that can be extracted from an image. Examples are points, lines, or circles, but also other kinds of characteristic data. *Feature processing* deals with

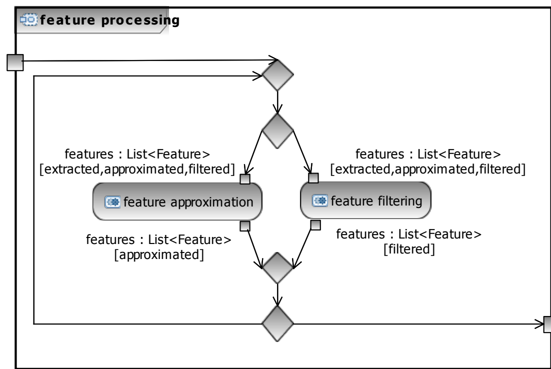


Figure 4: Feature processing.

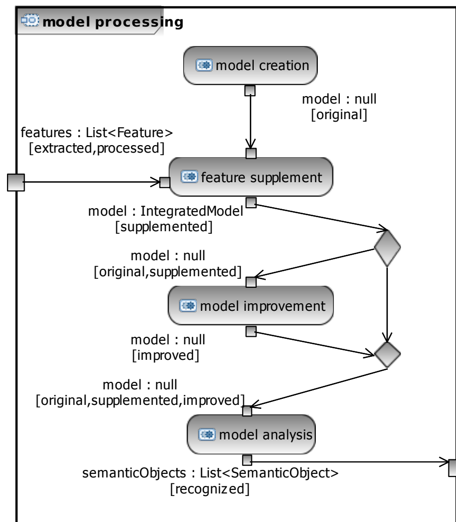


Figure 5: Model processing.

all activities performed on the features extracted from images. It can be split into two different sub-activities (Figure 4).

There are features that can be *approximated by more special features* and there are sets of features where one can *filter features with specific properties* from. Both activities can be carried out more than once and in any order, highly depending on the concrete features.

3.1.3 Model processing

All problem-relevant knowledge including relevant parts of the extracted features are stored in an *integrated model* (Section 3.2.3). This model is a problem-specific TGraph (Section 2.4). *Model processing* deals with all activities that are performed on such an integrated model (Section 3.2.3). It can be split into four different sub-activities (Figure 5).

First of all one has to *create* a model. This can be done by *loading* an existing model from a file or by *generating* a model. Afterwards one has to *supplement* the model with the features extracted from the image. Furthermore one can *improve* the model, where we distinguish between *topology improvement*, *geometry improvement* and/or *semantics improvement*. At last one has to *analyze* the model by an analyzer which is specific for the object(s) to be recognized.

Besides the mentioned activities there are also further management activities concerning models (for simplification reasons not visualized in the diagram). Although the results of model-based object recognition in images are the recognized objects, the model used for object recognition can also be *stored persistently* (again) in different formats for further experiments.

Further model processing activities concerning the *urban object model generation case study* can be found in [10].

3.1.4 Product-line view

The exemplary overview on object recognition activities given in the Sections 3.1.1-3.1.3 can also be viewed as a sketch of a *software product line*. The activity diagrams contain the basic actions and all non-refined activities may be implemented using concretely implemented software components.

Figure 6 visualizes the mentioned activities as well as further refinements and specifications in the form of a *feature*[19] *diagram* [6]. Here, for instance, image creation, image visualization and image storage are subsumed by the superfeature *image management*, and similarly, general and task specific image preprocessing are combined to a common feature *image preprocessing*.

In contrast to an activity diagram, a feature diagram shows a static view to the activities. Therefore, it can work as a pattern for a package hierarchy for the components derived from the identified activities (Section 3.3).

3.1.5 Example: Domino tile recognition

As an example, we show a recognition process for the domino tile case study - containing a choice of the before-mentioned activities in a specific order. Using the reference feature diagram of Figure 6 a specialized solution for the domino tile recognition problem is given in Figure 7. The corresponding dataflow is displayed in Figure 8 which visualizes the whole action as *UML 2.0 activity diagram*.

For domino recognition in images one can carry out the following activities. First of all one can *open an existing intensity image* with an image reader, for example the `AWT image reader`. After that one can *task-specifically preprocess the image*, for example using the `OpenCV Median operator` to reduce noise. Then one can *convert the intensity image into a binary* image with an image converter like the `OpenCV intensity to binary converter`. Afterwards one can *extract contours from the binary image* with the `OpenCV contour extractor`, *approximate these contours by polygons* using the `OpenCV polygon approximator` and *filter rectangles from these polygons* using the `OpenCV rectangle filter`. Concurrently one can *filter circles from the extracted contours* using the `OpenCV circle filter`. Following one can *generate an initial integrated model* with the `Initial model generator` and *supplement the extracted rectangles and circles* into the initial integrated model using the `Face supplementor`. Afterwards one can *improve the integrated model* by supplementing topological relations between the supplemented rectangles and circles using the `Topology supplementor`. Then on can *add domino tiles* and their semantical parts for every existing rectangle in the model using the `Domino tile supplementor`. At last on can *analyse the model and return the contained domino tiles* as semantic objects using the `Domino analyser`.

The solution has been implemented and assembled by ConQAT processors in the *STOR ConQAT component variant* (Section 3.3.3).

3.2 STOR Data Structures

The different activities for image recognition work on several data, like images, features, and the integrated model. This section introduces these *data structures* for model-based object recognition in images that have been developed in STOR. The data structures are visualized using *UML 2.0 class diagrams* which were extracted directly from the source code.

A *STOR data structure* consists of a *specification* noted as a Java interface and an *implementation* given by a non-abstract Java class. All STOR data structure interfaces extend the general interface `STORDataStructure`. All interfaces are usually implemented more than once, i.e. the interface works as a specification for different variants of a data structure with the same semantics but different non-functional properties. A STOR data structure class implements at least one STOR data structure interface. It must have at least *one public constructor*. In general the name of a data structure class is the name of its interface extended with the suffix `Impl`.

The specification and implementation of the STOR data structures develops in line with the specification of the activities for model-based object recognition in images (Section 3.1). Currently, there are three important groups of data structures, which are explained in the following: *images* (Section 3.2.1),

[19]Here, the term feature refers to a distinguishing characteristic of a software product line. It must not be confused with the term feature used in image recognition

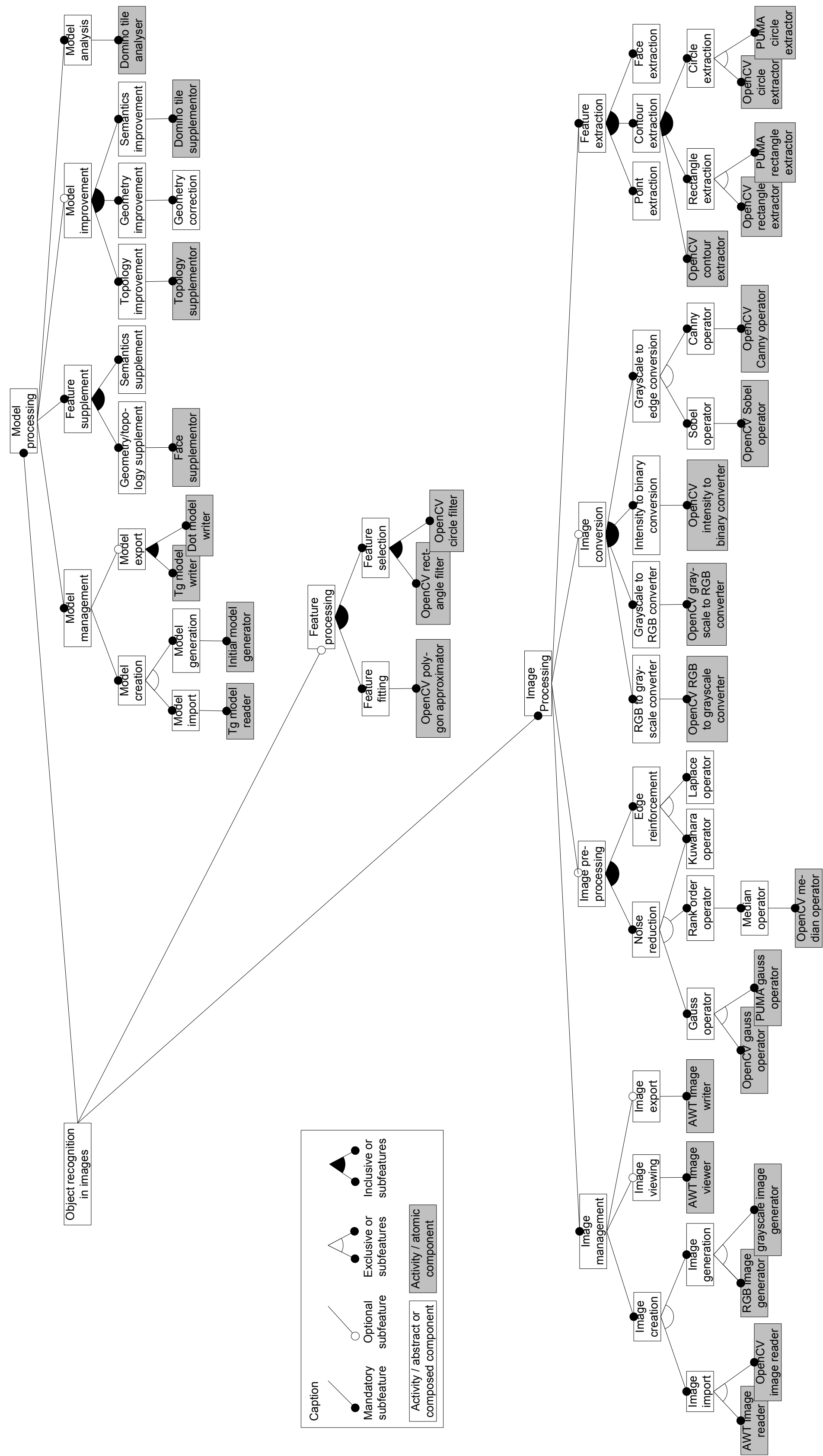


Figure 6: Feature diagram for STOR.

