NTNU

Norwegian University of Science and Technology

Master's degree thesis

IP501909 MSc thesis, discipline oriented master

A Knowledge-Based Approach for an Open Object Oriented Library in Ship Design

Candidatenumber: 1117/Author: Thiago Gabriel Monteiro

Number of pages including this page: 139

Aalesund, 03, June 2016

Mandatory statement

Each student is responsible for complying with rules and regulations that relate to examinations and to academic work in general. The purpose of the mandatory statement is to make students aware of their responsibility and the consequences of cheating. **Failure to complete the statement does not excuse students from their responsibility.**

Please complete the mandatory statement by placing a mark ***in each box*** *for statements 1-6 below.*		
1.	**I/we hereby declare that my/our paper/assignment is my/our own work, and that I/we have not used other sources or received other help than is mentioned in the paper/assignment.**	☒
2.	**I/we herby declare that this paper** 1. Has not been used in any other exam at another department/university/university college 2. Is not referring to the work of others without acknowledgement 3. Is not referring to my/our previous work without acknowledgement 4. Has acknowledged all sources of literature in the text and in the list of references 5. Is not a copy, duplicate or transcript of other work	Mark each box: 1. ☒ 2. ☒ 3. ☒ 4. ☒ 5. ☒
3.	**I am/we are aware that any breach of the above will be considered as cheating, and may result in annulment of the examination and exclusion from all universities and university colleges in Norway for up to one year, according to the Act relating to Norwegian Universities and University Colleges, section 4-7 and 4-8 and Examination regulations .**	☒
4.	**I am/we are aware that all papers/assignments may be checked for plagiarism by a software assisted plagiarism check**	☒
5.	**I am/we are aware that NTNU will handle all cases of suspected cheating according to prevailing guidelines.**	☒
6.	**I/we are aware of the NTNU's rules and regulation for using sources.**	☒

Publication agreement

ECTS credits: 120

Supervisor: Henrique Murilo Gaspar

Date: 03/06/2016

Master Thesis in Ship Design

Stud.Techn. Thiago Gabriel Monteiro

"A Knowledge-Based Approach for an Open Object Oriented Library in Ship Design"

Spring 2016

Background

Given the complexity of a vessel, a designer can face several difficulties, which can influence the project's final result. With the competitive maritime market, the old fashion design spiral is not anymore sufficient to provide good and innovative solutions for the ship owners, who have to operate every day with a lower profit margin.

This problem requires a new approach to address the design problem in a different way, allowing innovation to be the main goal. Only by chasing new concepts can a designer make his project stand out from the rest.

The chosen way for doing so is addressing the design problem using the knowledge based system design approach, which will be implemented as an open object oriented library, using JavaScript. The design theory, combined with the ship design principle allows the creation of a tools library to deal with the vessel design problems in a new way, focusing on the system design from the beginning of the design process, not only at its final stages.

Overall aim and focus

The thesis aims to create a knowledge based and open source ship design library to address, mainly, the following topics:

- System Based Design Applied to Ship Design Process;
- Open Source Technology Applied to Ship Design Library Development.

Main activities

During the Pre-Study Phase (Task 1), the following topics will be approached:

- Task 1.1: Knowledge Based System Design Theory
- Task 1.2: Ship Design Performance Indicators Study
- Task 1.3: Open Source Community Concepts

The Model Development Phase (Task 2) will be composed by:

- Task 2.1: Vessel Prototype Definition and Implementation

- Task 2.2: Tools Library Development
- Task 2.3: Open Source Ship Design Library Creation

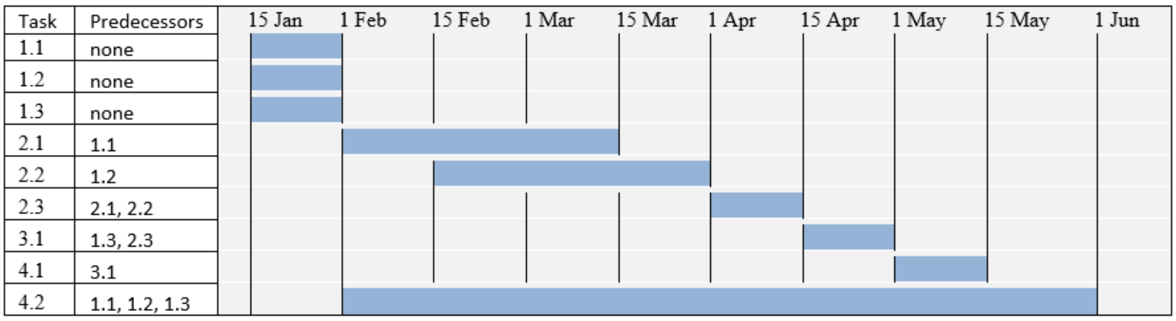
During the Analysis Phase (Task 3), a study will be conducted to evaluate the advantages of the open source platform over the stiffer and traditional ship design methods.

- Task 3.1: Open Source Platform Performance on Ship Design

The Conclusion Phase (Task 4) will address the closure of the project, where time is put in the evaluation of all the findings and in the report preparation.

- Task 4.1: Findings Evaluation
- Task 4.2: Report Preparation

The following Gantt Chart shows how the tasks are organized during the working period from the 15th of January until the 1st of June.



Task	Predecessors	15 Jan	1 Feb	15 Feb	1 Mar	15 Mar	1 Apr	15 Apr	1 May	15 May	1 Jun
1.1	none	🟦									
1.2	none	🟦									
1.3	none	🟦									
2.1	1.1		🟦	🟦	🟦						
2.2	1.2			🟦	🟦	🟦					
2.3	2.1, 2.2						🟦				
3.1	1.3, 2.3							🟦			
4.1	3.1								🟦		
4.2	1.1, 1.2, 1.3		🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	

In addition to the specified tasks, a conference paper will be handled at the end of the research.

[illegible]

Henrique Murilo Gaspar

Professor / Main Supervisor

Abstract

A vessel is a complex and integrated system, which is composed by several subsystems and parts which can have common interfaces and interact in a non-linear way. With the continuous development of ship design techniques, a continuous increase in the amount of information generated and handled by the design process can be noted. Having an efficient way of handling all this information during the vessel design process is essential to produce a relevant design in the competitive ship market.

This work proposes an investigation about how the conceptual design phase of a vessel can be approached and improved using concepts from Knowledge-Based Design, System-Based Ship Design and Open Source Software. These theories are combined to put together an Open Source Conceptual Ship Design Tools Library, which provides a set of design tools to be applied in the beginning of a vessel design process.

The development of the tools library is approached in details in this work. It is structured using the knowledge-based design prototype concept. The vessel is subdivided using system-based ship design theory to make the design task less cumbersome and easy to be handled by the tools library. JavaScript is used as an open standard to implement the tools library in a web-based platform. The way JavaScript should be used in order to better deal with the vessel subdivision is also discussed, putting some light in the object oriented programming methodology.

Once the tools library is implemented, a case study is conducted to evaluate how well and appropriate its performance is in a real world problem. This study is done in cooperation with Ulstein International, which provided precious information and discussions about their design process.

Acknowledgements

Firstly and foremost, I would like to thank my beloved wife Laís, for being always by my side during my journey throughout this work. For supporting me when I was too exhausted to stand by myself, for taking care of me and giving me all the love I can take. I would also like to thank her for helping me reviewing this report’s and correcting several flaws, especially the ones related to the English language.

I would like to thank my supervisor Henrique Murilo Gaspar for directing me towards relevant literature, for valuable discussions throughout the semester and for providing traveling grants through the EMIS project. Also would like to thank him for all the support he gave me since I first arrived in Ålesund, 2 years ago.

I would also like to thank the team at Ulstein International for their hospitality during the period I spent in Ulsteinvik. I want to thank specially Ali Ebrahimi, for his availability and the valuable discussions about conceptual ship design and Per Olaf Brett for providing important feedback about the development of the work.

I would like to give an especial thank for Gabriel Araujo von Winckler, for spending some of his time to read my work and give precious feedback.

Finally, I would like to thank my family, especially my mom Maria, for giving me the conditions to achieve all my goals, by providing me a good education and for being always ready to lend a helping hand, whenever I need it.

I dedicate this work to my late father Antonio, who always believed in my potential. He was the proudest dad in the world and although was not able to see my main achievements, will always be with me in every new challenge I overcome.

