Software Documentation template

Release 0.0.

Raphael Dürscheid, based on Template by Dr. Peter

Jul 01, 2020

USAGE AND INSTALLATION

The fees payment and management system is a digital platform designed to simplify the process of paying and managing fees for educational institutions such as schools, colleges, and universities. The goal of the system is to streamline and automate the fees payment process, making it more convenient for students and their families, as well as for school administrators and staff.

The system allows students to view their fee balance, make online payments, and receive receipts and confirmation of payments. It also provides real-time updates on fee collections and enables administrators to generate reports on fee payments, overdue payments, and other relevant data.

In addition to simplifying the fees payment process, the system can also help institutions to manage their finances more effectively. By automating many of the tasks associated with fee collection and management, the system can reduce administrative overhead and enable administrators to focus on other important tasks.

Overall, the fees payment and management system is a powerful tool that can help educational institutions to improve their financial management and provide better service to their students and families.

ONE

RELEVANT BACKGROUND INFORMATION AND PRE-REQUISITS

Potential Users:

To use the fees payment and management system, a potential user needs to be familiar with basic computer skills such as using a web browser, creating and using online accounts, and making online payments. They should also be familiar with the fee structure and policies of their educational institution.

Before using the system, users should read and understand the terms and conditions, privacy policy, and any other relevant documentation provided by the institution or the system provider. They should also ensure that their payment information is accurate and up-to-date.

Developers:

To further understand the code of the fees payment and management system, a developer should be familiar with web development technologies such as HTML, CSS, JavaScript, and PHP. They should also have experience working with database management systems such as MySQL.

Developers should read and understand the system's architecture, design patterns, and coding conventions. They should also be familiar with the system's security protocols and best practices for secure coding.

Relevant documents:

1. System documentation: This document provides an overview of the system's architecture, functionality, and user interface. It also includes instructions on how to install, configure, and use the system.
2. User manual: This document provides step-by-step instructions on how to use the system. It includes screenshots and illustrations to help users navigate the system.
3. Technical specifications: This document provides detailed information on the system's technical specifications, including hardware and software requirements, programming languages and frameworks used, and database schema.
4. Security policy: This document outlines the system's security policies and procedures, including encryption standards, access control measures, and incident response protocols.

All of these documents can be hosted on a central documentation site or in a repository accessible to all stakeholders involved in the development, deployment, and use of the fees payment and management system.

TWO

REQUIREMENTS OVERVIEW

The software requirements define the system from a blackbox/interfaces perspective. They are split into the following sections:

- User Interfaces - *User Interfaces*
- Technical Interfaces - *Technical Interfaces*
- Runtime Interfaces and Constraints - *Technical Constraints / Runtime Interface Requirements*

User Interfaces:

1. The system should provide a user-friendly web-based interface for students to view their fee balance, make online payments, and receive receipts and confirmation of payments.
2. The system should provide a dashboard for administrators to monitor fee collections and generate reports on fee payments, overdue payments, and other relevant data.
3. The system should allow for customization of the user interface to match the branding and design guidelines of the educational institution.

Technical Interfaces:

1. The system should integrate with payment gateways such as PayPal, Stripe, and other commonly used payment methods to process payments securely.
2. The system should be compatible with commonly used web browsers such as Chrome, Firefox, Safari, and Edge, and support multiple devices such as desktops, laptops, tablets, and smartphones.
3. The system should integrate with the educational institution's existing student information system to retrieve relevant student data.

Runtime Interfaces and Constraints:

1. The system should be available 24/7 with minimum downtime to ensure students can make payments and view their fee balance at any time.

2. The system should be scalable and able to handle a high volume of concurrent users without significant degradation of performance.

3. The system should be designed to handle security threats such as SQL injection, crosssite scripting (XSS), and cross-site request forgery (CSRF) to protect user data and payment information.

Technical Constraints / Runtime Interface Requirements:

1. The system should be developed using programming languages such as PHP, HTML, CSS, and JavaScript.
2. The system should be hosted on a web server such as Apache or Nginx, with a backend database management system such as MySQL or PostgreSQL.
3. The system should be designed to be compatible with commonly used operating systems such as Linux, macOS, and Windows.

THREE

TODO:

Todo: Neighboring Systems:

1. Student Information System (SIS): The fees payment and management system should integrate with the educational institution's SIS to retrieve relevant student data such as enrollment status, program details, and fee structure.
2. Payment Gateway: The fees payment and management system should integrate with one or more payment gateways such as PayPal or Stripe to process payments securely and efficiently.

Logical Business Data Exchanged:

1. Student data: The fees payment and management system should exchange student data such as name, ID, program, and enrollment status with the SIS.
2. Fee structure: The fees payment and management system should receive fee structure information from the SIS and use it to calculate fee balances for individual students.
3. Payment information: The fees payment and management system should exchange payment information such as amount, method, and transaction ID with the payment gateway to process payments securely and efficiently.
4. Receipts and confirmations: The fees payment and management system should provide students with receipts and confirmations of payments made, which may include transaction IDs, payment amounts, and timestamps.

The fees payment and management system is designed to operate within an educational institution's existing infrastructure, integrating with the SIS and payment gateway to provide a seamless and efficient user experience for students and administrators. By exchanging relevant business data with these neighboring systems, the fees payment and management system can provide accurate fee balances, process payments securely, and provide timely receipts and confirmations of payments made.

Motivation.

Understanding the information exchange with neighboring systems (i.e. all input flows and all output flows).

1. Todo: Follow a consistent code style: Use a consistent code style throughout the project, such as using consistent indentation, variable naming conventions, and commenting practices. This will make it easier for other developers to read and understand your code.
2. Write modular code: Divide the code into small, self-contained modules that perform a single function. This will make the code easier to read, test, and maintain.
3. Write readable code: Write code that is easy to read and understand. Use meaningful variable and function names, write clear and concise comments, and avoid using obscure language features.
4. Write secure code: When writing code that deals with sensitive data such as payment information, make sure to follow security best practices to prevent unauthorized access and data breaches.
5. Use version control: Use a version control system such as Git to manage the code and track changes. This will make it easier to collaborate with other developers and keep track of changes made to the code over time.
6. Write testable code: Write code that is easy to test, using automated testing frameworks to ensure that the code is functioning correctly and prevent regression bugs.