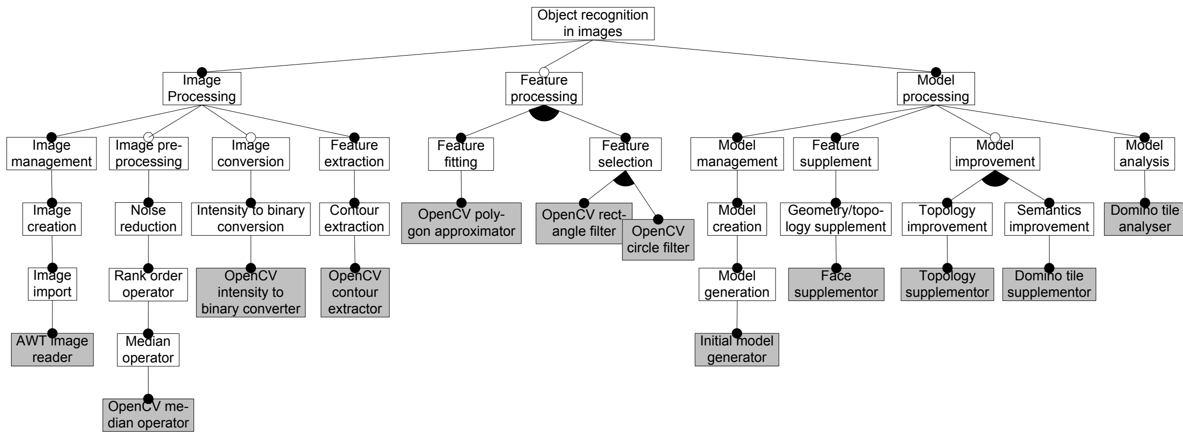


Figure 7: Domino tile recognition case study features.

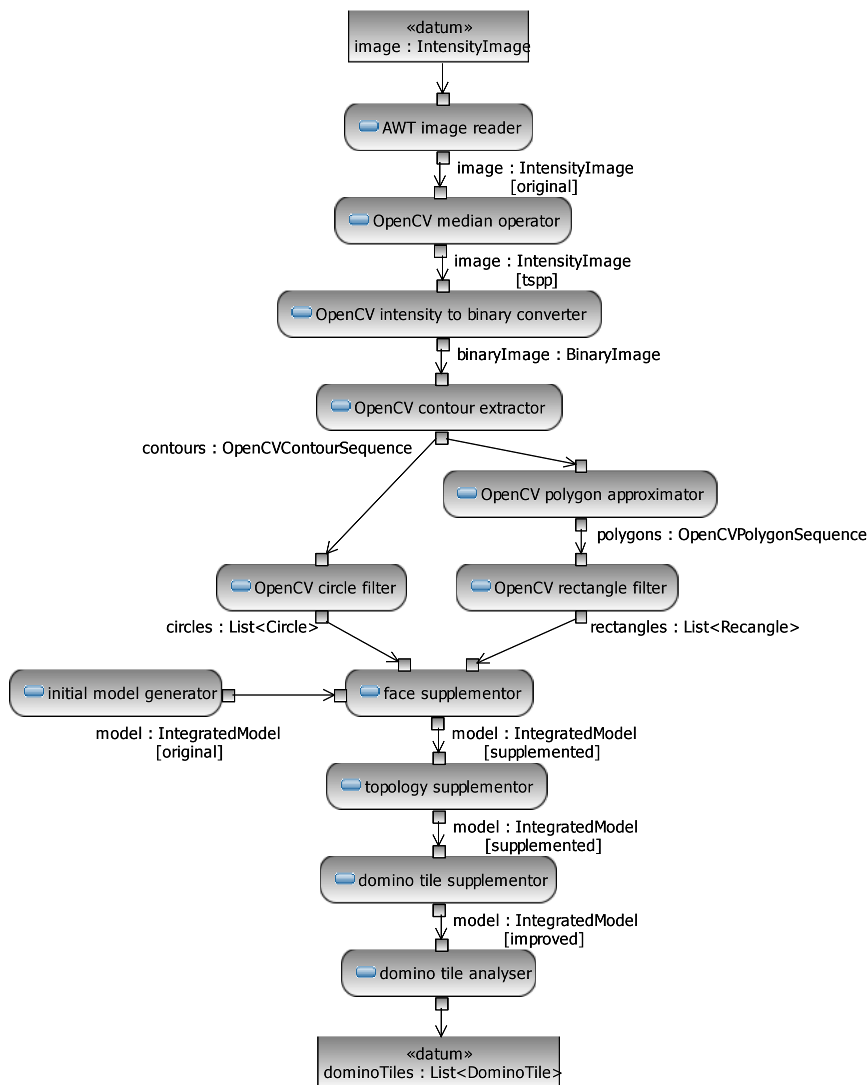


Figure 8: Domino tile recognition case study activities.

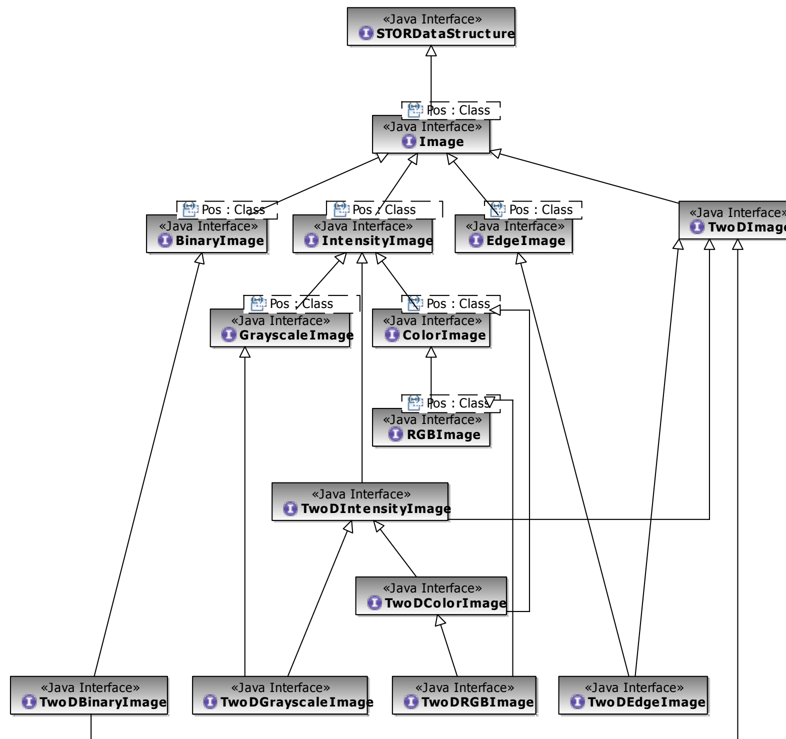


Figure 9: STOR image hierarchy.

features (Section 3.2.2) and *integrated models* (Section 3.2.3). Except *native objects* (a subclass of features), all other data structures are *serializable* and therefore implement the Java interface `Serializable`.

All *data structure interfaces* are collected in the package `de.uni_koblenz.stor.datastructures` and its subpackages. The corresponding *data structure classes* are kept in appropriate sub-packages with suffix `.impl`.

There is one kind of data structure that can occur in two different variants: *geometric objects* (a subgroup of features). It can occur as *independent data structures* and/or as *parts of an integrated model*. Inside an integrated model it is represented as a *graph node* or even split into different parts and represented as a *set of related graph nodes*, that can be related to other nodes. Outside an integrated model a geometric object is completely independent from other data structures, or contains only attributes from other data structures types.

3.2.1 Images

An *image* is defined as a mapping from a domain to a range. In STOR the *2d domain* and the *four ranges binary*, *grayscale*, *RGB* and *edge* are supported. Thus, at present there are four different kinds of images: *2d binary images*, *2d grayscale images*, *2d RGB images* and *2d edge images*. But the image data structures are flexible enough so that any domains or ranges can easily be added.

Figure 9 shows the *STOR image hierarchy*. One can see the strict separation of *domains* and *ranges* at the higher hierarchy levels. Every possible domain and every possible range has its own interface. In lower hierarchy levels a domain and a range are combined to a *concrete image type*, where the range is parameterized with a position `Pos` of the same dimension as the domain. In this way one can easily add domains and/or a ranges and combine them with existing ones to create new concrete image types. Examples could be a *3d domain* and/or a *HSV range*. Moreover one can see that some

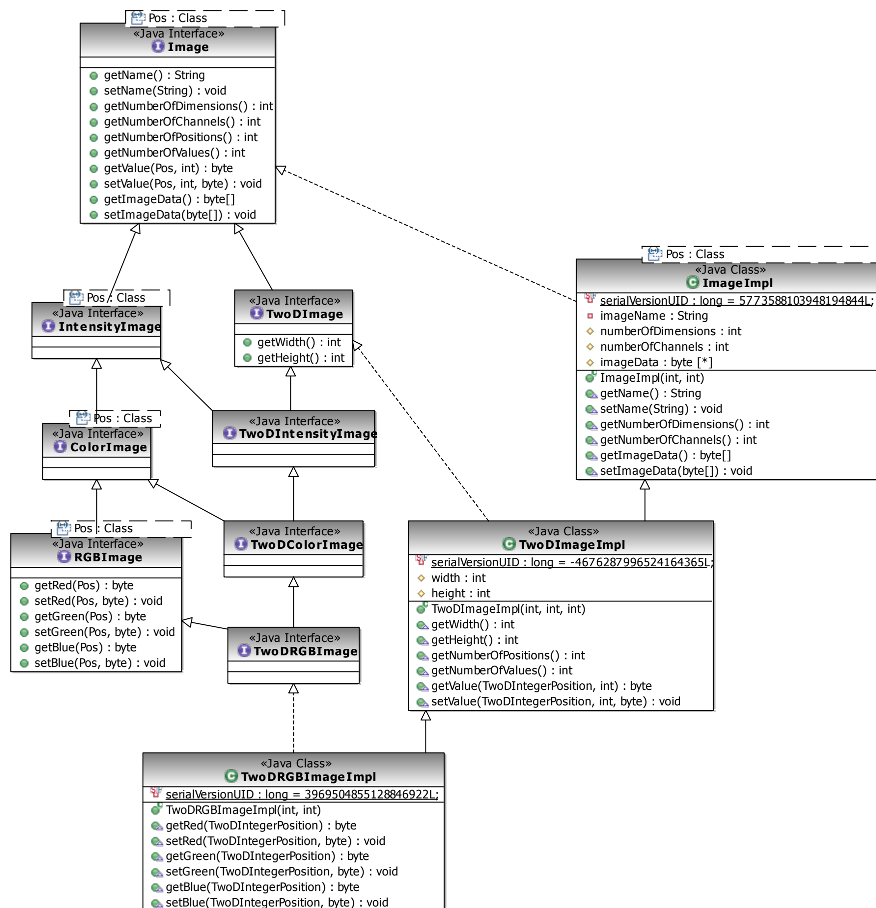


Figure 10: STOR 2d RGB image.

interfaces are only added for grouping concrete image types: `RGBImage` is a subclass of `ColorImage`, `GrayscaleImage` and `ColorImage` are subclasses of `IntensityImage`.