Ålesund, June 3 2016

Thiago Gabriel Monteiro

Thiago Gabriel Monteiro

Table of Contents

List of figures

List of tables

Abbreviations

CBP	Class-Based Programming
CSD	Conceptual Ship Design
DOE	Design of Experiments
KBD	Knowledge-Based Design
OOP	Object-Oriented Programming
OS	Open Standard
OSS	Open Source Software
PBD	Point-Based Design
SBD	Set-Based Design
SyBSD	System-Based Ship Design

1 Introduction

1.1 Background

The ship design task is complex by nature. A vessel is a complex and integrated system, which in composed by several subsystems and parts which can have common interfaces and interact in a non-linear way. (Erikstad, 1996) proposed seven characteristics to describe a vessel system: it is a self-contained structure operating in the boundary between two fluids; it consists of a multi-dimensional, partially non-monetary performance evaluation; there is high-cost of error if inefficiently designed; there is shallow knowledge structure between form and function; it holds a very traditional industry with preconceived standards; there are strict time and resource constraints; it is predominantly one of a kind and custom engineered.

With the continuous development of ship design techniques, a continuous increase in the amount of information generated and handled by the design process can be noted. Figure 1 presents the idea of this continuous growth through the last sixty years. It does not present a complete list of all improvements that happened in this period, but shows some reference example about each significant type of growth

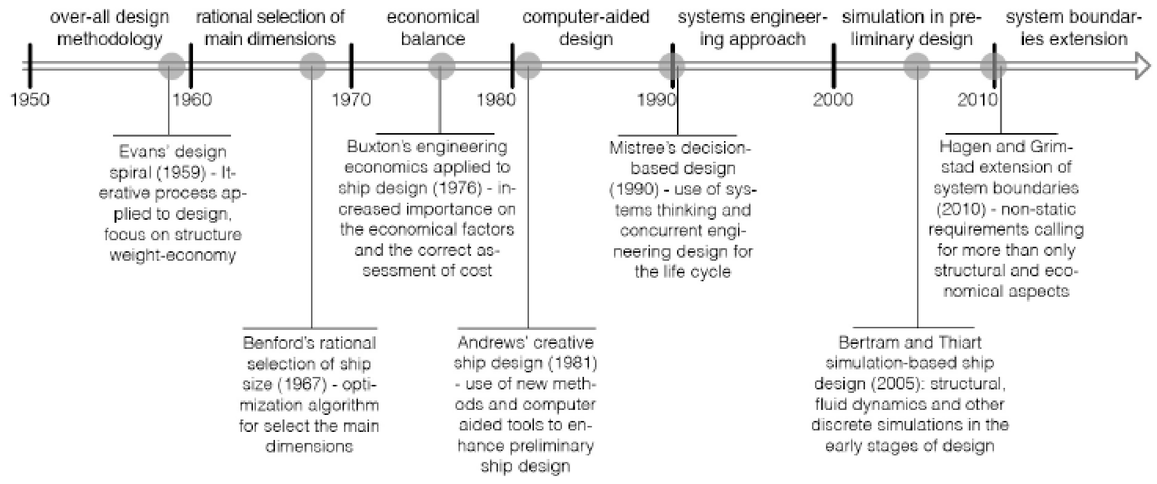

over-all design methodology
rational selection of main dimensions
economical balance
computer-aided design
systems engineering approach
simulation in preliminary design
system boundaries extension
1950
1960
1970
1980
1990
2000
2010
Evans' design spiral (1959) - Iterative process applied to design, focus on structure weight-economy
Buxton's engineering economics applied to ship design (1976) - increased importance on the economical factors and the correct assessment of cost
Mistree's decision-based design (1990) - use of systems thinking and concurrent engineering design for the life cycle
Hagen and Grimstad extension of system boundaries (2010) - non-static requirements calling for more than only structural and economical aspects
Benford's rational selection of ship size (1967) - optimization algorithm for select the main dimensions
Andrews' creative ship design (1981) - use of new methods and computer aided tools to enhance preliminary ship design
Bertram and Thiart simulation-based ship design (2005): structural, fluid dynamics and other discrete simulations in the early stages of design

Figure 1 - Simplified timeline for information growth in ship design through decades. (Gaspar et al., 2012a)

During the vessels' life-cycle (Figure 2), all these information need to be handle and exchanged by several people, being them part of the same company or members of different organizations involved with the vessels' life-cycle. In order to make the design process as efficient as possible, it is important to have a common standard to identify vessel's systems and components to be used for all the parts involved in the life-cycle.

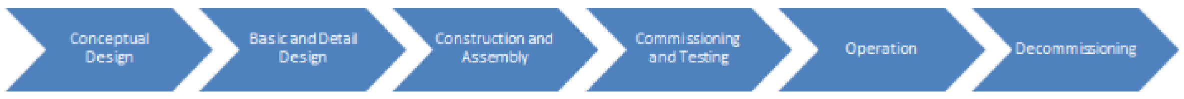


Figure 2 – Simplified vessel life-cycle (Monteiro et al., 2015)

One example of this kind of standard is the SFI Group System, which is a coding system to unequivocally identify any vessel component at any point of the vessel's life-cycle. This kind of coding system is useful for solving communication, cost and control issues. SFI is the most used and well-known coding system used around the world for vessel design.

Besides the standard for components and systems identification, there is also the need for a common platform for handling different types of analyses results involved in the vessel design process. Ship designers already have at their disposal advance techniques to evaluate the vessels' performance according to different merit figures, such as structural resistance, hydrodynamic forces, seakeeping capabilities, stability and so on. What they don't have is a common platform to perform all these kinds of analyses, handle the results and combine the data, which could definitely improve several aspects of the design process by means of accelerating the design tasks, reducing rework, saving time and money and etc.

The concept of open libraries was developed with the advent of internet, since it became easy for people to learn, develop and share knowledge about programming. Web based open source software can be an interesting alternative for fulfilling the need for this common platform, since several applications can be developed and integrated in a ship design library that can address the ship design problem from different angles. The applicability of web oriented open source technology needs to be investigated to check the extent of its collaboration to the ship design field.

1.2 Problem Statement and Research Questions

The main objective of this study is to develop an open source conceptual ship design tools library which will address vessels' conceptual design problems using knowledge-based design and system based design approaches. This object statement can be a little cumbersome at first glance, but it can be easily explained.

At first, the concept of a tools library needs to be approached. The term library here refers to a collection of objects, which works together to accomplish the required design task. These objects are the tools which the library contains and organizes. Think in these objects as structures to store and handle knowledge or perform functions, calculation or operation.

Secondly, the concept of open source software is relevant here, since it provides a powerful and yet free platform where the tools library can be implemented. Being developed in an open platform allows the library to not be bounded to any proprietary software, which add freedom and flexibility while developing and integrating different tools.

Finally, the conceptual design phase is the beginning of the design process, but still holds a big potential to affects the final design. Decisions taken here affect largely the cost structure of the product while being cheap to be done. This characteristic makes this phase an obvious target for improvement search, which is a problem that the tools library will try to address.

In this light, the main goal of this study is to develop, using a not proprietary platform, a collection of useful and integrated tools to be applied in the conceptual design phase of vessels. This problem statement leads to important questions which need to be investigated. They are:

- **Question 1**: How can the vessel be efficiently subdivided in order to allow better design control and knowledge handle in the initial stages of the design process?
- **Question 2:** Which aspects of open source technology can improve the conceptual ship design process?
- **Question 3:** How to develop a tools library which has a flexible enough structure to contemplate the design of any kind of vessel but at the same time is capable of providing results that are good enough to compose a solid ground for the continuation of the design process?

Question 1 points to the fact that a vessel is a complex system. It is composed by several subsystems, which can interact with each other and have common interfaces. A powerful standard to subdivide the vessel in order to make the design process of such a complex system more efficient is important to improve the current design process. It is also important to ensure that this standard can be used in the next design stages, in order to reduce the amount of rework as the project progresses through its lifecycle.

Question 2 wants to investigate how the advantages (and even the disadvantages) of the open source technology can impact in the conceptual design phase of a vessel. Open source technology is already well placed in the software development community and had provided improvements to several technological segments due to, mainly, its work philosophy and engaged community.

Question 3 puts some light in the fact that every kind of vessel is a specialized system, designed to fulfil specific needs. Even between vessels of the same kind, special requirements can make the designs quite different. All these differences can make it difficult to define and implement one single structure to handle any kind of vessel design problem in an efficient way.

1.3 Scope

In order to provide an answer for the research questions the problem will be approach by connecting three distinct knowledge areas, using concepts and theories from: knowledge-based system design, ship conceptual design and open source technology.

