Software development for checklist system

Kim Langvannskås   
Software Engineering   
University of South-Eastern NorwayPorsgrunn, Norway  
242402@usn.no

Abstract

Fire stations frequently perform equipment counts and maintenance to ensure readiness for emergencies. All equipment should be ready to use and in the correct position when needed. Often some equipment is in use or out for maintenance. Thus, checklists are gone through for redundancy making sure there are no deviations. This results in the accumulation of large, non-ecofriendly paper archives that are cumbersome to manage and inefficient in terms of time and resources. To address these challenges, this paper proposes the digitalization of checklists to increase efficiency and promote environmental sustainability.

Keywords: Digitalization, digital checklists, time efficient

# Introduction

Fire stations are vital for responding to emergencies, and ensuring equipment is ready is key. However, using paper checklists for equipment readiness has its challenges. This paper explores the shift from paper to digital checklists in fire stations, using the Unified Process framework to guide the transition.

# Methods

## Implementation

The application was developed using the Microsoft .NET framework and C# programming language. Visual Studio IDE was utilized for coding, debugging, and testing purposes. Microsoft SQL Server is used for data storage. Microsoft Excel is used for creating templates.

## Modeling

Some of the models in the paper are created using umletino.com.

# Results

A specification designed earlier in the inception phase is the starting point for the elaboration phase. This chapter contains 1 iteration of the elaboration phase and the construction phase.

## Elaboration phase

Functional and non-functional requirements are shown in table 1. From the requirements a use case diagram and a domain model has been created. These are shown respectively in Figure III‑1 and Figure III‑2.

Table 1 Requirements

|  |  |
| --- | --- |
| Functional requirements | * Take user input for filling checklist. * Upload checklist to database. * Indicate errors as alarm for supervisor. * Display contents for supervisor. * Let supervisor view and acknowledge alarms. * Add date and current shift automatically. * Allow for user to select different checklists. |
| Usability: | User interface needs to be as simple as one on paper. It should be intuitive to select checklist and fill in the boxes. |
| Reliability: | All data should be available for supervisor to view at any time. Checklists should be available for users to fill Friday through Sunday. |
| Performance: | The system needs to operate smoothly and be quick in loading phase. |
| Supportability: | The system needs to be scalable for further expansions. |

There should only be one iteration of the elaboration phase. This iteration should take no longer than one week.

There are numerous different architectural structures that a software application can be built upon. Two-tier architectures include two layers: a presentation layer, which in this case will be the user interface where the firefighters can interact with the checklist and the supervisor can acknowledge alarms. A data layer, database where the checklists, user information etc... will be stored.

Other structures like the tree-tier architecture could improve the system making it more robust and scalable. In the tree-tier architecture an application tier will handle the business logic. For this case that would include managing and validating user inputs and coordinating interactions between the presentation and data tiers.

Four-tier architecture is commonly used for larger systems. Here we divide the application tier from tree-tier architecture into application and business layer. For this system that would mean the application layer handles the service with GRASP pattern control while the business layer handles the business logic.

For this system a two-tier architecture should be sufficient for its purpose. Keeping a simple structure will also lower the workload making it more likely that the system will be implemented.

ChatGPT has been used to make the following requirements:

Functional Requirements:

User Interface:

- Intuitive interface for checklist completion.

- Supervisors can review checklists.

- Support on laptops and tablets.

Checklist Management:

- Pre-made checklist selection.

- Easy completion with comments and signatures.

- Automatic inclusion of date, shift, and user info.

- Archived completed checklists.

Notifications:

- Alert supervisors of deviations.

- Email or in-app notifications.

User Management:

- Authentication and authorization.

- Admin user account management.

Non-Functional Requirements:

Performance:

- Responsive interaction.

- Minimal latency.

Security:

- Robust authentication and encryption.

- Compliance with security standards.

Reliability:

- High availability.

- Data backup and recovery.

Scalability:

- Accommodate growing users and checklists.

Usability:

- User-friendly interface.

- Clear instructions for all users.

Accessibility:

- Accessibility for users with disabilities.

- Compliance with accessibility standards.

Compatibility:

- Support for various browsers and devices.

Maintainability:

- Clean and modular code.

- Documentation for future updates.