Note, that this elaborate hierarchy significantly helps to structure the recognition process. Since native data are used in the respective image processing libraries, this elaborate typing does not lead to additional efficiency problems.

As an example the *2d RGB image* is explained in detail (Figure 10). Every `Image` has the changeable property `name` and the read-only properties `numberOfDimensions`, `numberOfChannels`, `numberOfPositions` and `numberOfValues`. Moreover it has get- and set-methods for a single image value as well for all image values. Every `TwoDImage` has the read-only properties `width` and `height`. Every `RGBImage` has get- and set-methods for a single red, green or blue value. A 2d RGB image combines all mentioned properties and methods. The 2d RGB image implementation `TwoDRGBImageImpl` is derived from the two abstract classes `ImageImpl` and `TwoDImageImpl`, where some more general properties of (2d) images are implemented.

Currently, all image implementations store the *image values interleaved in a one-dimensional byte array*. This is a storage style which is well-suited for the exchange of image data between Java and native C++-libraries (Section 3.3.5).

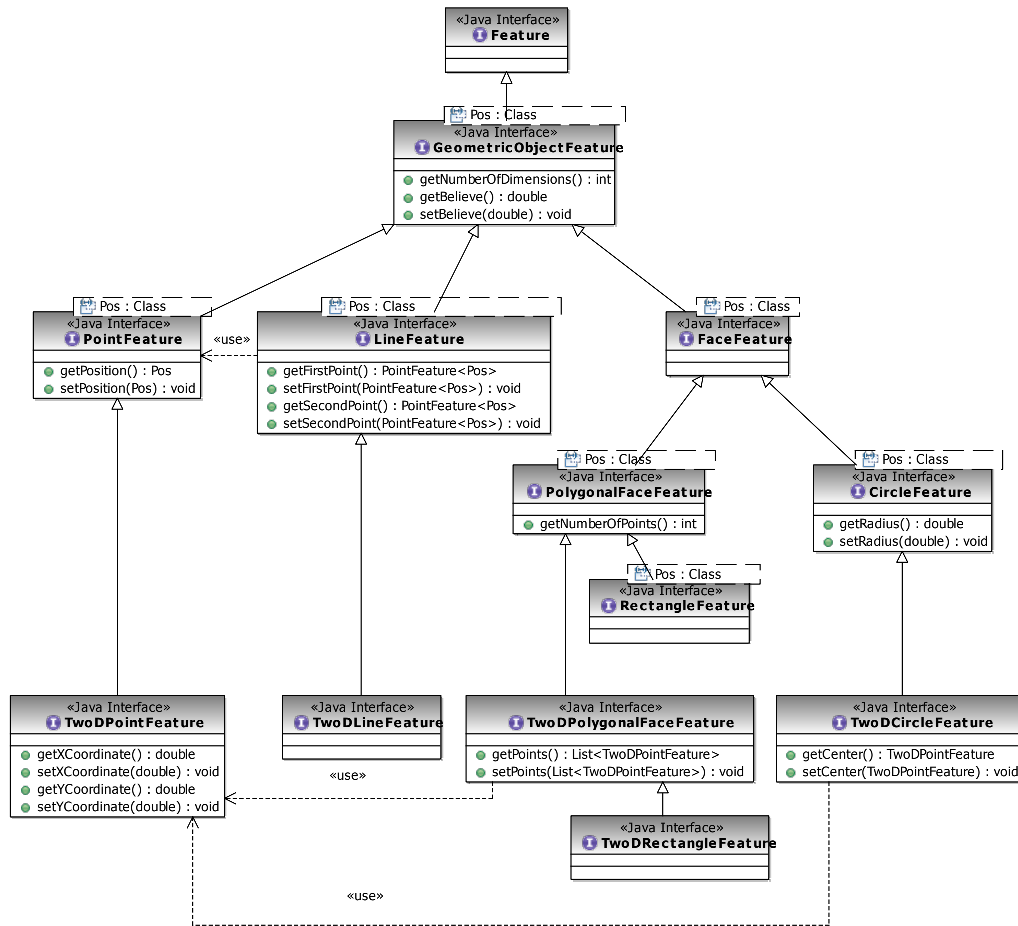


Figure 11: STOR geometric objects (features).

3.2.2 Features

In STOR any piece of information that can be extracted from an image and that is not part of an integrated model (yet) is called a *feature*. There are two different kinds of features explained in the following: *geometric objects* and *native objects*.

Geometric objects. A *geometric object* is a feature that represents some piece of geometric information extracted from an image. Geometric objects in general can be represented in different dimensions. Geometric objects that are extracted from a single image are two-dimensional.

In the domino tile recognition case study, there are three different kinds of geometric objects in STOR (Figure 11): points, lines and faces. Faces are specialized into polygonal faces and circles, polygonal faces are specialized into rectangles. There are variants of the described geometric objects as part of an integrated model (Section 3.2.3). Therefore all independent geometric objects have the suffix `Feature`.

Native objects. A *native object* is a data structure that has as properties only *references to the memory* (in Java stored as `long`). Native objects are always specific for one native library, and they are the only data structures that are *not serializable*.

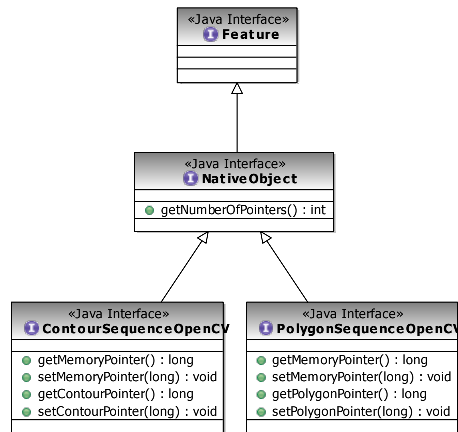


Figure 12: STOR native objects (features).

In native libraries, especially in function-oriented image processing libraries, there are often data that are represented as a *nested memory area* providing pointers to access it. The internal library functions are optimized to process these structures, but it is difficult to interpret them from the outside. Therefore those data are *wrapped* in native library specific objects on the Java layer. In this way they can be exchanged between Java components, but only those using the according native library functions can process them.

In the case study there are two different kinds of native objects (Figure 12). `ContourSequenceOpenCV` wraps an *OpenCV specific sequence of contours*, `PolygonSequenceOpenCV` wraps an *OpenCV specific sequence of polygons*. Both native objects have two pointers to the *OpenCV data type* `CvSeq` [2].

3.2.3 Integrated models

Integrated models are TGraphs (Section 2.4), whose structure, types and attributes help to model the different aspects (topology, geometry, semantics, and appearance) of the features recognized in a common integrated data structure. Their structure is defined by GrUML diagrams, which are exported to an `.xmi`-file from which the corresponding TGraph schema and the corresponding Java classes are generated automatically.

The schema of an integrated model and especially of its semantics part strongly depends on the application domain. Figure 13 shows a very simple integrated model used for the *domino tile recognition case study* with a *2d geometry*. A more elaborate integrated model for the *urban object model creation case study* based on a *3d geometry* can be found in [10].

In the integrated model for domino tile recognition there are currently *seven geometry/topology nodes* (left part of the figure) and *six semantics nodes* (right part of the figure). There are two different geometric/topologic objects: 2d points and 2d faces. A 2d face can be a 2d polygonal face or a 2d circle, a specialization of a 2d polygonal face is a 2d quadrangle and a specialization of a 2d quadrangle is a 2d rectangle. A 2d polygonal face contains four to any number 2d points, and a 2d circle has a 2d point as center.

There is one important *semantic object*, consisting of different parts: a *domino tile*. A domino tile consists of two domino tile parts and every domino tile part contains zero to six pips. A domino tile part is represented by a 2d quadrangle and has a layout. A pips[20] is represented by a 2d circle and has a layout position on its corresponding domino tile part. There are seven possible layouts and six possible layout positions.

[20] There is no singular for the word pips.

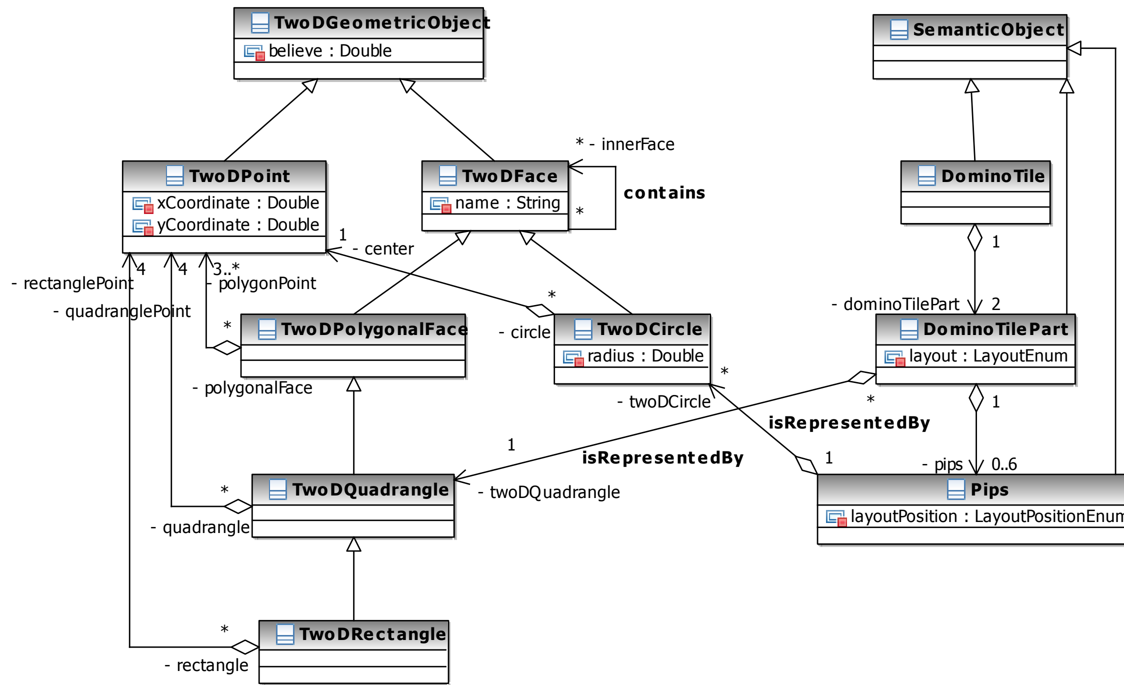


Figure 13: Integrated model for domino tiles.

3.3 STOR Components

STOR will supply a versatile scientific component system that supports the experimental construction of object recognition solutions. In this section the current form of *STOR components* is introduced. In the long run the component concept is still going to be refined. Currently, experiments with two existing component approaches are conducted: *JavaBeans components* and *ConQAT components*. Both variants have been chosen, since concrete assembly and execution environments are existing, though they do not fulfill all requirements of the final STOR assembly environment.