Constraint	**Description**
Security	Developers should adhere to strict security standards to protect sensitive data such as payment information. This may include using encryption to protect data at rest and in transit, following best practices for authentication and authorization, and adhering to regulatory compliance standards such as PCI-DSS.
Reliability	The fees payment and management system must be reliable and available at all times. This may include implementing redundancy and failover mechanisms, monitoring system performance and availability, and following best practices for error handling and recovery.
Maintainability	The system must be easy to maintain and update over time. This may include following coding guidelines, using version control, and implementing automated testing and deployment processes.

Interoperability	The system must be able to interact with other systems and services as needed. Developers should adhere to industry standards and best practices for APIs and data exchange formats to ensure seamless integration with other systems.
Compatibility	The system must be compatible with a range of hardware and software environments. Developers should test the system on a variety of operating systems, browsers, and devices to ensure compatibility.
Performance	The system must be able to handle a high volume of transactions and data processing. Developers should optimize code and infrastructure to ensure efficient use of system resources and minimize latency.

Todo: List all technical constraints in this section. This category covers runtime interface requirements and constraints such as:

- Hard- and software infrastructure
- Applied technologies - Operating systems - Middleware - Databases - Programming languages

Here are some technical constraints for the development of a fees payment and management system:

1. Hardware infrastructure: The system may have specific hardware requirements, such as a minimum amount of memory or storage capacity, that need to be considered during development.
2. Software infrastructure: The system may require specific software components, such as web servers or application servers, that need to be installed and configured correctly.

3. Operating systems: The system may need to run on specific operating systems, such as Windows, Linux, or macOS.
4. Middleware: The system may depend on specific middleware components, such as message queues or caching systems, that need to be integrated into the system.
5. Databases: The system may depend on specific databases, such as MySQL or PostgreSQL, that need to be installed and configured correctly.
6. Programming languages: The system may be developed using specific programming languages, such as Java, Python, or JavaScript, that developers need to be familiar with.
7. Security protocols: The system may require the use of specific security protocols, such as SSL/TLS or OAuth, that need to be implemented correctly to ensure the security of the system.
8. Scalability: The system may need to be designed to handle a large volume of transactions and users, requiring developers to consider issues such as load balancing and distributed architectures.
9. Integration with third-party systems: The system may need to integrate with third-party systems, such as payment gateways or accounting software, which may have specific technical requirements and constraints.
10. Performance requirements: The system may have specific performance requirements, such as response times or throughput, that need to be met for the system to function correctly.

Todo: For all your interfaces, define their first 3 levels of interoperability. You can use your doxygen documented source code to i.e. show all members of a class. Find more on the Breathe Documentation

---

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/roboy-swdocumentationtemplate/checkouts/lite/documentation/ScopeContext/3_technical_interfaces.rst, line 8.)

Todo: If you have a user interacting with the finished system, and especially if you have a UI or GUI, describe how it can be used. A good way of doing this, is by building a

- state transition diagram aka the ‘Dynamic UI Behaviuour’ and
- a mockup/picture of every screen the user can see - aka ‘Static UI’

State transition diagram:

A state transition diagram, also known as a state machine, is a visual representation of the possible states and transitions of a system. It can be used to describe the dynamic behavior of a UI or GUI, showing how users can navigate between different screens and interact with different components. The diagram typically includes states, events, and transitions, with each state representing a particular screen or user interface element, each event representing a user action or system event, and each transition representing a change in state based on an event.

Static UI mockups:

A static UI mockup is a visual representation of the user interface of a software system, typically in the form of a set of screen designs or wireframes. It can be used to describe the static behavior of the UI or GUI, showing the layout, structure, and visual design of the different screens and components. The mockup typically includes detailed designs of each screen, with annotations or notes describing the functionality of each component and how it can be used by the user.

Todo: If you want to use it, keep track of decisions that someone further developing this software or using it in the future might want to know about. This might save them time or clarify expectations. You can put them as a list, or whatever.

Things to consider:

- What exactly is the challenge?
- Why is it relevant for the architecture?
- What consequences does the decision have?
- Which constraints do you have to keep in mind?
- What factors influence the decision?
- Which assumption have you made?
- How can you check those assumptions?
- Which risks are you facing?
- Which alternative options did you consider?
- How do you judge each one?
- Which alternatives are you excluding deliberately?
- Who (if not you) has decided?
- How has the decision been justified?
- When did you decide?

Todo: Describe the installation process step by step.

1. Download the installation package from the designated source or obtain it from the system administrator.
2. Double-click the installation file to launch the installation wizard.
3. Follow the instructions provided by the installation wizard to choose the installation location, select the components to install, and configure any necessary settings.
4. Review and accept the license agreement if prompted.
5. Wait for the installation process to complete.
6. If necessary, restart your computer.
7. Launch the application and verify that it is working correctly.

Chapter 3.

Todo: How do you start the software after installation

The process for starting the fees payment and management system may vary depending on the specific operating system and installation configuration, but here are some general steps:

1. Locate the application shortcut or executable file on your computer. This may be on your desktop, in your Start menu, or in a designated installation folder.
2. Double-click the shortcut or executable file to launch the application.
3. If prompted, enter any necessary login credentials to access the system.
4. Once the application is launched, navigate to the desired features and functions as needed.

Todo: Describe your solution strategy.

Contents.

A short summary and explanation of the fundamental solution ideas and strategies.

Motivation.

An architecture is often based upon some key solution ideas or strategies. These ideas should be familiar to everyone involved into the architecture.

Form.

Diagrams and / or text, as appropriate. Keep it short, i.e. 1 or 2 pages at most!