The knowledge-based design theory is a design methodology used for designing complex systems. The prototype, which is one of the main knowledge-based design aspects, is a conceptual representation of a design, which provides a framework for storing, handling and processing design knowledge. This concept will be used to develop a vessel prototype, providing the base for the development of the conceptual vessel design open source tools library.

The conceptual ship design phase lies in the very beginning of the vessel design process. It can be approached in several different ways, being the most well-known and applied one (and yet maybe the most outdated) the Evan's Design Spiral. This classical ship design approach can work as a baseline for newer design approaches, which is the case of the System-Based Ship Design, that learns from the pros and cons of Evan's Spiral and offers a new methodology for handling the initial stages of the ship design task.

The open source technology is the chosen platform to implement the conceptual design open source tools library and JavaScript is the selected open standard, since it is a web-based language and can be used to extend the tools library functionalities and capabilities. The use of open source technology brings freedom to the tools library development, which would not be possible to reach using any proprietary software.

Some of individuals have already approached up to two of these topics together (the FREE!ship project (HydroNShip, 2016), the System-Based Design theory (Levander, 1991) and a Knowledge-Based Structural Design System approach (Lee et al., 1996) are some examples), but, to the best of my knowledge, nobody has done it before considering the three topics simultaneously, which comprehends this work's scope. (Figure 3).

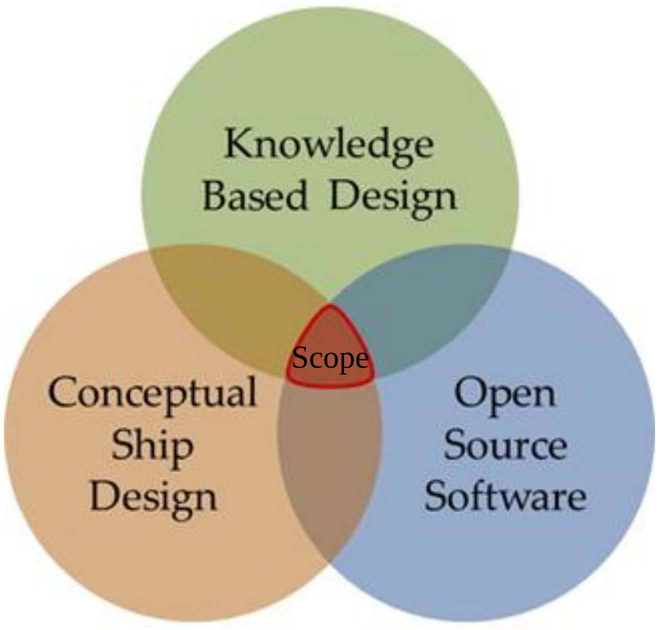


Figure 3 - Project's Scope: Intersection area of the three main topics.

1.4 Structure of the Thesis

From hereafter, the thesis is structured as follows:

- **Theory**

  Chapter 2 presents the main concept regarding the Knowledge-Based Design theory and how knowledge can be used to efficiently design complex systems. Chapter 3 approaches the conceptual design phase of a vessel, describing its main methodologies and making a comparison between them. Chapter 4 discusses the Open Source Software concept and its implication, besides introducing the JavaScript language.

- **Methodology**

  Chapter 5 develops and presents the methodology that will be used for the analysis. In this chapter it will be described how the concepts mentioned in the previews chapters are put together to compose the conceptual ship design open source tools library in JavaScript. A small case study is presented at the end of the chapter, to exemplify the tools library application.

- **Case study**

Chapter 6 presents a more complex case study, applying the developed methodology using the requirements for a real vessel, in order to verify how this work’s approach can deal with real design problems.

- **Results and discussion**

  Finally, Chapter 7 concludes the study and evaluates how it can be extended in the next steps of the work.

2 Knowledge-Based Design

In this chapter, the design process will be discussed, giving special attention to the knowledge-based design theory, which will be used to develop a vessel prototype. The prototype will be a conceptual schema that provides a framework for storing, handling and processing design knowledge. The prototype provides the base for the development of the conceptual vessel design open source tools library.

2.1 What is Design?

A common misunderstanding about design is confusing it with the final artifact. The design is, in reality, a description of the final artifact, from where it is possible to make predictions about the artifact's performance. This statement points that the design is not only about coming up with ideas and putting them together, but it is also related to predicting and understanding how these ideas will behave together.

A designer is a change agent in the society, whose main objective is to solve an existing problem, improving the world around us and trying to make it fit us better. The designer can address different issues, such as functionality, meaning, expression and aesthetics, covering activities with varying degrees of sophistication. The designer can focus on an adaptation of a well-established prototype or a creation of something entirely new. (Coyne et al., 1989)

Design is a purposeful and goal-oriented activity, which is constrained and involves decision-making, exploration and learning. (Gero, 1990) Since the design is purposeful and goal-oriented, a designer is always trying to achieve a desired state or performance. The design is also constrained by both the world and the context where it is going to be applied. The exploration process consists in evaluating both goal and decision variables (which guide the design activity) in order to judge what variables are pertinent or not to represent the model. The decision-making process involves choosing what values should be assumed for the selected variables, based on performance indicators. The learning process is carried out in parallel with the exploration, and should be used for increasing the knowledge about the design while the design task progresses.

Designing is a complex, extensive and ill-structured task, which can be approached in different ways that can lead to different results. The design activity can follow a "top-down" (goal to artifact) or a "bottom-up" (artifact to goal) approach. It can have different understandings for different people and it could be difficult for those people to agree about goals and methods, which can reinforce the ill-structured nature of the design activity. (Coyne et al., 1989) This ill-structured nature can also be extended to goals and requirements of the design activity, since, usually, the number of free design variables will be greater than the number of equality constraints. Thus, the design solution is not uniquely determined by the set of requirements. (Nowacki, 2010)

If the ill-structured characteristics of the design make the tasks and/or goals not clearly defined, the designer can solve this by applying decomposition techniques until suitable and understandable subtasks and/or goals are achieved. Even decomposing the design problem, some issues cannot be considered until some attention is given to the overall solution, which may require an iterative process.

2.1.1 Design as a Process

The design process can be seen as a problem solving process of searching through a state space, where the states represent the design solutions. This state space may be large and complex. The search will involve making decisions based on goal and decision variables, which can be constrained by the world or context where the design is applied or produced.

The design process can be schematized as shown in Figure 4. From a set of functions, which represents a non-structured description of the artifact, the design process provides a structured artifact, which can be produced. From this set of functions, it is possible to obtain a set of expected behavior for the artifact. From the structure description, it is possible to obtain a set of actual behaviors for the artifact. From the comparison between the expected and actual behaviors the quality and rightness of the design can be evaluated.

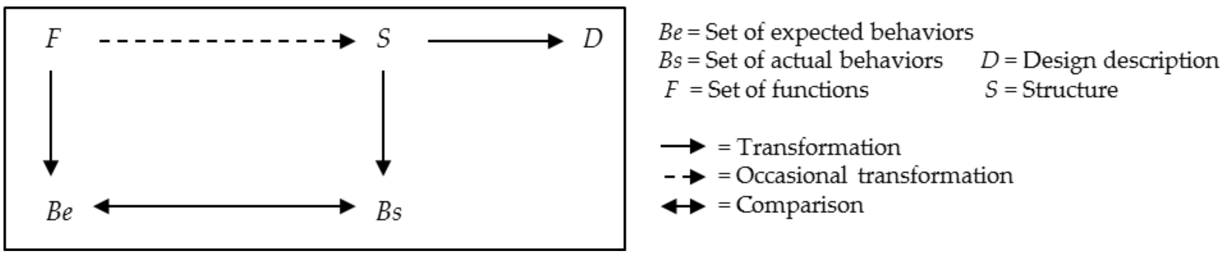


Figure 4 - Design as a process. (Gero, 1990)

According to (Gero and Rosenman, 1990), the design is not a logical process, from where it is possible to deduce a structure description from a set of functional requirements. Unlike science that aims to generalize results, the design task aims to specify them. The deduction can be used to, from a given design, to predict its behavior and performance unequivocally, but not in the opposite direction. The design process can be better classified as an abductive process, since it starts from the required results (set of functions) and uses the available knowledge to arrive in a description of an artifact that satisfies the initial needs (this description is not necessarily unique). More details about different ways of reasoning about a problem can be found on Section **Error! Reference source not found.**.

2.1.2 Routine, Innovative and Creative Designs

A design can be framed in three different categories according to how conventional it is: a routine design, an innovative design and a creative design. (Gero and Rosenman, 1990)

The more conventional category is the routine design, in which the design approach, the problem's structure variables, expected behavior, functions and common problems are already known. On this type of design, the design problem is already well structured and formulated, making the instantiating of variables and constraints values the real design problem.