ChatGPT provides a thorough list of requirements, encompassing both familiar elements and new additions. The functional requirements are well-detailed, offering comprehensive descriptions. However, some of the non-functional requirements may not align perfectly with the intended system operation. ChatGPT's limitations in fully understanding the system's behavior led to some requirements being speculative. As such, while ChatGPT serves as a valuable tool for organizing and inspiring requirements, the final specifications should be crafted by the system developers.

When requested for a domain model, ChatGPT provides the model displayed in Figure III‑3. The model is short compared to the one created earlier but includes a list of attributes for each class in the domain model. Upon adding these attributes, the two domain models are remarkably similar.

For the elaboration phase a first version class diagram have been made to give an overview of the class structure. As shown in **Feil! Fant ikke referansekilden.** the façade pattern is used to wrap the subsystem of the data access layer. The presentation layer will in this case include classes responsible for user interactions. [1]

Figure III‑5, Figure III‑6, Figure III‑7 and Figure III‑8 shows implementation of the C# program and the user interfaces. The user adds user information in the form of employee number and selects template for checklist. The templates are stored as .csv files. This information is then used to generate a new checklist for the user to interact with. The panels, labels and checkboxes in the checklist gets automatically generated based on the content of the .csv files. This simplifies the process of adding new checklist templates in the future. A user with administrator privileges can open former checklists and acknowledge them.

The data layer stores data in a Microsoft SQL Server database. The structure of the database is created with Erwin database modeler. The model from Erwin is shown in Figure III‑9

## Construction phase

The most important use case is “Fill checklist”. This is the origin of the system. The operator should get a user friendly and self-explanatory checklist based on a template. A fully dressed use case diagram of the use case can be seen in Figure III‑11

The domain model shown in Figure III‑2 includes multiple entities. Some of these are classes of super or sub style. The superclasses include system which encompasses all other entities in the system, and checklist presenting a checklist entity either for a template or for a stored checklist.

The subclasses include premade checklist and completed checklists which will be child classes of checklist.

Patterns have been used to make the class diagram shown in Figure III‑10 and the interaction diagram shown in Figure III‑12. The patterns used are shown in table 2.

Table 2 The design pattern [2]

|  |  |
| --- | --- |
| Controller | Create pattern, responsible for creating “Database handler”, “Display” and “Checklist”.  Controller pattern, responsible for controlling the system |
| System | Create pattern, responsible for creating “Controller” |
| Database handler | Low coupling pattern, only sends and collects data with database. |
| Checklist | Polymorphism pattern, can be created with or without admin privileges. |
| Template | Information expert pattern, holds the checklist templates |
| Display | High cohesion, specific responsibility to display checklist. |

Only one checklist should be displayed at a time. Hence an object diagram would be like the class diagram in Figure III‑10.

To test the use case a simple test plan is shown in Table 3. The test shows that not all requirements for this use case are implemented. This is useful information to bring on to the next iteration of the construction phase.

Table 3 test plan [3]

|  |  |  |
| --- | --- | --- |
| 1 | Test user input | OK |
| 2 | Upload checklist to database. | Not yet implemented |
| 3 | Date and current shift are added automatically. | Not yet implemented |
| 4 | User can select different checklists. | OK |

The first iteration of the construction phase has improved the program by giving a structured overview of the architecture for the system. The most important use case have been properly analyzed and used to create interaction diagram and class diagram. Summarized the iteration have been through requirements, analysis & design, implementation, test and evaluation.

# Conclusion

In total the usage of unified process has helped getting a better structure for the system. The total analysis of the system can help multiple developers get the same situational overview. Including some system sequence diagrams for the use case could have help analyze it even more.

##### References

|  |  |
| --- | --- |
| [1] | N.-O. Skeie, "Design pattern principles," Canvas, 01 03 2018. [Online]. Available: https://usn.instructure.com/courses/31275/pages/grasp-and-gof-patterns. [Accessed 01 05 2024]. |
| [2] | K. Langvannskås, "Software design of control system," USN, Porsgrunn, 2024. |
| [3] | N.-O. Skeie, "Summary of first iterations," Canvas, 17 03 2017. [Online]. Available: https://usn.instructure.com/courses/31275/pages/programming-and-testing. [Accessed 01 05 2024]. |