1. Identifying the problem or need: The first step in any software project is to identify the problem or need that the system is intended to address. This might involve gathering input from stakeholders or conducting research to determine the root cause of a problem. For the fees payment and management system, the problem might be inefficient or error-prone processes for managing fees payments, leading to delays or inaccuracies.
2. Gathering requirements: Once the problem or need has been identified, the next step is to gather requirements from stakeholders and end-users. This might involve conducting interviews, surveys, or focus groups to understand the desired functionality, user interface, and performance requirements. For the fees payment and management system, requirements might include the ability to accept various types of payments, generate receipts and reports, and provide secure access for different types of users.
3. Designing the system architecture and user interface: With the requirements in hand, the next step is to design the system architecture and user interface. This might involve creating diagrams and wireframes to visualize the system components and user interactions. For the fees payment and management system, this might include designing a database schema to store payment data, creating a web-based interface for users to access the system, and developing an admin interface for managing system settings.
4. Selecting appropriate technologies: Once the system design has been defined, the next step is to select appropriate technologies, programming languages, and development tools to support the project. This might involve evaluating different options based on factors such as cost, scalability, and ease of use. For the fees payment and management system, this might involve selecting a web framework like Django, choosing a payment processing service like Stripe, and selecting a hosting provider like AWS.
5. Developing and testing the software code: With the design and technology stack in place, the next step is to develop and test the software code. This might involve using an agile development methodology to iteratively build and test the system, using automated testing tools to ensure functional and performance requirements are met. For the fees payment and management system, this might involve creating payment processing workflows, implementing user authentication and access controls, and ensuring the system is secure and reliable.
6. Deploying and maintaining the system: Once the software code has been developed and tested, the final step is to deploy the system to production environments and provide ongoing support and maintenance. This might involve setting up production servers, monitoring system performance, and addressing any bugs or issues that arise. For the fees payment and management system, ongoing maintenance might include updating payment processing APIs, fixing any security vulnerabilities, and providing user support as needed.

Todo: If you have any tests describe:

- Where the tests are kept.
- How the tests are invoked.
- What you can/need to test manually.

1. Test Location: The tests should be kept in a separate directory or folder in the project directory. This makes it easy to locate and execute the tests. It is also important to ensure that the test files are not mixed up with the source code files.
2. Test Invocation: The tests can be invoked using a test runner tool, which can execute the test cases automatically and report the results. There are several test runner tools available for different programming languages such as JUnit for Java, pytest for Python, Mocha for JavaScript, etc.

3. Manual Testing: In addition to automated testing, it is also important to perform manual testing of the software. Manual testing involves executing the software manually and testing its functionality, usability, and performance. Manual testing can be done by following a test plan or test cases that are prepared in advance.

Todo: Inset a building block view:

- Static decomposition of the system into building blocks and the relationships thereof. • Description of libraries and software used
-

We specify the system based on the blackbox view from *UML System Context* by now considering it a whitebox and identifying the next layer of blackboxes inside it. We re-iterate this zoom-in until specific granularity is reached - 2 levels should be enough.

Motivation.

Software Documentation template, Release 0.0.

This is the most important view, that must be part of each architecture documentation. In building construction this would be the floor plan.

Tool

- Create diagrams as below.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/roboy-swdocumentationtemplate/checkouts/lite/documentation/development/06_building_block_view.rst, line 6.)

Todo:

- Define your groups using the /defgroup doxygen command
- Add @addtogroup tags to doxygen blocks of components in code as described here: http://www.stack.nl/~dimitri/doxygen/manual/grouping.html#modules
- Adapt the doxygencall to match the group name

Todo:

- Define your groups using the /defgroup doxygen command
- Add addtogroup tags to doxygen blocks of components in code as described here: http://www.stack.nl/~dimitri/ doxygen/manual/grouping.html#modules
- Adapt the doxygencall to match the group name

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/roboy-swdocumentationtemplate/checkouts/lite/documentation/development/06_building_block_view.rst, line 55.)

Todo: Add a runtime view of i.e. the startup of your code. This should help people understand how your code operates and where to look if something is not working.

Contents. alternative terms: - Dynamic view - Process view - Workflow view This view describes the behavior and interaction of the system's building blocks as runtime elements (processes, tasks, activities, threads, ...).

Select interesting runtime scenarios such as:

- How are the most important use cases executed by the architectural building blocks?
- Which instances of architectural building blocks are created at runtime and how are they started, controlled, and stopped.
- How do the system's components co-operate with external and pre-existing components?

- How is the system started (covering e.g. required start scripts, dependencies on external systems, databases, communications systems, etc.)?

  Note

  The main criterion for the choice of possible scenarios (sequences, workflows) is their architectural relevancy. It is not important to describe a large number of scenarios. You should rather document a representative selection.

  Candidates are:

  1. The top 3 – 5 use cases
  2. System startup
  3. The system's behavior on its most important external interfaces
  4. The system's behavior in the most important error situations Motivation.

  Especially for object-oriented architectures it is not sufficient to specify the building blocks with their interfaces, but also how instances of building blocks interact during runtime.

  Form.

  Document the chosen scenarios using UML sequence, activity or communications diagrams. Enumerated lists are sometimes feasible.

Using object diagrams you can depict snapshots of existing runtime objects as well as instantiated relationships. The UML allows to distinguish between active and passive objects.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/roboy-swdocumentationtemplate/checkouts/lite/documentation/development/07_runtime_view.rst, line 6.)

Todo: Add a deployment diagram. Contents.

This view describes the environment within which the system is executed. It describes the geographic distribution of the system or the structure of the hardware components that execute the software. It documents workstations, processors, network topologies and channels, as well as other elements of the physical system environment.

Motivation.

Software is not much use without hardware. These models should enable the operator to properly install the software.

You can separate this into different levels...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/roboy-swdocumentationtemplate/checkouts/lite/documentation/development/08_deployment_view.rst, line 6.)

Todo: List libraries you are using

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/roboy-swdocumentationtemplate/checkouts/lite/documentation/development/9_Libraries.rst, line 8.)

Todo: Introduce your project and describe what its intended goal and use is.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/roboy-swdocumentationtemplate/checkouts/lite/documentation/index.rst, line 10.)

Todo: Describe what a potential user needs to be familiar with. What should they read and understand beforehand Describe what a developer needs to be familiar with to further understand the code.

Link to relevant documents or create a new page and add them there.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/roboy-swdocumentationtemplate/checkouts/lite/documentation/index.rst, line 19.) Contents:

3.1 Installation

Todo: Describe the installation process step by step.

3.2 Getting started

Todo: How do you start the software after installation

3.3 Context

Todo: Create a black box view of your software within its intendend environment. Identify all neighboring systems and specify all logical business data that is exchanged with the system under development.