The innovative design category is less common than the routine design but it is absolutely essential. The innovative designs emerge when there are no conventional solutions that meet the design requirements. This kind of situation can happen due to a new constraint in the problem, the need for a different behavior, the use of a new structural material and so on. So, in order to fulfill this need, the routine design space needs to be modified or extended, providing a broader design space.

The least common category of design is the creative design. Although all design is creative since the design process provides a description of something new, that did not exist before, the creative design category needs a change of paradigms to arise. This change of paradigm will demand a new design structure, which should be feed with knowledge that, maybe, doesn't even exist yet. In creative design, the expansion of the routine design space is not sufficient, it is necessary to create a new design space.

Figure 5 presents a representation of the design space of these three kinds of design.

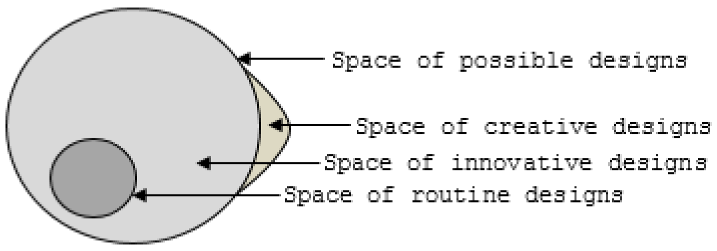

Space of possible designs
Space of creative designs
Space of innovative designs
Space of routine designs

Figure 5 – Routine, Innovative and Creative design spaces representation. (Gero, 1990)

2.2 Knowledge-Based Design

The knowledge-based design (KBD) process can be seen as a problem solving process of searching through a state space defined by the syntactic knowledge (design variables) and the interpretative knowledge (design performances), where the states represent the design solutions. The searching process should be done using reasoning based on goal and decision variables, which can be constrained by the world or context where the design is applied or produced. (Coyne et al., 1989)

The KBD theory can be used to address any design problem by doing the mapping between the syntactic and the interpretative spaces (Figure 6).

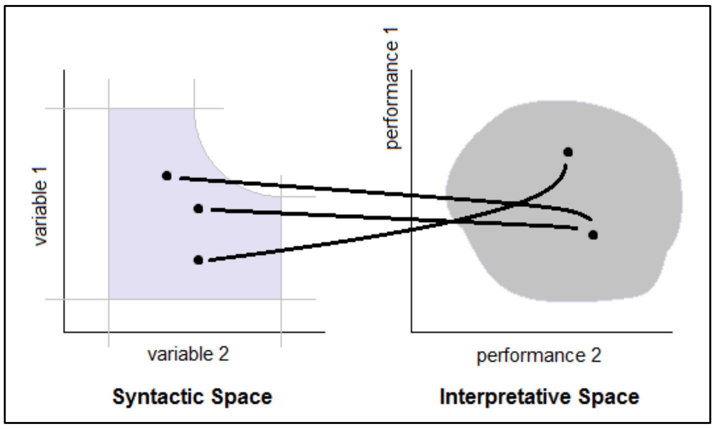


Figure 6 - Example of mapping between syntactic and interpretative spaces. (Coyne et al., 1989)

2.3 Knowledge-Based Design Process

(Gero and Rosenman, 1990) structured a process workflow for applying KBD theory in a design problem. This workflow is composed by five steps, which are:

1 - Understand a problem and formulate its functions and behaviors.

In this step, the designer needs to investigate the design problem in order to obtain an initial understanding about the variables, constraints, functions and goals involved. Some design challenges cannot be foreseen by the designer, and will appear as the design process progresses. These challenges need to be approached and fixed as soon as possible to allow the design to converge to a satisfying solution.

2 - Arrive at a satisfactory structure vocabulary from which to select.

It is important for the design to be familiar with the vocabulary of the approached design problem. The vocabulary represents the syntactic knowledge and is specific for each design problem.

3 - Select satisfactory structure vocabulary elements.

Between the applicable vocabulary elements, it is important to approach only the ones really relevant to the design problem, in order to simplify the needed structure but without under thinking the problem. This selection should be conducted based on the representation of the expected behaviors of the designed artifact.

4 - Configure structure vocabulary elements.

The designer should apply knowledge in order to configure and relate vocabulary item. The knowledge can come from previous experience or be obtained as the design progresses and new information and constraints are found.

5 - Select among competing solutions.

Once possible design solutions are configured, it is important to correctly evaluate their performance to verify how well they meet the design requirements in order to find the best possible solution.

2.4 Schema and Design Prototype

One of the fundamental concepts in KBD is the Schema. Schemas consist of generalizations of knowledge obtained from a set of alike design cases. They form a class from which individuals can be inferred and must at least be able to incorporate function, structure, behavior, and design description and be accessed by elements within these components. (Gero, 1990) The Schemas should also be capable of addressing routine, innovative and creative design problems.

The schemas can be divided in three basic types: archetypes, stereotypes and prototypes. Archetypes are the first and often singular examples of their type. Stereotypes are copies without change of an original design, which is ideal for a mass production system. Prototypes are the first design over which others are modeled. (Gero, 1990) Usually prototypes are used for applying KBD theory, since it allows the derivation of different designs from an initial schema.

Given that designers design from experience, it is needed a system of storing this experience in a coherent structure. (Gero and Rosenman, 1990) The prototype should be understood as a conceptual schema for knowledge or a clear way of representing the design and its properties.

The prototype represents a class of elements from which instances of elements can be derived. A prototype brings together the three types of variables groups (function, structure, behavior) that define the designed artifact and the relation between them, which includes process for selecting and obtaining values for variables.

Instance can be derived by inheriting properties, functions and variables from a generic prototype. It is possible to derivate new instances from prototype that have already been derived from other prototype, which makes it possible to develop a complex hierarchy in the design process. (Gero and Rosenman, 1990)

A diagram of a prototype schema can be seen in Figure 7. The function properties include the intended function in the form of goals and requirements, and the expected behaviors as attributes and variables. The structure properties include the vocabulary, the prototype description, its configuration and the actual behaviors as attributes and variables of the prototype.

Knowledge plays a big role into the prototype schema. Relational knowledge is required for every mapping from a property to another. Besides the relational knowledge, the prototype also stores qualitative knowledge, computational knowledge and context knowledge. The qualitative knowledge complements the relational knowledge and provides information on the effects of changing structure variables values on behavior and function properties. The computational knowledge is the quantitative counterpart of qualitative knowledge and specifies symbolic or mathematical relationships among the properties variables. The context knowledge identifies the external variables for a design situation, which should come from the context where the design is inserted. The qualitative and computational knowledges are subjected to constraints, which on function properties appear as expected behaviors and on structure properties reduce the set of possibilities.

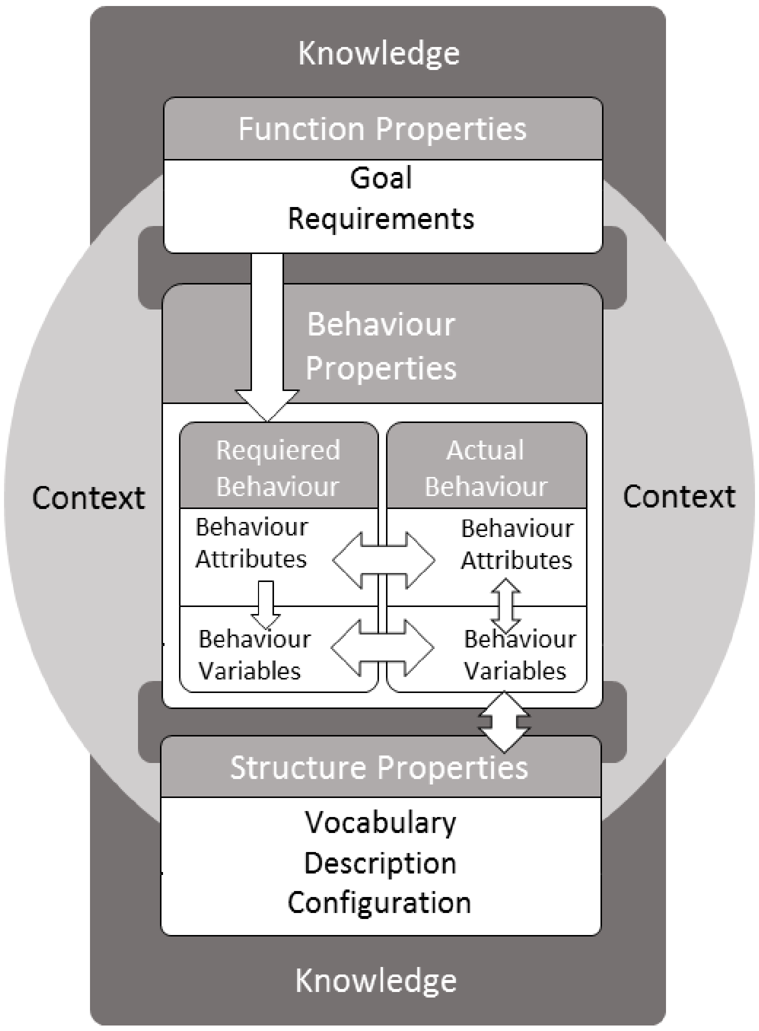


Figure 7 - Prototype schema diagram. (Gero and Rosenman, 1990)

2.5 Difficulties Applying Knowledge-Based Design

According to (Rosenman, et al., 1989), in order to apply the knowledge-based system design theory, three main problems should be addressed by the designer: the prototype definition, the syntactic spaced definition and the interpretative space definition.