Section 3.3.1 describes *STOR components in general*, Section 3.3.2 specializes on the *STOR JavaBeans component variant* and Section 3.3.3 introduces the *STOR ConQAT component variant*.

3.3.1 STOR components in general

Like STOR data structures, *STOR components* consist of a *specification* in terms of a Java interface and an *implementation* in the form of a non-abstract Java class. A component interface extends the interface `STORComponent`. An interface usually is implemented more than once by different variants with the same functionality but different non-functional properties. A component class implements the `STORComponent` interface. It must have exactly one public parameterless constructor. A component class can use a *native library wrapper* (Section 3.3.5) to implement its functionality, but it should not use other components internally.

Component input data. STOR components strictly differentiate between two kinds of input-data: *parameters* and *arguments*. A *parameter* is a datum that configures a component and therefore affects its behavior. An *argument* is a datum that is processed by a component as its actual input. Parameters are stored as private global attributes of a component. They are handled by public get- and set-methods

defined in the component interface and usually have a *default value*. Arguments are passed to a component via an execute-*method* also defined in the component interface. Parameters of a component can already be set *at assembly time*, arguments are passed to a component *at runtime*. Preconditions for parameters and arguments besides their data type can be tested at runtime using assertions (Section 2.1).

Component output data. A STOR component can change its arguments and/or create new output data. If a component only changes its arguments, its execute-method has no return value. If it creates new output data, one output datum is the return value of the component's execute-method and more results can be accessed via further get-methods.

Component hierarchy. The set of STOR components is organized and grouped by the identified set of *activities for model-based object recognition in images* (Section 3.1). Components with comparable functionality have the same signature for their execute-methods and similar parameters. Therefore they implement the same or similar interfaces and are stored in the same or a similar location in the package hierarchy.

Currently, there are three main kinds of components: *image processing components*, *feature processing components* and *model processing components*. The feature diagram in Figure 6 shows the whole component hierarchy. All *component interfaces* are collected in the package de.uni_koblenz.stor.components and its subpackages. The corresponding *component classes* are kept in appropriate sub-packages with suffix .impl.

Example: median operator component. As an example the *STOR median operator component* and its interface/class hierarchy are explained in more detail (Figure 14).

The *median operator* [17, 22] in general is an *image preprocessing operator* for *noise reduction with edge preservation*. It is a *rank-order operator* and therefore it is *non-linear* and works *channel-wise* in a *local neighborhood*. It is a typical preprocessing step for edge detection and can be used for the *removal of extreme values* like salt and pepper noise or defective pixels. The median operator replaces every value of a channel by the median of a defined neighborhood. Usually the used neighborhood is squared (typically 3×3 or 5×5), but in general it can have any shape. If the neighborhood has an even number of values, the median is the arithmetic average of the two median values.

Every *component* has a method setDefaultValue, that sets all parameters of a component to their default values. Many image preprocessing components should be applicable to an image more than once. Therefore every *image preprocessing component* has a parameter numberOfPasses that decides how often the operator is applied to an image during one call of a components service, and a defaultNumberOfPasses with value 1. Moreover image preprocessing components should be applicable for all intensity images (Section 3.2.1). Therefore every image preprocessing component has an execute-method that gets an intensity image and returns an intensity image, which implies, that the component works out-of-place. Operators that are able to work in-place can define an additional execute-method without return value. A *median operator component* has a parameter maskSize, that decides the size of the squared neighborhood for median computation, and a defaultMaskSize with value 3.

The implemented median operator variant MedianOperatorImpl implements the described attributes and methods. Because the used OpenCV operator (cvSmooth() with parameter smoothtype=CV_MEDIAN) is only able to apply a median operator with an odd squared mask [2], the set-method of the maskSize-attribute accepts only odd values.

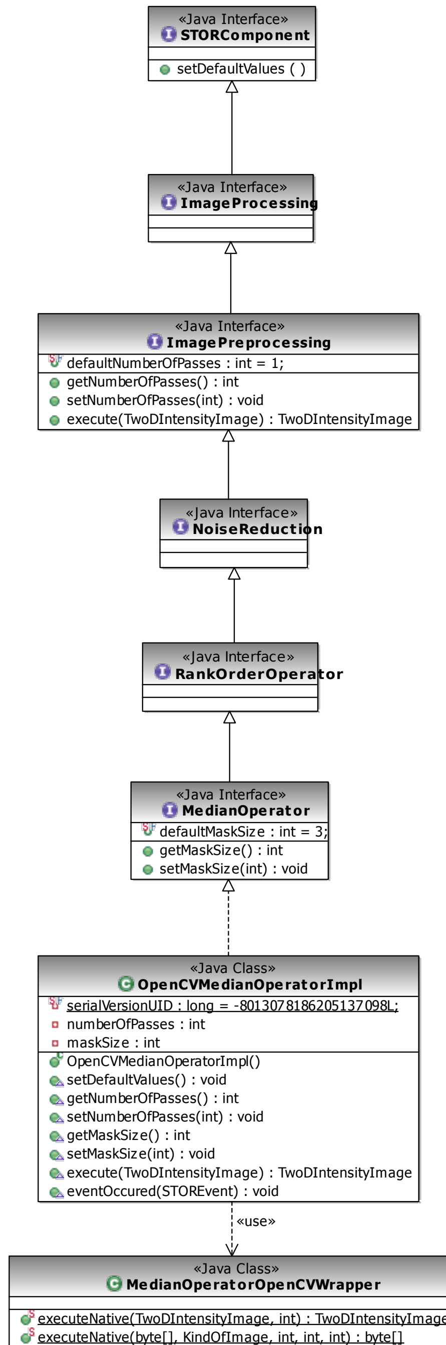


Figure 14: Median operator component hierarchy.

3.3.2 STOR JavaBeans components

The current STOR components (Section 3.3.1) already fulfill a lot of the *JavaBeans requirements*. They possess a parameterless public constructor and all attributes visible to the environment are private and have public get-methods and where applicable public set-methods. To make them fully comply with the JavaBeans specification they firstly are to be made serializable. Therefore we let the super

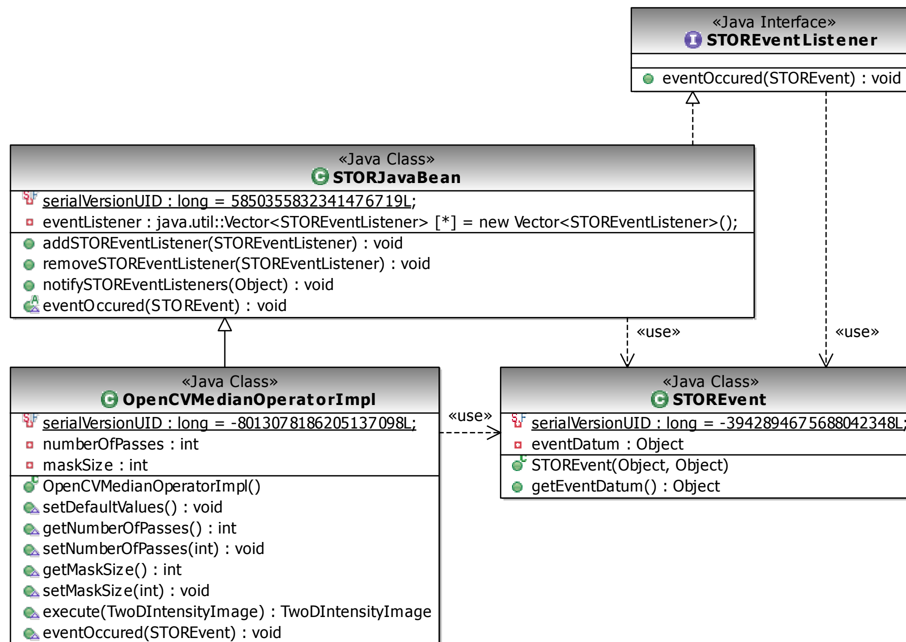


Figure 15: STOR JavaBeans Median operator component.

interface `STORComponent` extend the Java interface `Serializable` and add a generated serial version id to all implementing component classes.

Secondly as an experiment a simple *event concept* was developed so that STOR components can be assembled in a JavaBeans style. The idea is that every component has to be an *event listener* as well as an *event producer* so that every component can register itself by another one. This is achieved by the superclass `STORJavaBean` that has to be extended by every STOR component class. `STORJavaBean` implements the interface `STOREventListener` defining a method `eventOccured` with a parameter `STOREvent` that contains an attribute `eventDatum` of type `Object`. At the same time it offers methodes for the *(de-)registration of stor event listeners* and the *notfication of registered stor event listeners*, fullfilling the JavaBeans naming conentions. Every component has to implement the method `eventOccured` in the same way. It tests (via cast), if it is able to process the *events* `eventDatum` attribute. If this is the case and the component has all required input data, it processes the `eventDatum` with its `execute` method and offers the result to its registered components via firing a new `STOREvent` containing the result as `eventDatum`. As an example Figure 15 shows a class diagram containing a *STOR JavaBeans Median operator component* in form of its implementation `OpenCVMedianOperatorImpl`, the component superclass `STORJavaBean` as well as their relations to `STOREventListener` and `STOREvent`.

Figure 16 presents a screen shot of the whole *domino tile recognition case study* (Section 3.1.5) in *JBeanStudio*. The `JBeanStudio frame` above shows beside the menu a `palette STORPalette` containing a `folder STORFolder` that contains all existing STOR beans. Unfortunately the bean names are only visible during the mouse is moved about their icons. The `WireDesignFrame` below on the left shows the currently assembled bean instances. The `DefaultGUIFrame` below on the right shows the GUI that is currently defined by the assembled bean instances. The Frame `Properties` (accessible via the context menu of a focused bean) currently shows in its `Property tab` the changeable properties of the `OpenCVMedianOperatorImpl` bean but there are more tabs (namely `Customize`, `Input`, `Output` and `CallBack`).