List of all (a-l-l) neighboring systems.

Motivation.

Understanding the information exchange with neighboring systems (i.e. all input flows and all output flows).

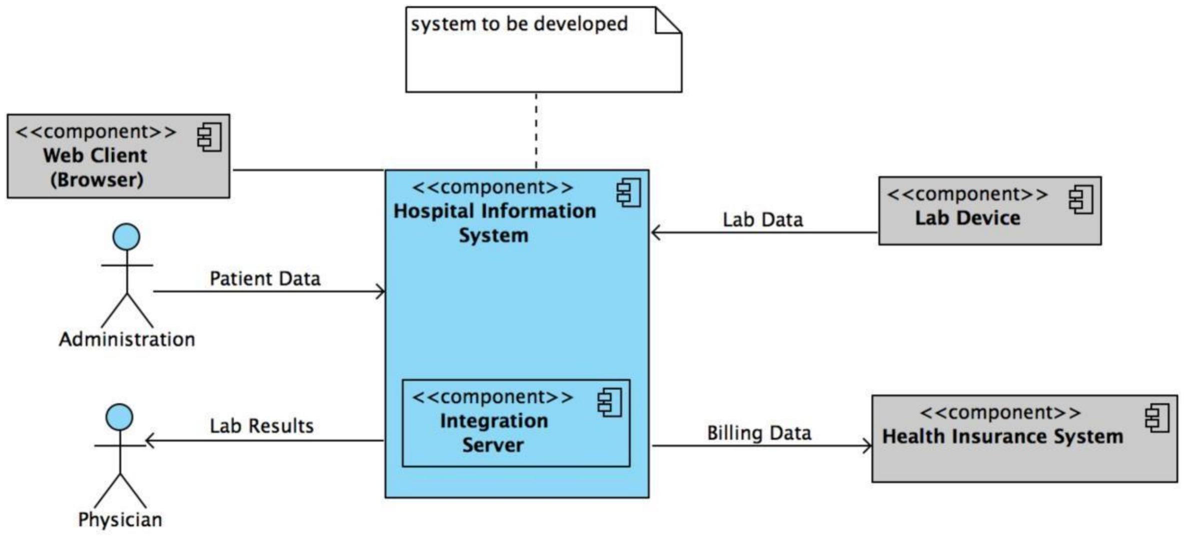

system to be developed
<<component>> Web Client (Browser)
<<component>> Hospital Information System
<<component>> Integration Server
<<component>> Lab Device
<<component>> Health Insurance System
Lab Data
Patient Data
Administration
Lab Results
Physician
Billing Data

Fig. 1: UML System Context

UML-type context diagram - shows the birds eye view of the system (black box) described by this architecture within the ecosystem it is to be placed in. Shows orbit level interfaces on the user interaction and component scope.

3.4 Conventions

We follow the coding guidelines:

Todo: Define what coding guideline they should use, if you differ from the ones below.

Table 1: Coding Guidelines

Language	Guideline	Tools
Python	https://www.python.org/dev/peps/pep-0008/	
C++	http://wiki.ros.org/CppStyleGuide	clang-format: https://github.com/davetcoleman/roscpp_code_format

3.5 Architecture Constraints

Todo: Describe what constraints someone further developing this software should adhere to, and why. Should they not use tool x or operating system y... You can use the table as below, or just put a list.

3.5.1 Technical Constraints / Runtime Interface Requirements

Todo: List all technical constraints in this section. This category covers runtime interface requirements and constraints such as:

- Hard- and software infrastructure
- Applied technologies - Operating systems - Middleware - Databases - Programming languages

Table 2: Hardware Constraints

Constraint Name	Description
Altera FPGA	All code is highly specific to the Altera FPGA. Intel has bought Altera and aims to integrate their SoC with FPGAs. We are on the right horse!
Intel RealSense	Only the intel real sense can sense it..

Table 3: Software Constraints

Constraint Name	Description
Altera FPGA	All code is highly specific to the Altera FPGA. Intel has bought Altera and aims to integrate their SoC with FPGAs. We are on the right horse!

Intel RealSense	Only the intel real sense can sense it...

Table 4: Operating System Constraints

Constraint Name	Description
Windows 8 or higher	Due to the Intel RealSense SDK only being supported on Windows, we are stuck with Windows

Table 5: Programming Constraints

Constraint Name	Description
CouchDB	We have to use the CouchDB because the type of data we have to store changes at runtime...

3.5. Architecture Constraints

3.6 Technical Interfaces

This section describes the data interfaces to other systems around it. It follows 3 of the levels of interoperability (IO):

Todo: For all your interfaces, define their first 3 levels of interoperability. You can use your doxygen documented source code to i.e. show all members of a class. Find more on the Breathe Documentation

- Technical interoperability - Datastreams btwn systems. i.e. TCP/IP, RS232, ...
- Syntactic interoperability - Units within the stream. i.e. XML, CSV, HL7, DICOM
- Semantic interoperability - Common definition of unit meaning.

3.6.1 Powerlink for MotorControl

Technical IO

Powerlink

Syntactic IO

SDO, PDO, NMT messages

Semantic IO

The package names or registers or **class**

Nutshell

Public Types **enum**

Tool

Our tool set.

The various tools we can opt to use to crack this particular nut *Values:*

enumerator kHammer = 0

What? It does the job. **enumerator**

kNutCrackers Boring. **enumerator**

kNinjaThrowingStars Stealthy.

Public Functions

Nutshell()
Nutshell constructor.

~Nutshell()
Nutshell destructor.

void **crack**(*Tool tool*)
Crack that shell with specified tool.

Parameters

- tool: - the tool with which to crack the nut

bool **isCracked**()

Return Whether or not the nut is cracked

3.7 User Interfaces

How does a user interact with the system.

Todo: If you have a user interacting with the finished system, and especially if you have a UI or GUI, describe how it can be used. A good way of doing this, is by building a

- state transition diagram aka the 'Dynamic UI Behaviuour' and
- a mockup/picture of every screen the user can see - aka 'Static UI'

Don't overdo it...

3.7.1 Dynamic UI Behaviour

3.7.2 Static UI

3.8 Design Decisions

This can document decisions on the design of the software.