The prototype should be understood as a Schema, which is used as a conceptual structure for organizing knowledge. This schema provides a framework for mapping the characteristics and performances of a design, besides carrying all the relevant information about it. The prototype definition is a complex problem, especially for a system as complex as a ship.

This high level of complexity makes it even more important that the mapping process between the characteristics and performances be done using correct reasoning. The prototype should be prepared to deal with the ship design problem complexity and constraints when applying the performance evaluation tools, in order to provide the best possible results with the minimum interference from the user.

The definition of the syntactic space is a crucial step of the design process, since an individual outside the boundaries of this space cannot be evaluated by the prototype. An inaccurate definition of the syntactic space could lead to the no consideration of a potentially good individual. Again, due to the complexity of a vessel design problem, a good definition of the syntactic space is difficult. This space can have dozens of dimensions (depending on how complex the system representation will be) and the definition and implementation of the set of constraints that limits each dimension is a demanding task. This task can be handled by applying design of experiments (DOE) technics (Fisher, 1971) in order to provide an appropriated initial population for the design process or by the use of knowledge from experts.

The definition of the interpretative space covers all the performance evaluation tools that are present on the prototype. In the case of this project. these tools aim the assessment of several aspects of the vessel’s lifecycle performance while providing a base for the decision making process on an early stage of design. The definition, implementation and integration of each tool in the vessel prototype represent part of the complexity of this phase.

3 Conceptual Ship Design Methodologies

In this chapter, the more common conceptual ship design methodologies are going to be approached. The main aspects of each methodology will be presented and the methodology for conducting the studies in this thesis will be chosen.

3.1 Introduction

As stated by (Gaspar et al., 2012b), a ship is a complex and specific product. It is composed by several subsystems, which should work together in order to ensure the correct operation of the vessel, making it capable of performing its required tasks as well and efficiently as possible.

Historically, the ship design process was conducted using Heuristic methodology (Papanikolaou, 2010), which means that the design process was learned by means of try and error and was highly dependent on past experiences. With the need for achieving better results during the design process, analytical models started to be used to relate the vessel's required tasks to the design parameters (or requirements). (Cho and Žanić, 2006) These design parameters are the result of an extensive discussion involving the vessel's stakeholders and experienced decisions makers. In this discussion, end-users and people involved in the vessel design and construction try to make their desires and expectations count, but also considering the tradeoffs they are willing to permit. (Papanikolaou, 2010)

The ship design task should address all the vessel's life-cycle, and the latter should consider at least the conceptual design phase, the contract and details design phase, the construction and fabrication phase, operation and economic life and scrapping and decommissioning. Since the different phases of the vessel's life-cycle have contrasting objectives, it can be taken that the final design is a result of a Holistic process, where the whole design should be considered as not being only the sum of parts, but should take into account the relation between these parts as well. This fact makes the ship design problem so complex that even in its simplest phase (namely the conceptual design phase) the design cannot be reduced to its fundamental part, being yet present the need for Holistic approach. (Papanikolaou, 2010)

(Chou, 2004) suggests that the conceptual design process includes four main phases: the necessities identification, the requirements definition, the selection of design criteria and the development of a solution framework. This process can be addressed, mainly, by using two different approaches: the point-based design and the set-based design.

3.2 Point-Based Design

The most traditional approach used for initial ship design process is the classical design spiral (Figure 8) (Evans, 1959). On the spiral model, the different vessel design features such as displacement, volume, weight, stability, seakeeping, should be approached in sequence, following the design spiral. On each turn around the spiral, the design features should be considered with increasing detail. The model is iterated until a reasonable solution that satisfies the design goals is found. Since the design process seeks for a single solution for the problem, this approach is called point-based design (PBD). (Parsons, 2003)

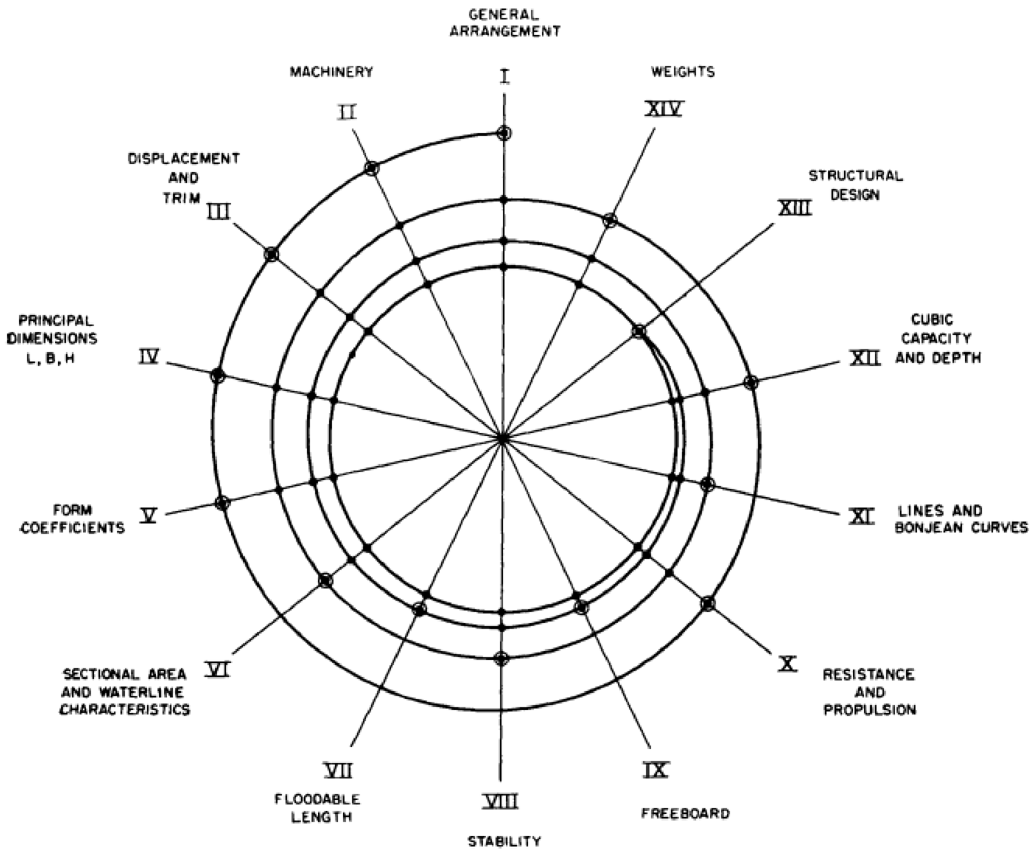


Figure 8 – Classic design spiral. (Evans, 1959).

As summarized by (Liker et al., 1996), the PBD process can be divided in five main steps. The process begins with the problem definition, based on the design requirements. Several alternative design concepts which satisfy the stablished problem are then generated. These alternatives are analyzed in order to select the most promising one. The selected alternative is evaluated and modified until a solution that matches all the design criteria is found. If, by any reason, the selected solution proves itself unfeasible or fails to match one or more of the desired goals, the designers should try to select another alternative or, in the worst case, rethink the problem definition and restart the process from the beginning.

According to (Parsons, 2003), the PBD approach has a big disadvantage related to finding optimum solutions. Although it can produce a feasible design that meets all the stakeholders' requirements, there is no way to ensure that this solution is a global optimum (regarding the vessel's key performance indicators) due to the lack of comparisons with other feasible solutions.