As one can see there is *one additional bean* to the domino tile recognition case study, the `JBeanStudioStartButton` bean. This is because there seems to be no possibility to start the execu-

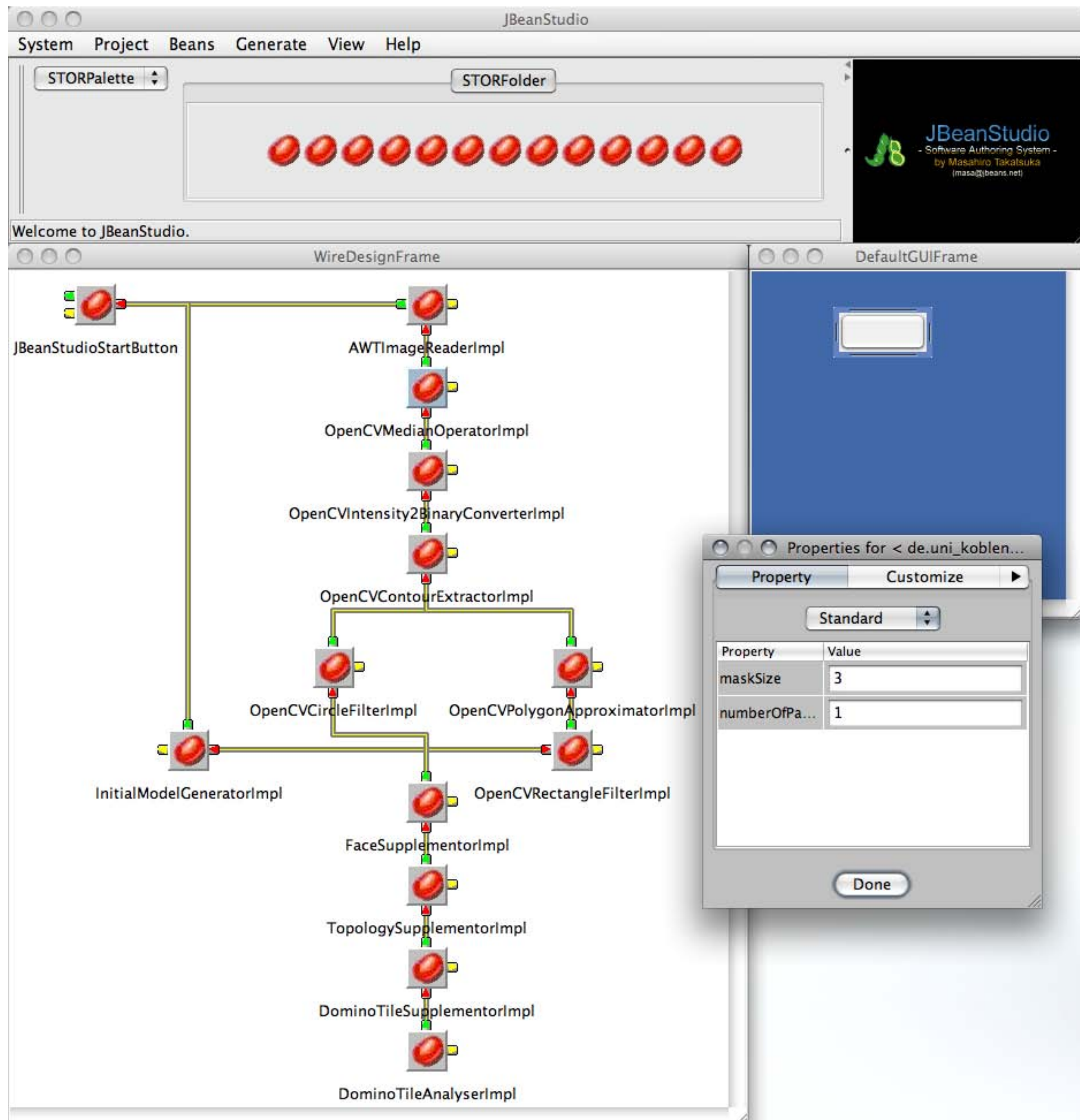


Figure 16: STOR JavaBeans domino tile recognition components assembled in JBeanStudio.

tion of a composit in JBeanStudio from the (conext) menu. Therefore this special bean was developed that has the same properties as a STOR bean but additionally extends `JButton` that has a graphical representation. The `JBeanStudioStartButton` fires a `STOREvent` to its registered beans if the mouse is clicked on it in the `DefaultGUIFrame`. In this way the execution of a composit can be started.

3.3.3 STOR ConQAT components

As an experiment a *STOR ConQAT bundle* was developed containing *ConQAT processors* for all required components for the domino tile recognition case study, where each processor *wraps* one of the

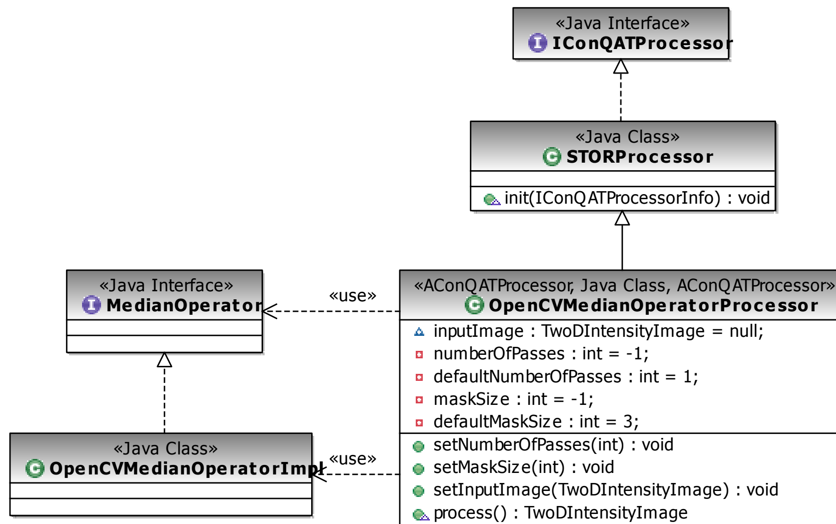


Figure 17: STOR ConQAT Median operator component.

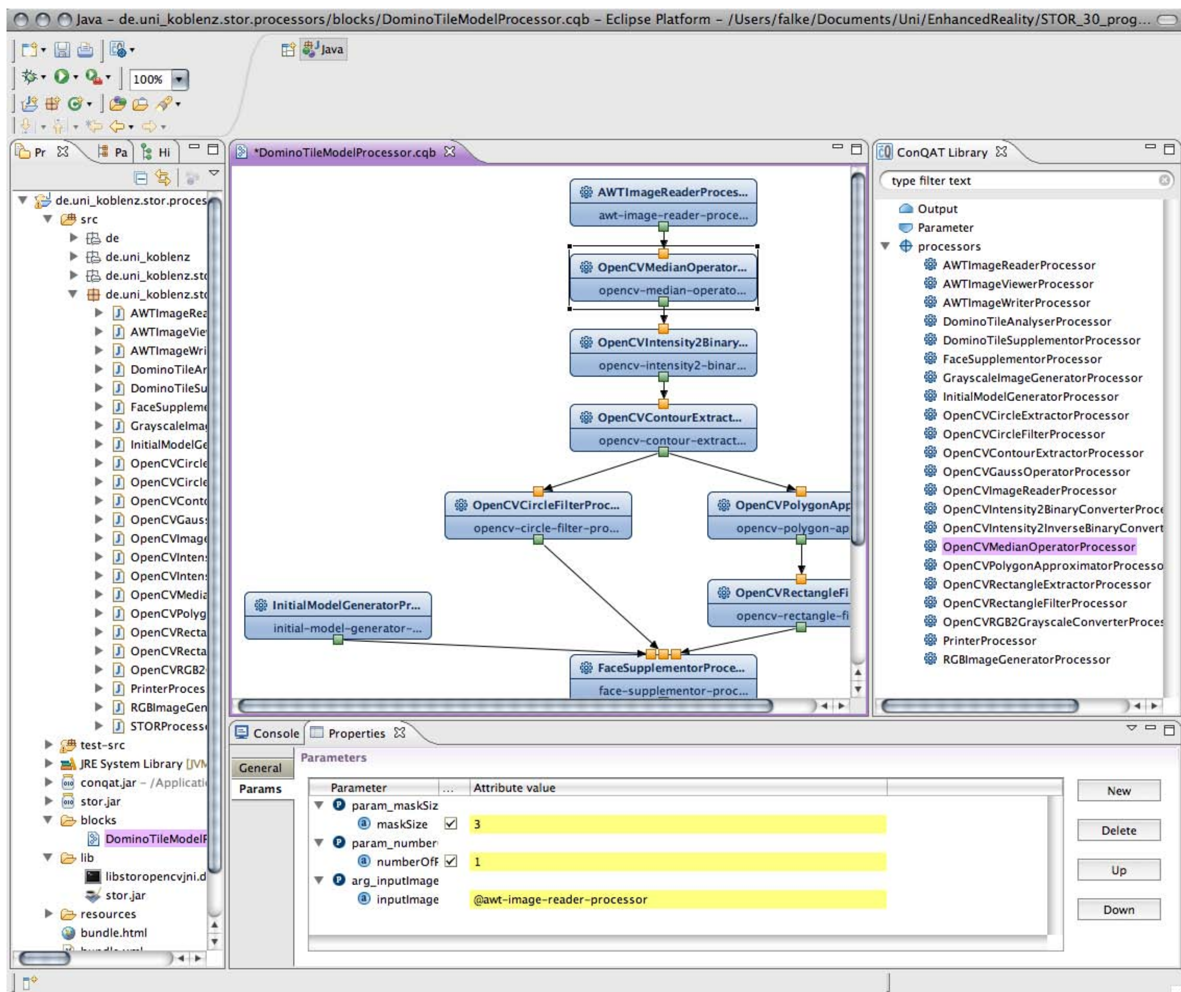


Figure 18: STOR ConQAT domino tile recognition block in CqEdit.

original STOR components. Technically the current STOR sources are packed in a Java archive so that they can be used by the processors. In this way ConQAT's development and assembly environment could be tested using the current STOR components without changing the project structure and/or the component syntax.

The mentioned bundle (named `de.uni_koblenz.stor.processors`) was created automatically using CqEdit, the jar archive was also generated automatically (Section A.2). But of course the processors (named like the used component with suffix `Processor`) had to be created manually. As an example Figure 17 shows a class diagram containing a *STOR ConQAT Median operator component* in form of its interface `MedianOperator`, its implementation `OpenCVMedianOperatorImpl` and its processor `MedianOperatorProcessor` as well as their relations to each other and to some ConQAT core classes.

Figure 18 presents a screen shot of the whole *domino tile recognition case study* (Section 3.1.5) in *CqEdit*. The `Project Explorer` on the left side shows the `src`-folder containing all existing processors, the `blocks`-folder containing among others the `DominoTileRecognition`-block and the `lib`-folder containing the Java archive `stor.jar` and the dynamically loadable OpenCV library `libstoropencvjni.dylib`. The `ConQATLibrary view` on the right side displays all available processors and blocks. In the `ConQAT Block Editor` above in the middle one can see the opened `DominoTileRecognition`-block containing the assembled processors. In the `menu item Params` of the `Properties view` below the current parameters of the `MedianOperatorProcessor` instance are shown.[21]

3.3.4 Domino tile recognition intermediary result images

This section presents some intermediary result images from the domino tile recognition case study, described in Section 3.1.5 and implemented with JavaBeans (Section 3.3.2) as well as with ConQAT (Section 3.3.3).