Todo: If you want to use it, keep track of decisions that someone further developing this software or using it in the future might want to know about. This might save them time or clarify expectations. You can put them as a list, or whatever.

Things to consider:

- What exactly is the challenge?

- Why is it relevant for the architecture?
- What consequences does the decision have?
- Which constraints do you have to keep in mind?
- What factors influence the decision?

3.7. User Interfaces

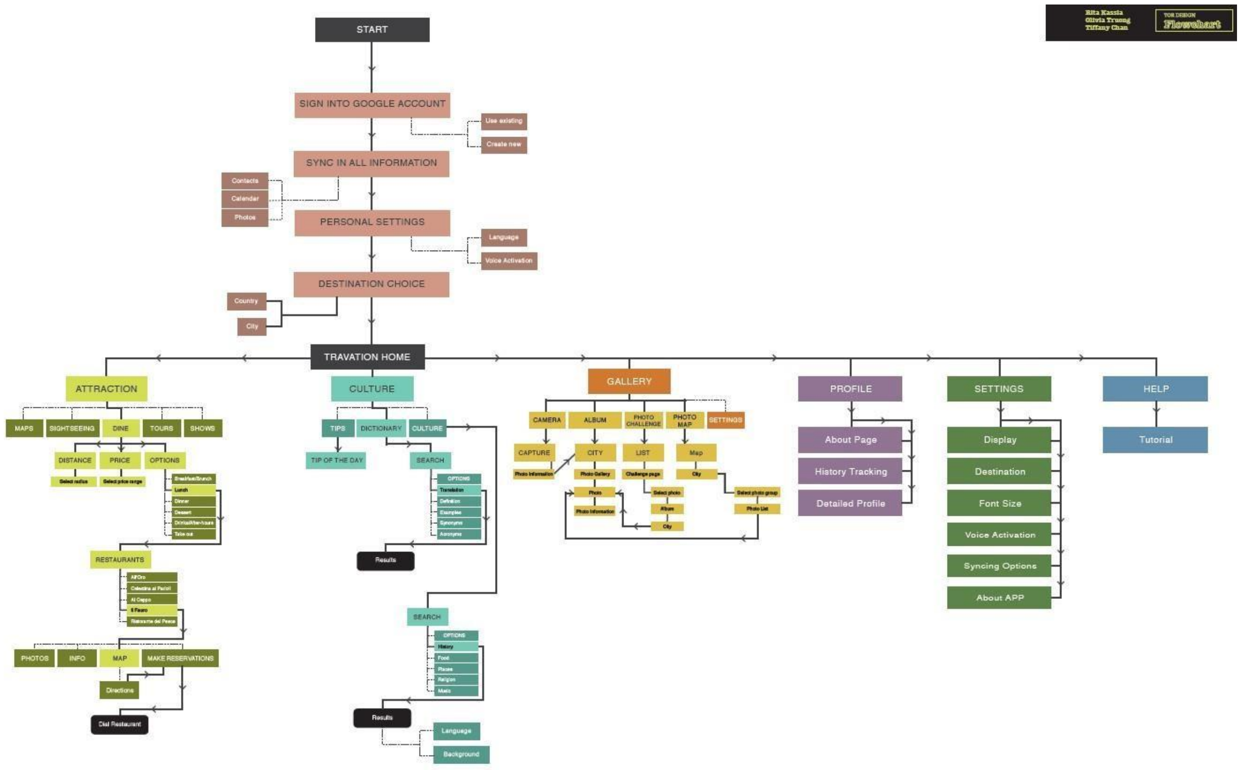


Fig. 2: Diagram showing the dynamic behaviour of the user interface i.e. State diagram

- Which assumption have you made?
- How can you check those assumptions?
- Which risks are you facing?
- Which alternative options did you consider?
- How do you judge each one?
- Which alternatives are you excluding deliberately?
- Who (if not you) has decided?
- How has the decision been justified?
- When did you decide?

3.9 Public Interfaces

class Nutshell

Fig. 3: Mock-up screens of the individual views/screens of the GUI i.e. Wireframes, whiteboard sketches

3.9. Public Interfaces

Public Types enum

Tool

Our tool set.

The various tools we can opt to use to crack this particular nut

Values:

enumerator kHammer = 0
What? It does the job.

enumerator kNutCrackers
Boring.

enumerator kNinjaThrowingStars
Stealthy.

Public Functions

Nutshell() *Nutshell*
constructor.

~Nutshell()
Nutshell destructor.

void **crack**(*Tool tool*)
Crack that shell with specified tool.

Parameters

- tool: - the tool with which to crack the nut bool

isCracked()

Return Whether or not the nut is cracked

Private Members

bool **m_isCracked**
Our cracked state.

file **nutshell.h**
An overly extended example of how to use breathe.

Enums

enum [anonymous] *Values:*

enumerator YellowRoller

enumerator RedRoller enumerator

GreenRoller enum

[anonymous] *Values:*

enumerator Chocolate enumerator

Mossy enumerator CremeFraiche

group **nuttygroup**
The group for all nutjobs.

No nut can withstand us!

Enums

enum [anonymous] *Values:*

enumerator YellowRoller

enumerator RedRoller enumerator

GreenRoller

group **nuttygroup2**

The group for all nutjobs.

More documentation for the second group.

Enums

enum [anonymous] *Values:*

enumerator Chocolate enumerator

Mossy enumerator CremeFraiche

dir **/home/docs/checkouts/readthedocs.org/user_builds/roboy-sw-documentation-template/checkou**

3.10 Solution Strategy

Todo: Describe your solution strategy.

Contents.

A short summary and explanation of the fundamental solution ideas and strategies.

Motivation.

An architecture is often based upon some key solution ideas or strategies. These ideas should be familiar to everyone involved into the architecture.

Form.

Diagrams and / or text, as appropriate. Keep it short, i.e. 1 or 2 pages at most!

3.11 Test Strategy

Todo: If you have any tests describe:

- Where the tests are kept.
- How the tests are invoked.

- What you can/need to test manually.