Another disadvantage of this approach is related to design convergence. Since the iterative process around the design spiral is an extensive and time consuming task, it is usually limited by the project's budget and schedule. The project will be considered complete when these resources become scarce, despite of its convergence to an optimum or suboptimum global solution (since it matches, at least, the basic design requirements). (Singer et al., 2009)

3.3 Set-Based Design

The main feature of the set-based design (SBD) methodology is the fact that it defines broad sets for the parameters' design, in order to allow a concurrent design to begin, and keeps these sets open, so that the design teams can see the difference in performance and cost among the different solutions

Nowadays, a different approach, taken from the automotive industry, is being used in the ship conceptual design. (Brinati et al., 2007) From the study of Toyota's production system, famous for its world-class design and production of automobiles, it led to the conceptualization of Lean Manufacturing (Womack et al., 1991). The Toyota design processes produce excellent designs in a shorter time than other automobile manufacturers.

The main features of Toyota's design process include (Singer et al., 2009):

- concurrent design based on a broad set of design parameters;
- more tradeoff information is obtained keeping the set unconstrained for longer time;
- the set is gradually narrowed instead of abruptly constrained, until a more global optimum solution is revealed and refined.
- the design fidelity increases when the set narrows.

This design process was characterized by Alan Ward as SBD. It works in a different way from the traditional PBD, where the system's interfaces are specified early in the process, so that the design can go on. Usually these definitions and theirs correspondent constrains are imposed long before the needed tradeoff information is available, resulting in sub-optimal overall designs. (Parsons et al., 1999)

The Toyota design process approaches the design process in a different manner than the one expected to be the most efficient, and brought with it two paradoxes. The first paradox is related to Toyota's Lean Manufacturing System and Just-In-Time Inventory. Ideally, the economy of scale should be the best path to produce good designs with lower price. But while companies minimize price by maximizing machine speed and capacity, they also neglect the impact of space, transportation, and inventory. Toyota, on the other hand, operated with little to no inventory and manufactured vehicles at a lower cost and with better quality. (Singer, 2008)

The second paradox (Ward et al., 1995) is related to how Toyota is capable of having a shorter time to market then other companies even though it delays its decision making process. The reasons for that are related to with cost structure, design team influence and knowledge about the design process, as it will be discussed in the next section.

3.3.1 The Effects of Early Decision Making

The PBD methodology values the early decision making. This approach can harm the outcome of the design process, mainly due to three factors: evolution of the product's cost, management's ability to affect these costs, and evolution of the designers' knowledge about a design problem. The negative effect of these factors in the outcome of the design process can be reduced by means of delaying the decision making as far as possible. (Bernstein, 1998)

The first factor to be approached is the product's cost. The design team is responsible for defining everything related to the product's cost. The chosen design, the way it will be produced, how it should be transported and required selling price are all important decisions defining the structure of a product's cost. This decision process is a tricky task, since the choices made in the very beginning of the design process (with the least data) have the most impact in the product's costs structure. (Ward et al., 1995)

At the end of the conceptual design phase, between 60% and 80% of the product's total life-cycle cost is determined. (Dierolf and Richter, 1989) Although the impact of the conceptual design phase decisions in the final product are enormous, the amount of resources spent in this stage are minimal. For a vessel design, the time and money allocated to the conceptual design phase are about 2% of that of the design, detailing and construction phases all together. (Levander, 1991)

(Gaspar, 2013) also considered in this evaluation the effect of the changes in the design over the vessel's life-cycle (Figure 9). The cost of removing or correcting a design flaw or making any change in the vessel's design in the construction phase at the ship yard can be as big as 1000 times higher than if the same change was made during the conceptual design phase.

With these considerations, it is possible to notice the importance of the decisions taken in the early product development and how they can affect the product's cost structure. Also, the impact in lowering the cost structure of decisions made in the later stages of development is very small. (Anderson, 1997)

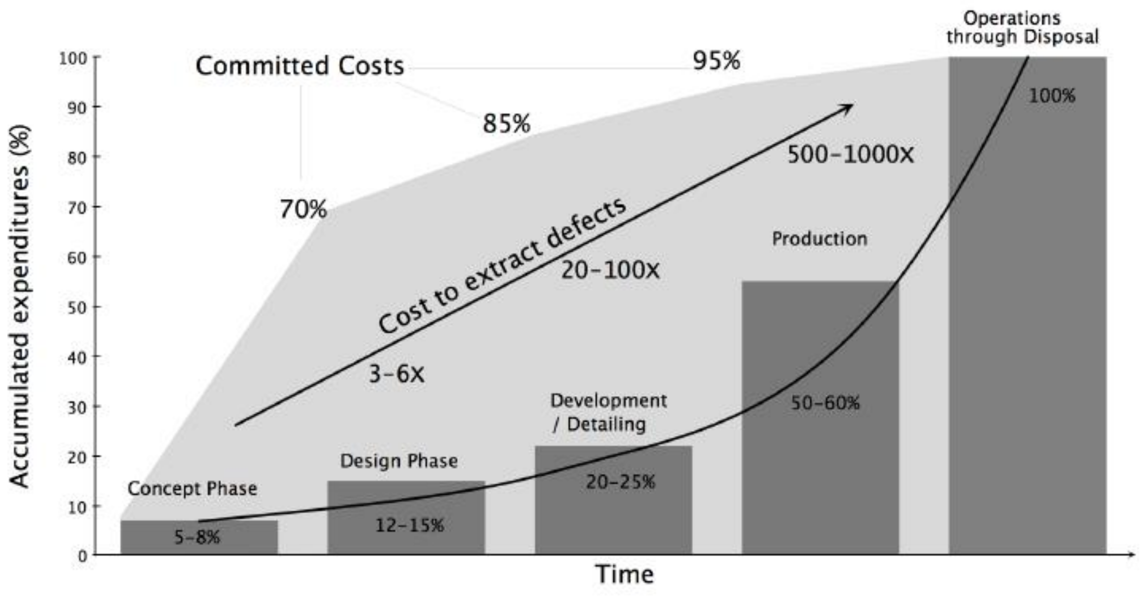


Figure 9 – Costs evolution through the main design phases. (Gaspar, 2013)

Other factor influencing the product's cost structure is the management's ability to influence a product's design. During the conceptual design phase, the design team has a great capacity to influence the product's design. As the design progresses, this capacity is greatly diminished.

This characteristic is due to the fact that the design team constraints the available options for design solutions with each decision made. (Krishnan et al., 1991) So, as the design progresses, more decisions are made and more constraints are inserted in the design formulation, making the management's power influence the product's costs structure to decline, while these costs increase.

The conceptual design phase is also characterized by the lack of knowledge from the design team. At the beginning of the product's development, the design team is not really aware about the problem's variables, constraints and user's needs. As the problem is worked, the knowledge starts to consolidate and the design team becomes more capable of making better decisions.

As already discussed, these initial decisions have the most impact in the product’s cost structure, however they are made when the design team has the least knowledge about the product and the process.

(Erikstad, 1996) corroborates with the relation between the design knowledge and the design freedom and how they evolve over the life-cycle of a vessel. At the beginning of the conceptual design, no decisions have yet been made, and the only constraints are the ones related to the top-level mission requirements. All subsequent decisions will constraint the design freedom.

The limited design knowledge can be mainly attributed to the uncertainty in the relation between the form and the function of the vessel. The uncertainty in these mapping functions can be related to the vessel itself or to the vessel’s environment. Especially in the early stages of the design process, the mapping between form and function needs to be modelled, in a large extent, using heuristic and empirical rules.

Figure 10 shows how useful delaying the decision make process can be. Firstly, it can delay the commitment of costs for a moment when more knowledge about the product and the design process is available. Secondly, it can increase the manager’s influence over the late design, since several constraints can only be considered when the decisions regarding them are made, which would give more options for the product’s design. The knowledge is considered unchangeable, since it is already considered that it is obtained as soon as possible. (Bernstein, 1998)

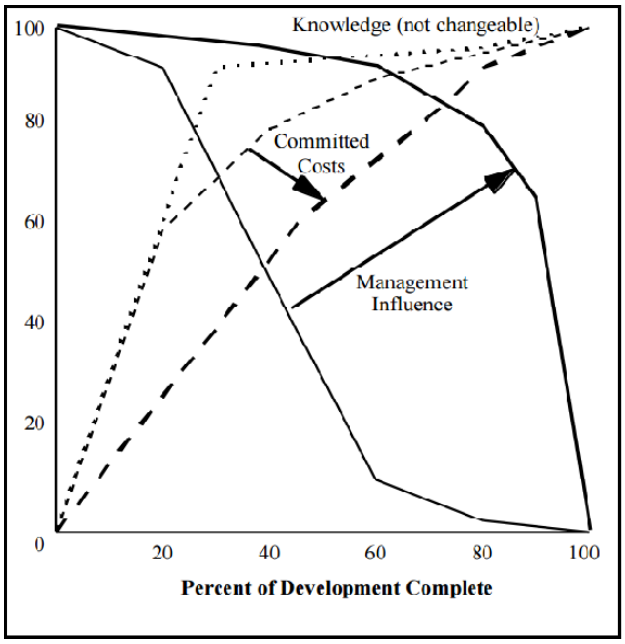


Figure 10 - Perks of delaying the decision making process. (Bernstein, 1998)

(Kalyanaram and Krishnan, 1997) pointed additional benefits of delaying the decision making process:

- better balance between customer needs and technical feasibility;
- keeps the design open to receive the latest technology available;
- more competitive products, both in terms of price and performance;

better track changes in customer desires.