Figure 19 shows an original RGB image containing a domino tile, the RGB image smoothed via median operator, the smootehd RGB image converted to a grayscale image[22], the grayscale image converted to a binary image and the rectangle as well as the circles that are extracted from the given image.

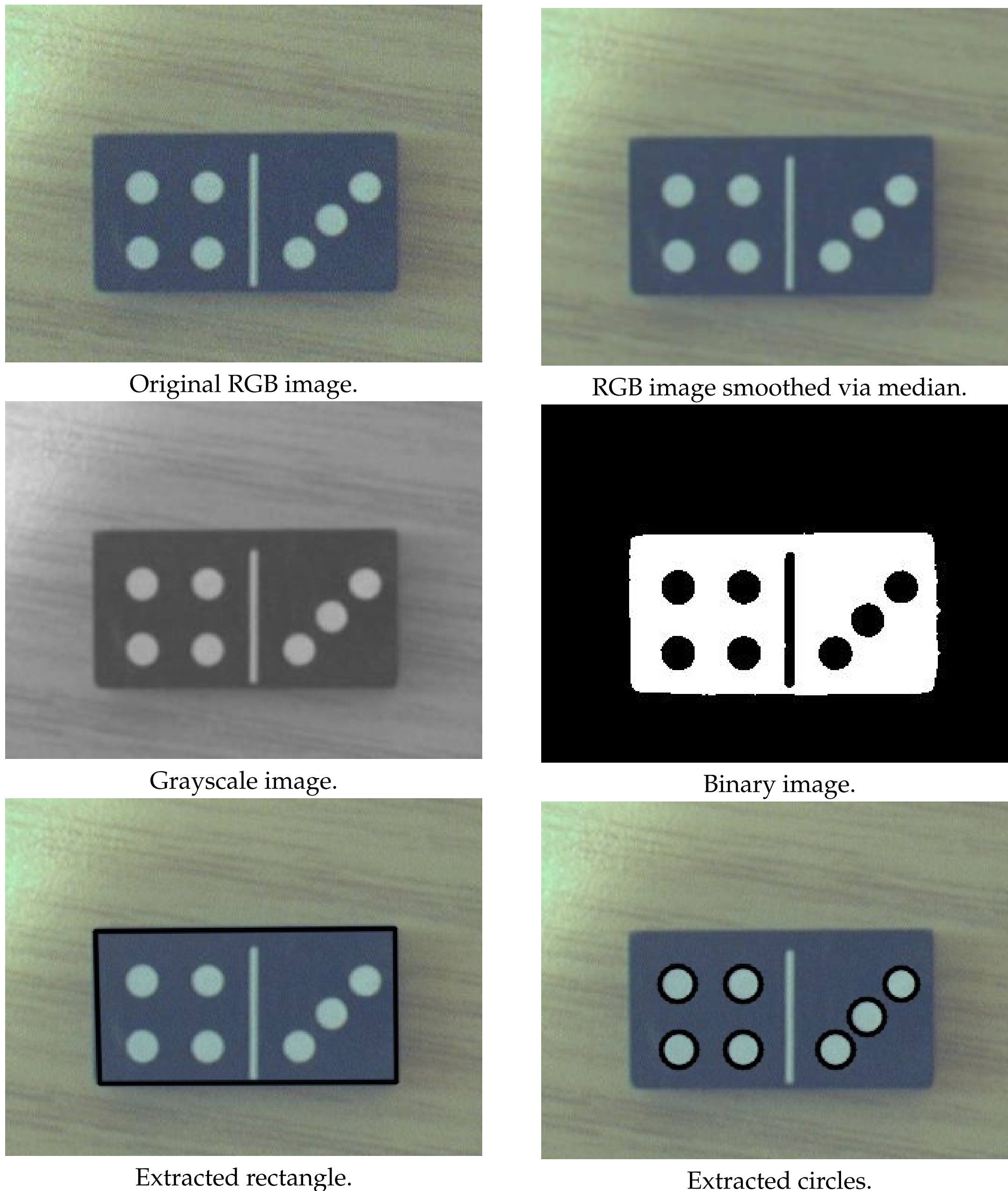
3.3.5 Native library wrappers

This section introduces the so-called *native library wrappers* that wrap parts of a native library's functionality to be usable from Java-based STOR components via JNI.

In the context of STOR a native library wrapper is a Java class containing one or more native methods for a specific native library. A native method is a Java method with the *modifier* `native` and *without any body*.

There are native library wrappers for *OpenCV*, *PUMA* and *KIPL* (Section 2.3). The wrappers for each native library are packed in their *own subpackage* containing a common *superclass* the wrappers have to extend (Figure 20). During project build (Section A.2) the task `generateC++headers` generates a C++ header for every existing native library wrapper. Such a generated C++ header contains the same method signatures as the native library wrapper class, but in C++ syntax instead of Java syntax. Now, a corresponding C++ source file can be programmed. Also during project build the task `buildexternallibraries` generates one dynamically lodadable library for every native library, containing the corresponding header and source files. During runtime (more precisely during the first initialization of a native library wrapper instance for a specific native library) the corresponding dynamically loadable library is loaded and the native methods can be called.

[21] The block name for identification can only be seen in the `menu item General` of the `Properties view`.
[22] This activity was not mentioned before.



Original RGB image. RGB image smoothed via median.

Grayscale image. Binary image.

Extracted rectangle. Extracted circles.

Figure 19: Intermediary result images during domino tile recognition.

Figure 21 shows an example for a `MedianOperatorOpenCVWrapper`. It is used by the STOR component class `OpenCVMedianOperatorImpl` and inherits the class `OpenCVWrapper`. The *wrapper class* contains two different native methods. The first method has the STOR image type `IntensityImage` (Section 3.2.1) in its signature, the second one reduces its signature to atomic data types and a byte array. Using the first method, the corresponding C++ program has to access the Java object of type `IntensityImage` via JNI. Using the second method, the corresponding C++ program can directly work on the given data. The first variant is more adequate, but the other variant is needed for testing, as well. For this native library wrapper a header file `MedianOperatorOpenCV.h` was generated and a corresponding source file `MedianOperatorOpenCV.cpp` was programmed that calls the corresponding C++-methods.

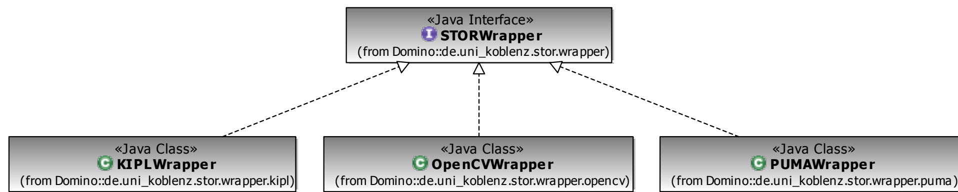


Figure 20: Native library wrapper.

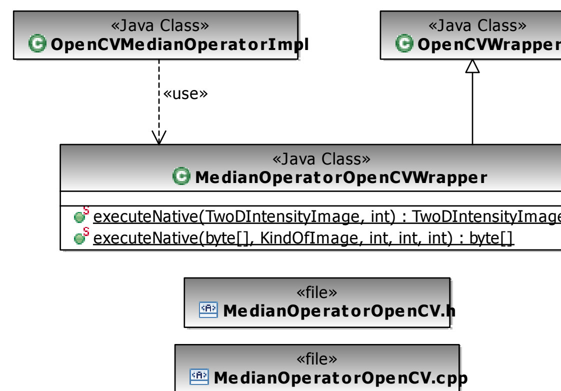


Figure 21: STOR median operator OpenCV wrapper.

4 Conclusion and future work

In this technical report the *software engineering part* of the *STOR project* was introduced. After some state of the art the *main activities* for model-based object recognition in images were described, and based thereon the *used data structures* and the *derived components*, that are implemented in two different two-language variants, were explained. These entities and the experiences and requirements collected during development are forming a base for the *STOR scientific component system*.

Further research tasks are the construction of a *general STOR scientific component system* and its application to the (model-based) object recognition in images domain. This includes the development of a coherent *component concept*, an own *assembly environment*, a *component library*, containing components for the most important activities for all facets of object recognition in images (image processing, feature processing and model processing) and the implementation in a *prototype*. Component concept and assembly environment development includes the definite specification of a components interface and a general type system containing the described states as well as an assembly concept offering support for all component life cycle activities, especially the assembly of components without manual programming and the use of composed components as components again. Component library development includes the refinement of the identified activities, data structures and components (including used wrappers) and the identification and development of new ones to complete the described data structure and component hierarchies. Moreover it contains the development of composed components for reference solutions of specific object recognition activities. Of course another research task is the evaluation of the whole system.

Acknowledgements. The work reported here was done in tight cooperation with the Work Group Active Vision (Dietrich Paulus, Stefan Wirtz, Peter Decker) and the Image Recognition Laboratory (Lutz Priese, Frank Schmitt) and was supported by our students Judith Haas [15] and Sabine Orth [20].

A STOR development environment

Since STOR aims at experimental work, a first STOR development environment hast been created to support the experiments. Section A.1 describes the STOR folder structure, and Section A.2 introduces the STOR build environment.

A.1 Folder structure

All data for a concrete STOR case study are kept together in a project folder. Such a folder has the following contents.

- `build`: generated files (described below)
- `lib`: external and generated internal libraries (described below).
- `models`: integrated model files.
- `src`: Java and C++ sources.
- `testdata`: any test data.
- `testsrc`: any test sources.
- `build.xml`: ANT file

The `build` folder consists of some more specific folders containing the generated artifacts, e.g.

- `build/classes`: generated Java .class files.
- `build/doc`: generated Javadoc documentation files.
- `build/jar`: the generated .jar archive.
- `build/objects`: generated C/C++ .o files.
- `build/temp`: temporarily generated files.

The `lib` folder contains the libraries needed for the project, e.g.

- `cpptasks.jar`: ANT-Contrib jar-archive.
- `jama.jar`: JAMA jar-archive.
- `jgralab.jar`: JGraLab jar-archive.
- `libstorkipljni.dylib`: KIPL dynamically loadable library.
- `libstoropencvjni.dylib`: OpenCV dynamically loadable library.
- `libstorpumajni.dylib`: PUMA dynamically loadable library.

The `src` and the `build/classes` folder structures contain the package structure.