3.11.1 System Tests

3.11.2 Integration Tests

3.11.3 Unit Tests

3.12 Building Block View

3.12.1 Overview

Todo: Inset a building block view:

- Static decomposition of the system into building blocks and the relationships thereof. • Description of libraries and software used
-

We specify the system based on the blackbox view from *UML System Context* by now considering it a whitebox and identifying the next layer of blackboxes inside it. We re-iterate this zoom-in until specific granularity is reached - 2 levels should be enough.

Motivation.

This is the most important view, that must be part of each architecture documentation. In building construction this would be the floor plan.

Tool

- Create diagrams as below.

The white box view of the first level of your code. This is a white box view of your system as shown within the in Context in figure: *UML System Context*. External libraries and software are clearly marked.

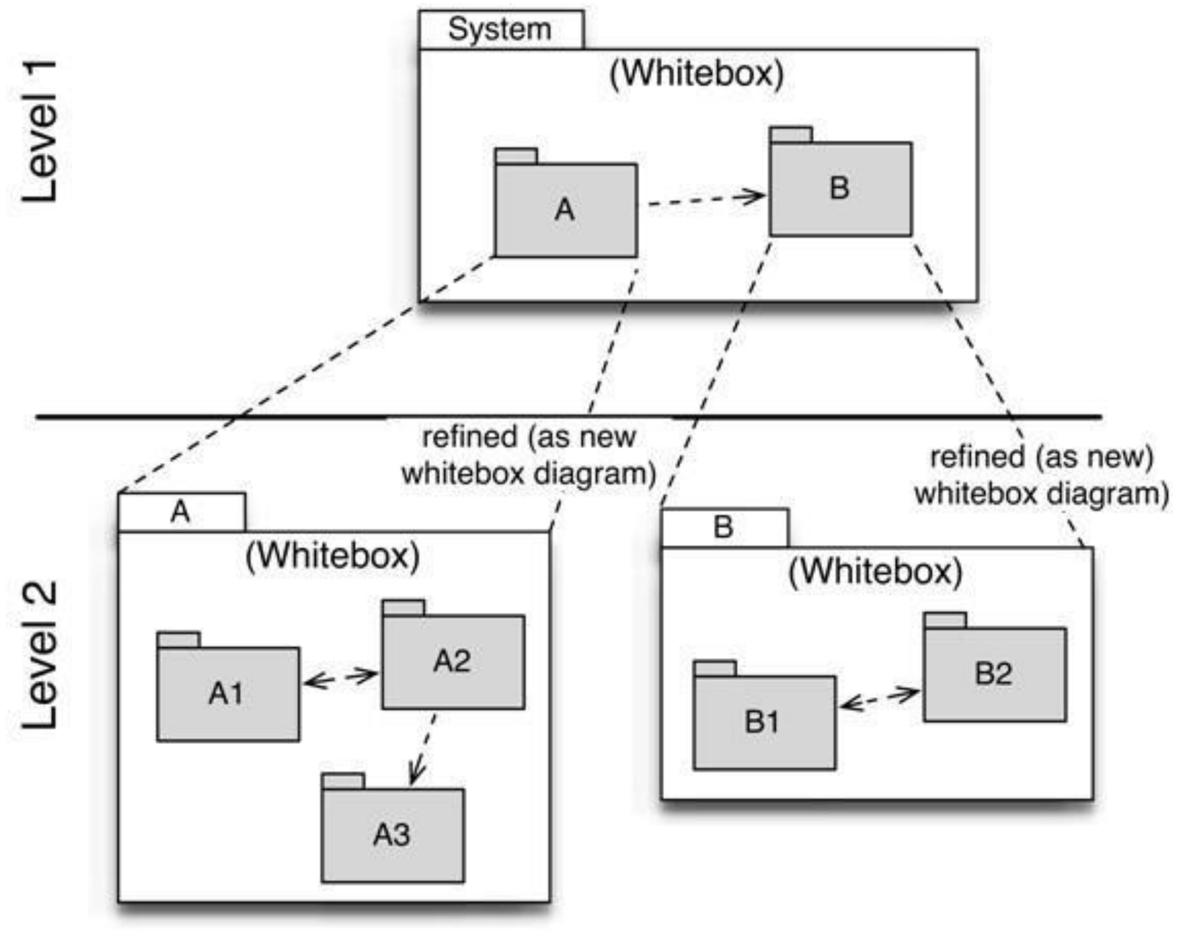

Level 1
System
(Whitebox)
A
B
refined (as new whitebox diagram)
refined (as new) whitebox diagram)
Level 2
A
(Whitebox)
A1
A2
A3
B
(Whitebox)
B1
B2

Fig. 4: Building blocks overview

3.12.2 Level 1 - Components

The highest level components

Todo:

- Define your groups using the /defgroup doxygen command

- Add @addtogroup tags to doxygen blocks of components in code as described here: http://www.stack.nl/~dimitri/doxygen/manual/grouping.html#modules
- Adapt the doxygencall to match the group name

---

group **nuttygroup** The group for all nutjobs.

No nut can withstand us!

Enums

enum [anonymous] *Values:*

enumerator YellowRoller

enumerator RedRoller enumerator

GreenRoller class Nutshell

3.12.3 Level 2 - Components within each component of level 1

Todo:

- Define your groups using the /defgroup doxygen command
- Add addtogroup tags to doxygen blocks of components in code as described here: http://www.stack.nl/~dimitri/doxygen/manual/grouping.html#modules
- Adapt the doxygencall to match the group name

---

group **nuttygroup2**

The group for all nutjobs.

More documentation for the second group. **Enums**

enum [anonymous] *Values:*

enumerator Chocolate enumerator

Mossy enumerator CremeFraiche

3.13 Runtime View

Todo: Add a runtime view of i.e. the startup of your code. This should help people understand how your code operates and where to look if something is not working.

Contents. alternative terms: - Dynamic view - Process view - Workflow view This view describes the behavior and interaction of the system’s building blocks as runtime elements (processes, tasks, activities, threads, ...).

Select interesting runtime scenarios such as:

- How are the most important use cases executed by the architectural building blocks?
- Which instances of architectural building blocks are created at runtime and how are they started, controlled, and stopped.
- How do the system's components co-operate with external and pre-existing components?
- How is the system started (covering e.g. required start scripts, dependencies on external systems, databases, communications systems, etc.)?