3.4 System-Based Ship Design

According to (Hubka and Eder, 1988), who described the bases for technical systems, the system thinking has as its main features (Levander, 2012):

- Delivers the relationships that are valid for all products;
- Presents an opportunity to treat problems as a whole;
- Is a necessary pre-condition for a successful design and engineering effort;
- Provides a framework for the design task and formalizes many logical operations;
- Supports those human operations, that are not strictly logical, as intuition and creativity.

Kai Levander, who believed this system approach could help the development of innovative vessel designs, applied the idea of system thinking to ship design, developing a design methodology called System-Based Ship Design (SyBSD). This methodology works as a framework for the vessel design task. This framework is structured over the idea of dividing the vessel in different systems, based on their functions, which work together to accomplish the desired ship mission. (Levander, 2012)

Differently from a top-down design approach where the design starts from the vessel and continues to detail the vessel's systems, the SyBSD uses a bottom-up approach, going from the vessel's required functions to the composition of the vessel itself. The designs start from the mission specification, which defines task, capacities and expected performance by the vessel's stakeholders. This approach straightens the beginning of the design spiral, delaying the beginning of the decision process and reduces the number of iterations needed to find a feasible solution. A comparison between the SyBSD spiral and the conventional one can be seen in Figure 11.

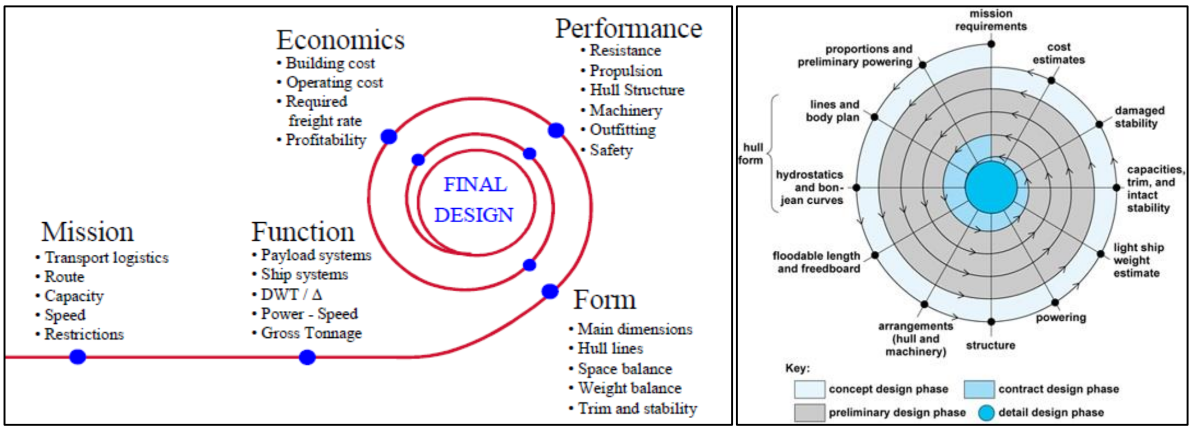


Figure 11 – Comparison Between System-Based Design Spiral - Left (Erikstad and Levander, 2012) and Conventional Point Based Spiral - Right (Gale and Slutsky, 2014)

According to (Erikstad and Levander, 2012), the SyBSD process can be summarized as follows:

Customer requirements - Mission statement

- Task, capacity, performance demands, range and endurance;
- Rules, regulations and preferences;
- Operating conditions (wind, waves, currents, ice).

Functional requirements - Initial sizing of the ship

- Based on capacity, where the areas and volumes needed for cargo spaces and task related equipment define the size of the vessel;
- Based on weight, where the cargo weight and the weight of task related equipment and of the ship itself define the size of the vessel.

Form - Parametric exploration

- Variation of main dimensions, hull form and layout of spaces on board to satisfy the demands for both capacity and weight.

Engineering synthesis

- Calculating and optimizing ship performance, speed, endurance and safety.

Evaluation of the design

- Calculating building cost and operation’s economics.

A functional breakdown is used in order to divide the vessel in systems. The vessel is split into the categories Ship Systems and Payload Systems. The Ship Systems are all systems related to the safe and correct operation of the vessel, without taking the cargo into consideration. The

Payload Systems are functions and requirements that generate cash flow for the vessel, which can include cargo and cargo related systems but also specific systems for specialized vessels such as offshore support vessels, which can have winch and heavy lift cranes. Due to these special cases, the Payload Systems can also be called Task Related Systems. An example of this division can be seen in Figure 12.

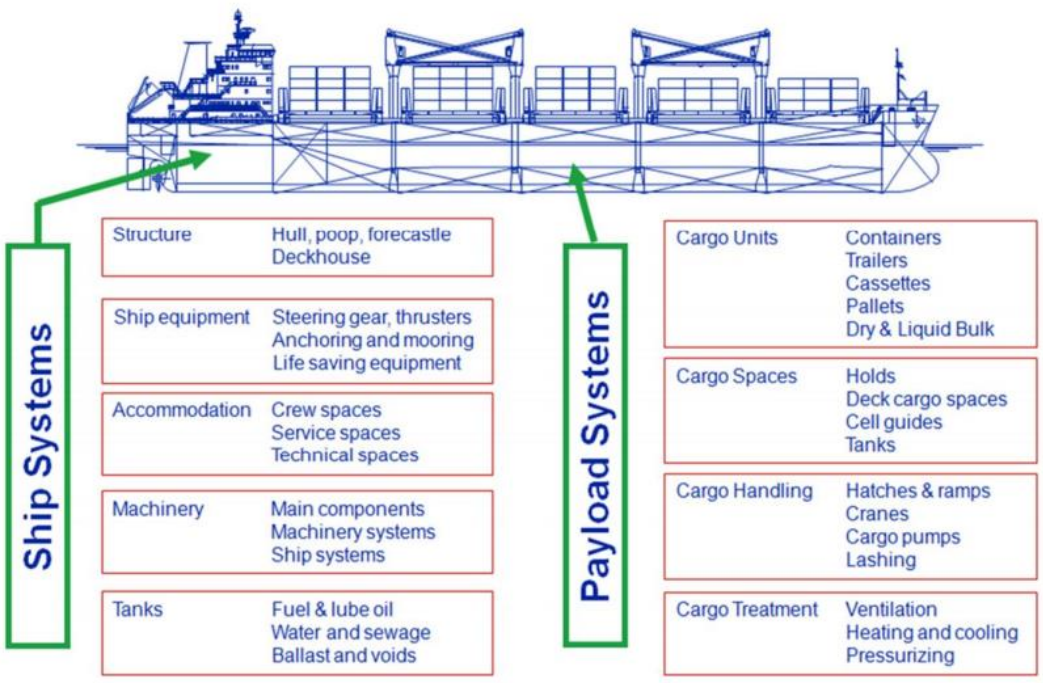


Figure 12 -Payload and ship functions in a cargo vessel. (Levander, 2012)

In order to facilitate data collection, the SyBSD divides the systems based on the structure of the SFI Group System. (Urke, 1976) The SFI group system was developed at the Norwegian Ship Research Institute and is the most widely used classification system for the maritime and offshore industry worldwide. It is an international standard that stablishes a functional breakdown of the ships' technical and economic information and is used by shipping and offshore companies, shipyards, consultancies, software suppliers, authorities and classification societies. It helps the control of operations by tying together all their procedures such as purchasing, accounting, maintenance and technical records.