Following common practice in Java development, all STOR source files are packed in a package hierarchy beginning with the package `de.uni_koblenz.stor`[23]. The package contains the following sub-packages:

- `beans`: JavaBeans specific source code (Section 3.3.2).
- `components`: STOR components (Section 3.3).
- `datastructures`: STOR data structures (Section 3.2).
- `wrapper`: STOR native library wrappers (Section 3.3.5).

A.2 Build environment

There is a STOR specific build environment based on the Java-based build tool *Apache ANT*[24] and the additional project *ANT-Contrib*[25], providing a collection of specific ANT tasks for C++ development. The STOR ANT file `build.xml` contains a lot of *tasks* responsible for all development steps that can be carried out automatically. The tasks can be executed individually or combined (Figure 22).

[23] Since a hyphen is not allowed in Java package names, an underscore is used between `uni` and `koblenz`.
[24] `http://ant.apache.org`
[25] `http://ant-contrib.sourceforge.net`

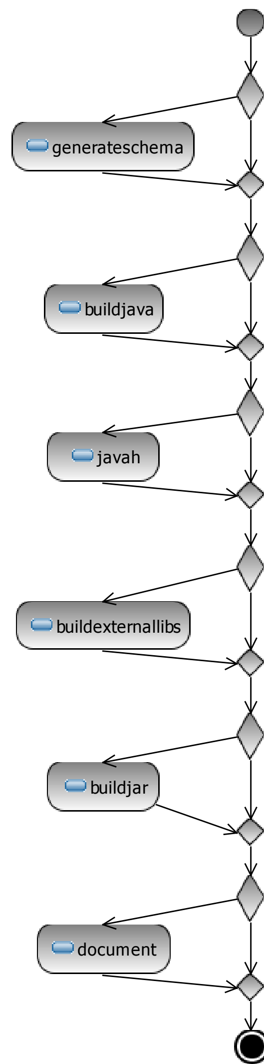


Figure 22: STOR build tasks.

The task `converteschema` converts the integrated model schema from an .xmi-file to a .tg file, `generatejavaschema` generates the according Java classes and both tasks are combined in the task `generateschema` (Section 3.2.3). `buildjava` compiles all Java files. `javah` generates header files for native methods and `buildexternallibs` compiles C++ source files and generates dynamically loadable libraries for the used native libraries (Section 3.3.5). It can be divided into the tasks `buildkipl`, `buildpuma` and `buildopencv`. For all of these tasks there are three platform specific tasks (Mac OS X, Linux and Windows) and ANT automatically calls the right tasks per platform. The task `unjar` unpacks the required library jar archives, `jar` packs all required files in a new STOR jar archive and both tasks are combined in the task `buildjar`. Last but not least `document` documents all Java sources using Javadoc. The task `buildall` executes all other tasks described before.

References

[1] D. Balthasar. *Drei neue Verfahren zum Matching und zur Klassifikation unter Echtzeitbedingungen*. PhD thesis, Universität Koblenz, Verlag Fölbach Koblenz, 2006.

[2] G. Bradski and A. Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 10 2008.

[3] L. P. G. Cable. Extensible Runtime Containment and Server Protocol for JavaBeans. Technical report, Sun Microsystems, 12 1998. Version 1.0.

[4] L. P. G. Cable. Proposal for a Drag and Drop subsystem for the Java Foundation Classes. Technical report, Sun Microsystems, 08 1998. Final draft: 0.96.

[5] B. Calder and B. Shannon. JavaBeans Activation Framework Specification. Technical report, Sun Microsystems, 05 1999. Version 1.0a.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[7] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. Mas y Parareda, and M. Pizka. Tool Support for Continuous Quality Control. *IEEE Software*, 25(5):60–67, 2008.

[8] F. Deissenboeck, M. Pizka, and T. Seifert. Tool Support for Continuous Quality Assessment. In *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*, pages 127–136, Los Alamitos, CA, USA, 9 2005. IEEE Computer Society.

[9] J. Ebert, V. Riediger, and A. Winter. Graph Technology in Reverse Engineering, The TGraph Approach. In R. Gimnich, U. Kaiser, J. Quante, and A. Winter, editors, *10th Workshop Software Reengineering (WSR 2008)*, volume 126 of *GI Lecture Notes in Informatics*, pages 67–81, Bonn, 2008. GI.

[10] K. Falkowski and J. Ebert. Graph-based urban object model processing. In *City Models, Roads and Traffic (CMRT'09): Object Extraction for 3D City Models, Road Databases and Traffic Monitoring - Concepts, Algorithms and Evaluation*, Paris, France, 09 2009. International Society for Photogrammetry and Remote Sensing (ISPRS). Accepted for publication.

[11] K. Falkowski, J. Ebert, P. Decker, S. Wirtz, and D. Paulus. Semi-automatic generation of full CityGML models from images. In *Geoinformatik 2009*, volume 35 of *ifgiPrints*, pages 101–110, Osnabrück, Germany, 4 2009. Institut für Geoinformatik Westfälische Wilhelms-Universität Münster.

[12] I. Forman and N. Forman. *Java Reflection in Action*. In Action series. Manning Publications, 11 2004.

[13] K. E. Gorlen, S. M. Orlow, and P. S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. Wiley, 1990.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. The Java series. Addison-Wesley Longman, 3 edition, 6 2005.

[15] J. Haas. Analyse, Evaluation und Vergleich von Bildverarbeitungsbibliotheken aus Sicht der Softwaretechnik. Master's thesis, Universität Koblenz-Landau, Institut für Softwaretechnik, Arbeitsgruppe Softwaretechnik, 4 2009.

[16] G. Hamilton (Editor). JavaBeans. Technical report, Sun Microsystems, 08 1997. Version 1.01-A.

[17] T. Huang, editor. *Two-Dimensional Digital Signal Prcessing II*, volume 43/1981 of *Topics in Applied Physics*. Springer Berlin / Heidelberg, 1981.

[18] M. Johnson. The beanbox: Sun's javabeans test container. *Java World*, 1997.

[19] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. The Java series. Prentice Hall, 1 edition, 6 1999.

[20] S. Orth. Entwicklung eines Konzepts zur Selbstauskunftsfähigkeit für STOR-Komponenten. Master's thesis, Universität Koblenz-Landau, Institut für Softwaretechnik, Arbeitsgruppe Softwaretechnik, 2009. In Process.

[21] D. Paulus. *Objektorientierte und wissensbasierte Bildverarbeitung*. PhD thesis, Universität Erlangen-Nürnberg, 1992.

[22] D. Paulus and J. Hornegger. *Applied Pattern Recognition*. Vieweg, 2003.

[23] M. Rivera-Malpica. Analyse und vergleich computervisualistischer softwarebausteine aus sicht der softwaretechnik. Master's thesis, Universität Koblenz-Landau, Institut für Softwaretechnik, Arbeitsgruppe, 7 2006.

[24] M. Takatsuka. Jbeanstudio: A component-oriented visual software authoring system for a problem solving environment - supporting exploratory visualization. In P. M. A. Sloot, D. Abramson, A. V. Bogdanov, J. Dongarra, A. Y. Zomaya, and Y. E. Gorbachev, editors, *International Conference on Computational Science*, volume 2659 of *Lecture Notes in Computer Science*, pages 985–994. Springer, 2003.

[25] M. Takatsuka. A component-oriented software authoring system for exploratory visualization. *Future Gener. Comput. Syst.*, 21(7):1213–1222, 2005.

[26] G. Voss. Java Beans, Pt 3: Testing Beans in the BDK BeanBox. Tutorial, Sun Microsystems, 02 1997.

Bisher erschienen

Arbeitsberichte aus dem Fachbereich Informatik
(http://www.uni-koblenz.de/fb4/publikationen/arbeitsberichte)

Kerstin Falkowski, Jürgen Ebert, The STOR Component System, Interim Report, Arbeitsberichte aus dem Fachbereich Informatik 14/2009

Sebastian Magnus, Markus Maron, An Empirical Study to Evaluate the Location of Advertisement Panels by Using a Mobile Marketing Tool, Arbeitsberichte aus dem Fachbereich Informatik 13/2009

Sebastian Magnus, Markus Maron, Konzept einer Public Key Infrastruktur in iCity, Arbeitsberichte aus dem Fachbereich Informatik 12/2009

Sebastian Magnus, Markus Maron, A Public Key Infrastructure in Ambient Information and Transaction Systems, Arbeitsberichte aus dem Fachbereich Informatik 11/2009

Ammar Mohammed, Ulrich Furbach, Multi-agent systems: Modeling and Virification using Hybrid Automata, Arbeitsberichte aus dem Fachbereich Informatik 10/2009

Andreas Sprotte, Performance Measurement auf der Basis von Kennzahlen aus betrieblichen Anwendungssystemen: Entwurf eines kennzahlengestützten Informationssystems für einen Logistikdienstleister, Arbeitsberichte aus dem Fachbereich Informatik 9/2009

Gwendolin Garbe, Tobias Hausen, Process Commodities: Entwicklung eines Reifegradmodells als Basis für Outsourcingentscheidungen, Arbeitsberichte aus dem Fachbereich Informatik 8/2009

Petra Schubert et. al., Open-Source-Software für das Enterprise Resource Planning, Arbeitsberichte aus dem Fachbereich Informatik 7/2009

Ammar Mohammed, Frieder Stolzenburg, Using Constraint Logic Programming for Modeling and Verifying Hierarchical Hybrid Automata, Arbeitsberichte aus dem Fachbereich Informatik 6/2009

Tobias Kippert, Anastasia Meletiadou, Rüdiger Grimm, Entwurf eines Common Criteria-Schutzprofils für Router zur Abwehr von Online-Überwachung, Arbeitsberichte aus dem Fachbereich Informatik 5/2009

Hannes Schwarz, Jürgen Ebert, Andreas Winter, Graph-based Traceability – A Comprehensive Approach. Arbeitsberichte aus dem Fachbereich Informatik 4/2009

Anastasia Meletiadou, Simone Müller, Rüdiger Grimm, Anforderungsanalyse für Risk-Management-Informationssysteme (RMIS), Arbeitsberichte aus dem Fachbereich Informatik 3/2009

Ansgar Scherp, Thomas Franz, Carsten Saathoff, Steffen Staab, A Model of Events based on a Foundational Ontology, Arbeitsberichte aus dem Fachbereich Informatik 2/2009

Frank Bohdanovicz, Harald Dickel, Christoph Steigner, Avoidance of Routing Loops, Arbeitsberichte aus dem Fachbereich Informatik 1/2009

Stefan Ameling, Stephan Wirth, Dietrich Paulus, Methods for Polyp Detection in Colonoscopy Videos: A Review, Arbeitsberichte aus dem Fachbereich Informatik 14/2008

Tassilo Horn, Jürgen Ebert, Ein Referenzschema für die Sprachen der IEC 61131-3, Arbeitsberichte aus dem Fachbereich Informatik 13/2008