Note

The main criterion for the choice of possible scenarios (sequences, workflows) is their architectural relevancy. It is not important to describe a large number of scenarios. You should rather document a representative selection.

Candidates are:

1. The top 3 – 5 use cases
2. System startup
3. The system's behavior on its most important external interfaces
4. The system's behavior in the most important error situations Motivation.

Especially for object-oriented architectures it is not sufficient to specify the building blocks with their interfaces, but also how instances of building blocks interact during runtime.

Form.

Document the chosen scenarios using UML sequence, activity or communications diagrams. Enumerated lists are sometimes feasible.

Using object diagrams you can depict snapshots of existing runtime objects as well as instantiated relationships. The UML allows to distinguish between active and passive objects.

3.13. Runtime View

3.13.1 Runtime Scenario 1

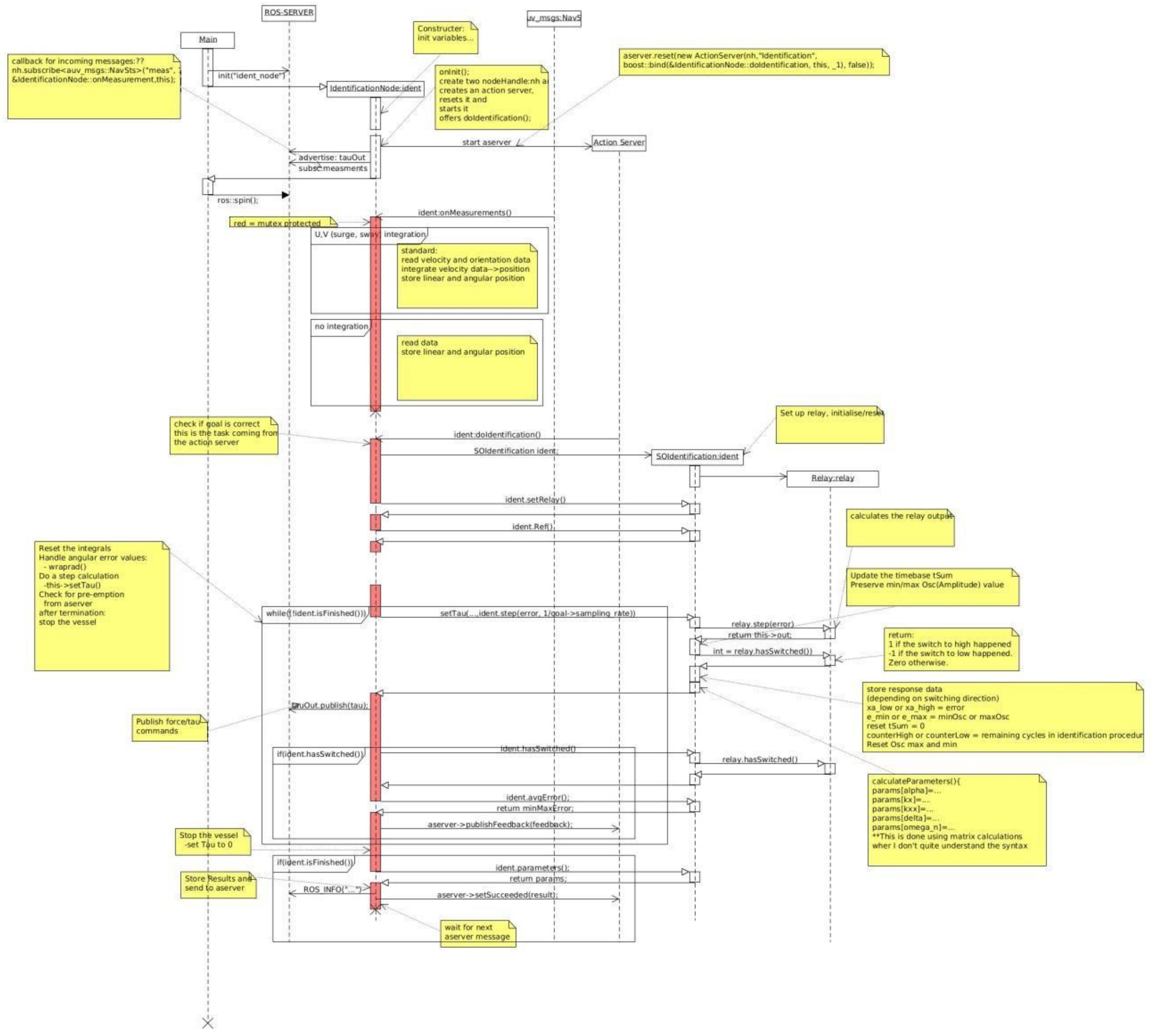

ROS-SERVER
uv_msgs:NavS
Main
Constructer:
init variables...
callback for incoming messages:??
nh.subscribe<auv_msgs::NavSts>("meas",
&IdentificationNode::onMeasurement,this);
init("ident_node")
IdentificationNode:ident
onInit();
create two nodeHandle:nh a
creates an action server,
resets it and
starts it
offers doIdentification();
aserver.reset(new ActionServer(nh,"Identification",
boost::bind(&IdentificationNode::doIdentification, this, _1), false));
start aserver
Action Server
advertise: tauOut
subsc:measments
ros::spin();
ident:onMeasurements()
red = mutex protected
U,V (surge, sway) integration
standard:
read velocity and orientation data
integrate velocity data-->position
store linear and angular position
no integration
read data
store linear and angular position
check if goal is correct
this is the task coming from
the action server
ident:doIdentification()
SOIdentification ident;
SOIdentification:ident
Set up relay, initialise/reset
Relay:relay
ident.setRelay()
ident.Ref()
calculates the relay output
Reset the integrals
Handle angular error values:
- wraprad()
Do a step calculation
-this->setTau()
Check for pre-emption
from aserver
after termination:
stop the vessel
Update the timebase tSum
Preserve min/max Osc(Amplitude) value
while(!ident.isFinished())
setTau(...,ident.step(error, 1/goal->sampling_rate))
relay.step(error)
return this->out;
int = relay.hasSwitched())
return:
1 if the switch to high happened
-1 if the switch to low happened.
Zero otherwise.
store response data
(depending on switching direction)
xa_low or xa_high = error
e_min or e_max = minOsc or maxOsc
reset tSum = 0
counterHigh or counterLow = remaining cycles in identification procedur
Reset Osc max and min
tauOut.publish(tau);
Publish force/tau
commands
if(ident.hasSwitched())
ident.hasSwitched()
relay.hasSwitched()
calculateParameters(){
params[alpha]=...
params[kx]=...
params[kxx]=...
params[delta]=...
params[omega_n]=...
**This is done using matrix calculations
wher I don't quite understand the syntax
ident.avgError();
return minMaxError;
aserver->publishFeedback(feedback);
Stop the vessel
-set Tau to 0
if(ident.isFinished())
ident.parameters();
return params;
Store Results and
send to aserver
ROS_INFO("...")
aserver->setSucceeded(result);
wait for next
aserver message

Fig. 5: UML-type sequence diagram - Shows how components interact with each other during runtime. **3.13.2**

Runtime Scenario 2

...

3.14 Deployment View

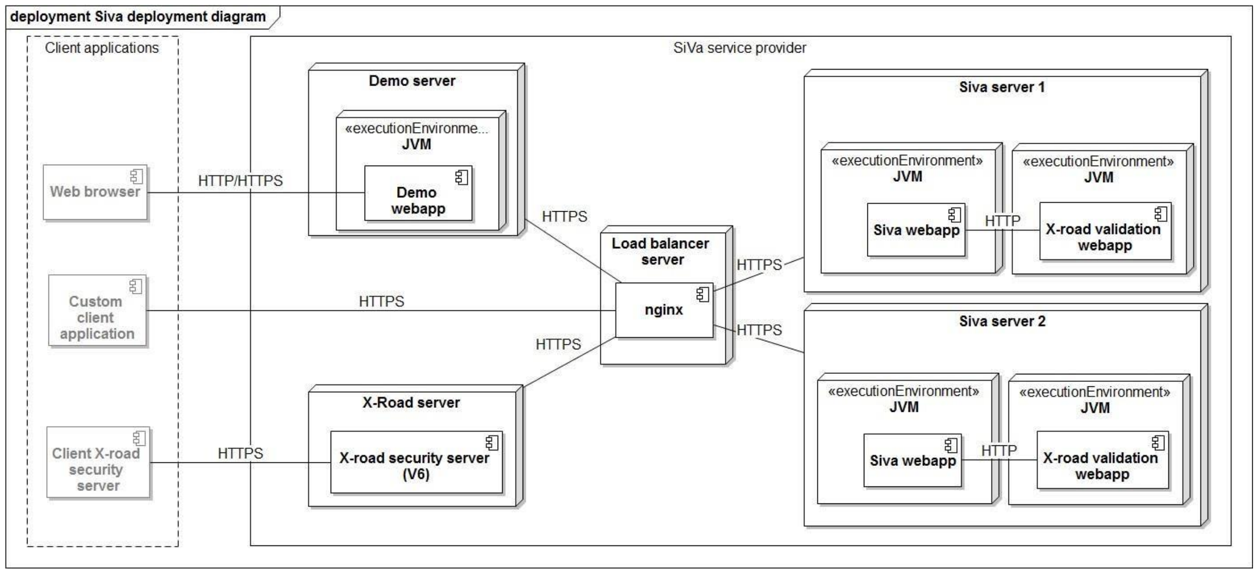
Todo: Add a deployment diagram. Contents.

This view describes the environment within which the system is executed. It describes the geographic distribution of the system or the structure of the hardware components that execute the software. It documents workstations, processors, network topologies and channels, as well as other elements of the physical system environment.

Motivation.

Software is not much use without hardware. These models should enable the operator to properly install the software.

You can separate this into different levels...



3.15 Libraries and external Software

Contains a list of the libraries and external software used by this system.

Todo: List libraries you are using

Table 6: Libraries and external Software

Name	URL/Author	License	Description
arc42	http://www.arc42.de/template/	Creative Commens Attribution license.	Template for documenting and developing software

3.16 About arc42

This information should stay in every repository as per their license: http://www.arc42.de/template/licence.html arc42, the

Template for documentation of software and system architecture.

By Dr. Gernot Starke, Dr. Peter Hruschka and contributors.

Template Revision: 6.5 EN (based on asciidoc), Juni 2014

Note

This version of the template contains some help and explanations. It is used for familiarization with arc42 and the understanding of the concepts. For documentation of your own system you use better the *plain* version.

3.16.1 Literature and references

Starke-2014 Gernot Starke: Effektive Softwarearchitekturen - Ein praktischer Leitfaden. Carl Hanser Verlag, 6, Auflage 2014.

Starke-Hruschka-2011 Gernot Starke und Peter Hruschka: Softwarearchitektur kompakt. Springer Akademischer Verlag, 2. Auflage 2011.

Zörner-2013 Softwarearchitekturen dokumentieren und kommunizieren, Carl Hanser Verlag, 2012

3.16.2 Examples

- HTML Sanity Checker
- DocChess (german)
- Gradle (german)
- MaMa CRM (german)
- Financial Data Migration (german)

3.16.3 Acknowledgements and collaborations

arc42 originally envisioned by Dr. Peter Hruschka and Dr. Gernot Starke.

Sources We maintain arc42 in *asciidoc* format at the moment, hosted in GitHub under the aim42-Organisation.

Issues We maintain a list of open topics and bugs.

We are looking forward to your corrections and clarifications! Please fork the repository mentioned over this lines and send us a *pull request*!

3.16.4 Collaborators

We are very thankful and acknowledge the support and help provided by all active and former collaborators, uncountable (anonymous) advisors, bug finders and users of this method.

Currently active

- Gernot Starke
- Stefan Zörner
- Markus Schärtel
- Ralf D. Müller
- Peter Hruschka
- Jürgen Krey

Former collaborators

(in alphabetical order)

- Anne Aloysius
- Matthias Bohlen
- Karl Eilebrecht
- Manfred Ferken
- Phillip Ghadir
- Carsten Klein
- Prof. Arne Koschel
- Axel Scheithauer

3.16. About arc42

INDEX