Thomas Franz, Ansgar Scherp, Steffen Staab, Does a Semantic Web Facilitate Your Daily Tasks?, Arbeitsberichte aus dem Fachbereich Informatik 12/2008

Norbert Frick, Künftige Anfordeungen an ERP-Systeme: Deutsche Anbieter im Fokus, Arbeitsberichte aus dem Fachbereicht Informatik 11/2008

Jürgen Ebert, Rüdiger Grimm, Alexander Hug, Lehramtsbezogene Bachelor- und Masterstudiengänge im Fach Informatik an der Universität Koblenz-Landau, Campus Koblenz, Arbeitsberichte aus dem Fachbereich Informatik 10/2008

Mario Schaarschmidt, Harald von Kortzfleisch, Social Networking Platforms as Creativity Fostering Systems: Research Model and Exploratory Study, Arbeitsberichte aus dem Fachbereich Informatik 9/2008

Bernhard Schueler, Sergej Sizov, Steffen Staab, Querying for Meta Knowledge, Arbeitsberichte aus dem Fachbereich Informatik 8/2008

Stefan Stein, Entwicklung einer Architektur für komplexe kontextbezogene Dienste im mobilen Umfeld, Arbeitsberichte aus dem Fachbereich Informatik 7/2008

Matthias Bohnen, Lina Brühl, Sebastian Bzdak, RoboCup 2008 Mixed Reality League Team Description, Arbeitsberichte aus dem Fachbereich Informatik 6/2008

Bernhard Beckert, Reiner Hähnle, Tests and Proofs: Papers Presented at the Second International Conference, TAP 2008, Prato, Italy, April 2008, Arbeitsberichte aus dem Fachbereich Informatik 5/2008

Klaas Dellschaft, Steffen Staab, Unterstützung und Dokumentation kollaborativer Entwurfs- und Entscheidungsprozesse, Arbeitsberichte aus dem Fachbereich Informatik 4/2008

Rüdiger Grimm: IT-Sicherheitsmodelle, Arbeitsberichte aus dem Fachbereich Informatik 3/2008

Rüdiger Grimm, Helge Hundacker, Anastasia Meletiadou: Anwendungsbeispiele für Kryptographie, Arbeitsberichte aus dem Fachbereich Informatik 2/2008

Markus Maron, Kevin Read, Michael Schulze: CAMPUS NEWS – Artificial Intelligence Methods Combined for an Intelligent Information Network, Arbeitsberichte aus dem Fachbereich Informatik 1/2008

Lutz Priese,Frank Schmitt, Patrick Sturm, Haojun Wang: BMBF-Verbundprojekt 3D-RETISEG Abschlussbericht des Labors Bilderkennen der Universität Koblenz-Landau, Arbeitsberichte aus dem Fachbereich Informatik 26/2007

Stephan Philippi, Alexander Pinl: Proceedings 14. Workshop 20.-21. September 2007 Algorithmen und Werkzeuge für Petrinetze, Arbeitsberichte aus dem Fachbereich Informatik 25/2007

Ulrich Furbach, Markus Maron, Kevin Read: CAMPUS NEWS – an Intelligent Bluetooth-based Mobile Information Network, Arbeitsberichte aus dem Fachbereich Informatik 24/2007

Ulrich Furbach, Markus Maron, Kevin Read: CAMPUS NEWS - an Information Network for Pervasive Universities, Arbeitsberichte aus dem Fachbereich Informatik 23/2007

Lutz Priese: Finite Automata on Unranked and Unordered DAGs Extented Version, Arbeitsberichte aus dem Fachbereich Informatik 22/2007

Mario Schaarschmidt, Harald F.O. von Kortzfleisch: Modularität als alternative Technologie- und Innovationsstrategie, Arbeitsberichte aus dem Fachbereich Informatik 21/2007

Kurt Lautenbach, Alexander Pinl: Probability Propagation Nets, Arbeitsberichte aus dem Fachbereich Informatik 20/2007

Rüdiger Grimm, Farid Mehr, Anastasia Meletiadou, Daniel Pähler, Ilka Uerz: SOA-Security, Arbeitsberichte aus dem Fachbereich Informatik 19/2007

Christoph Wernhard: Tableaux Between Proving, Projection and Compilation, Arbeitsberichte aus dem Fachbereich Informatik 18/2007

Ulrich Furbach, Claudia Obermaier: Knowledge Compilation for Description Logics, Arbeitsberichte aus dem Fachbereich Informatik 17/2007

Fernando Silva Parreiras, Steffen Staab, Andreas Winter: TwoUse: Integrating UML Models and OWL Ontologies, Arbeitsberichte aus dem Fachbereich Informatik 16/2007

Rüdiger Grimm, Anastasia Meletiadou: Rollenbasierte Zugriffskontrolle (RBAC) im Gesundheitswesen, Arbeitsberichte aud dem Fachbereich Informatik 15/2007

Ulrich Furbach, Jan Murray, Falk Schmidsberger, Frieder Stolzenburg: Hybrid Multiagent Systems with Timed Synchronization-Specification and Model Checking, Arbeitsberichte aus dem Fachbereich Informatik 14/2007

Björn Pelzer, Christoph Wernhard: System Description:"E-KRHyper", Arbeitsberichte aus dem Fachbereich Informatik, 13/2007

Ulrich Furbach, Peter Baumgartner, Björn Pelzer: Hyper Tableaux with Equality, Arbeitsberichte aus dem Fachbereich Informatik, 12/2007

Ulrich Furbach, Markus Maron, Kevin Read: Location based Informationsystems, Arbeitsberichte aus dem Fachbereich Informatik, 11/2007

Philipp Schaer, Marco Thum: State-of-the-Art: Interaktion in erweiterten Realitäten, Arbeitsberichte aus dem Fachbereich Informatik, 10/2007

Ulrich Furbach, Claudia Obermaier: Applications of Automated Reasoning, Arbeitsberichte aus dem Fachbereich Informatik, 9/2007

Jürgen Ebert, Kerstin Falkowski: A First Proposal for an Overall Structure of an Enhanced Reality Framework, Arbeitsberichte aus dem Fachbereich Informatik, 8/2007

Lutz Priese, Frank Schmitt, Paul Lemke: Automatische See-Through Kalibrierung, Arbeitsberichte aus dem Fachbereich Informatik, 7/2007

Rüdiger Grimm, Robert Krimmer, Nils Meißner, Kai Reinhard, Melanie Volkamer, Marcel Weinand, Jörg Helbach: Security Requirements for Non-political Internet Voting, Arbeitsberichte aus dem Fachbereich Informatik, 6/2007

Daniel Bildhauer, Volker Riediger, Hannes Schwarz, Sascha Strauß, „grUML – Eine UML-basierte Modellierungssprache für T-Graphen", Arbeitsberichte aus dem Fachbereich Informatik, 5/2007

Richard Arndt, Steffen Staab, Raphaël Troncy, Lynda Hardman: Adding Formal Semantics to MPEG-7: Designing a Well Founded Multimedia Ontology for the Web, Arbeitsberichte aus dem Fachbereich Informatik, 4/2007

Simon Schenk, Steffen Staab: Networked RDF Graphs, Arbeitsberichte aus dem Fachbereich Informatik, 3/2007

Rüdiger Grimm, Helge Hundacker, Anastasia Meletiadou: Anwendungsbeispiele für Kryptographie, Arbeitsberichte aus dem Fachbereich Informatik, 2/2007

Anastasia Meletiadou, J. Felix Hampe: Begriffsbestimmung und erwartete Trends im IT-Risk-Management, Arbeitsberichte aus dem Fachbereich Informatik, 1/2007

„Gelbe Reihe"
(http://www.uni-koblenz.de/fb4/publikationen/gelbereihe)

Lutz Priese: Some Examples of Semi-rational and Non-semi-rational DAG Languages. Extended Version, Fachberichte Informatik 3-2006

Kurt Lautenbach, Stephan Philippi, and Alexander Pinl: Bayesian Networks and Petri Nets, Fachberichte Informatik 2-2006

Rainer Gimnich and Andreas Winter: Workshop Software-Reengineering und Services, Fachberichte Informatik 1-2006

Kurt Lautenbach and Alexander Pinl: Probability Propagation in Petri Nets, Fachberichte Informatik 16-2005

Rainer Gimnich, Uwe Kaiser, and Andreas Winter: 2. Workshop ''Reengineering Prozesse'' – Software Migration, Fachberichte Informatik 15-2005

Jan Murray, Frieder Stolzenburg, and Toshiaki Arai: Hybrid State Machines with Timed Synchronization for Multi-Robot System Specification, Fachberichte Informatik 14-2005

Reinhold Letz: FTP 2005 – Fifth International Workshop on First-Order Theorem Proving, Fachberichte Informatik 13-2005

Bernhard Beckert: TABLEAUX 2005 – Position Papers and Tutorial Descriptions, Fachberichte Informatik 12-2005

Dietrich Paulus and Detlev Droege: Mixed-reality as a challenge to image understanding and artificial intelligence, Fachberichte Informatik 11-2005

Jürgen Sauer: 19. Workshop Planen, Scheduling und Konfigurieren / Entwerfen, Fachberichte Informatik 10-2005

Pascal Hitzler, Carsten Lutz, and Gerd Stumme: Foundational Aspects of Ontologies, Fachberichte Informatik 9-2005

Joachim Baumeister and Dietmar Seipel: Knowledge Engineering and Software Engineering, Fachberichte Informatik 8-2005

Benno Stein and Sven Meier zu Eißen: Proceedings of the Second International Workshop on Text-Based Information Retrieval, Fachberichte Informatik 7-2005

Andreas Winter and Jürgen Ebert: Metamodel-driven Service Interoperability, Fachberichte Informatik 6-2005

Joschka Boedecker, Norbert Michael Mayer, Masaki Ogino, Rodrigo da Silva Guerra, Masaaki Kikuchi, and Minoru Asada: Getting closer: How Simulation and Humanoid League can benefit from each other, Fachberichte Informatik 5-2005

Torsten Gipp and Jürgen Ebert: Web Engineering does profit from a Functional Approach, Fachberichte Informatik 4-2005

Oliver Obst, Anita Maas, and Joschka Boedecker: HTN Planning for Flexible Coordination Of Multiagent Team Behavior, Fachberichte Informatik 3-2005

Andreas von Hessling, Thomas Kleemann, and Alex Sinner: Semantic User Profiles and their Applications in a Mobile Environment, Fachberichte Informatik 2-2005

Heni Ben Amor and Achim Rettinger: Intelligent Exploration for Genetic Algorithms – Using Self-Organizing Maps in Evolutionary Computation, Fachberichte Informatik 1-2005