The SFI group system provides major advantages for shipping and offshore operations in areas such as communication, computerization, cost and quality control, development, education and training and standardization. The system is independent of company organization and methods of ship building, ship operation, maintenance and repair and when it was established, the basic criteria for its design were (Manchinu and McConnell, 1977):

- It must be applicable to all users;

- It must be applicable to all types of ships;
- It must be simple and easy to understand;
- It must be capable of future expansion.

According to (Manchinu and McConnell, 1977), in order to meet the purpose of and the requirements to the system, the ships were divided into functions, which structure and systemize all the ship's different systems and components through a three-digit decimal classification system with ten main groups at the highest level, from which two groups (0 and 9) are reserved for especial use. Each of the main groups (one digit numbers) consist of ten groups (two digit numbers) and each group is further subdivided into ten subgroups (three digit numbers). Hence, the structure of the group system numbers is as follows:

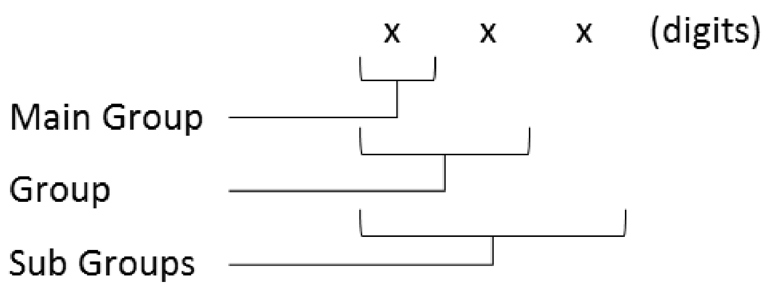


Figure 13 – SFI group system's code structure. (Manchinu and McConnell, 1977)

In the SFI group system, the main groups are divided as shown in Figure 14. Further detailing about the group system subdivision can be seen in APPENDIX D.

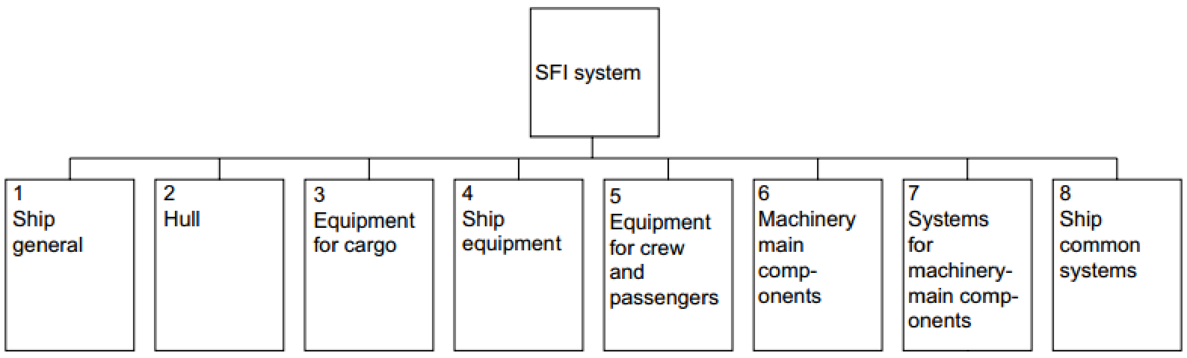

SFI system
1 Ship general
2 Hull
3 Equipment for cargo
4 Ship equipment
5 Equipment for crew and passengers
6 Machinery main comp-onents
7 Systems for machinery-main comp-onents
8 Ship common systems

Figure 14 – SFI main groups structure. (Utne, 2009)

It is also possible to further break this subdivision down using a 6-digit detailed code, as shown in Figure 15.

The correlation between SyBSD and SFI is not completely accurate, since the former does not distinguish between payload and ship systems, which results in some minor differences between the subdivisions of these two structures. Although some differences exist, they can be overcome making some special relations among discordant items. (Erikstad and Levander, 2012) The complete correlation between SyBSD and SFI subdivisions can be seen in APPENDIX E.

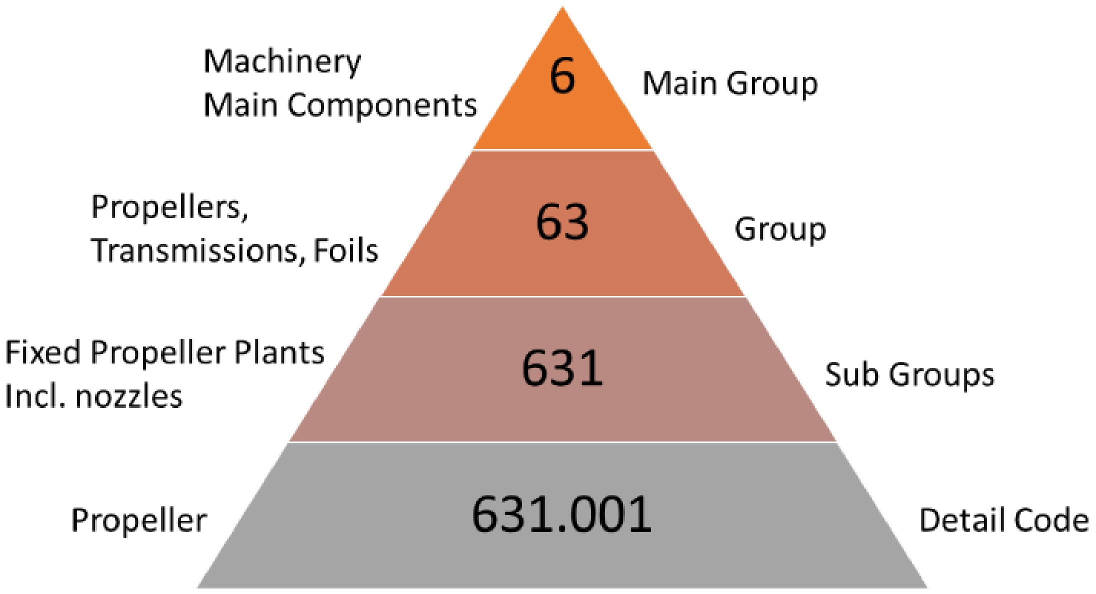


Figure 15 – SFI subdivision examples: Main Group 6.

Since the vessels are usually very generic, they follow a design pattern based on previous experience that resembles a scaling process. This traditional approach easily locks the designer to his first assumption, making him patch and repair the same design, what makes this traditional method not prone to innovation.

The SyBSD, using a bottom-up approach determines the needed area, volume and weights for each vessel's function, and from this figures estimates the displacement, main dimensions and building costs. By doing this evaluation without defining the vessels dimension, the SyBSD method does not lock assumptions in the conceptual phase and supports a more creative process in the start of the project.

The SyBSD method is suitable for the early design decisions, and can be considered as a checklist that reminds the designer of all the factors that affect the design and records his choices. Its use ensures that the design is based on the most fitted basis ship, and reduces the number of iterations in the design spiral later on. (Vestbøstad, 2011) The final product of the SyBSD method application is a complete description of the new ship, which can be used as an advanced start point for the next design phases

As discussed in this section, the SyBSD mix characteristics of PBD and SBD in one methodology. The straightening of the beginning of the design spiral gives the SyBSD an advantage over the PBD towards the SBD. Since the SyBSD methodology is already well developed for specific application in ship design tasks, it will be chosen for developing the conceptual ship design open source tools library.

4 Open Source Software

This chapter is going to present the main aspects of open source software, including why they exist, who makes them possible and how they can succeed. In addition, the open standard concept will be introduced in the form of JavaScript, the computational language that will be used in this work.

4.1 Introduction

In order to protect intellectual property and ensure the possibility of profit, several commercial software companies keep their software's source code in secret. The source code is a sequence of logical instructions written in a certain programming language and used by a computer to execute a given task or achieve a given purpose. In order to protect the source code, companies release the program in a binary version (the code converted in a sequence of zeros and ones), which can be understood by computer but is difficult for users to interpret. (Simon, 1996)

This kind of approach goes against the origins of computer programming, where codes were written and shared by several individuals, with no commercial interest involved. (von Hippel and von Krogh, 2003) To fight this tendency, the now called Open Source Software (OSS) movement was created, aiming to ensure free access to the programs' source code, making it possible for users to understand, change, adapt and improve the original programs' source code.

4.2 Concept

The term "free software" is not related to price or value. It is about freedom. According to (Stallman, 1999) a free software can be called so if an individual user can:

- Run the program for any purpose;
- Have access to the program's source code so it is possible to modify it;
- Redistribute copies of the original program, either gratis or for a fee;
- Distribute modified versions of the program.

The possibility of selling copies of the program is a crucial feature to finance the free and open source software development and communities. In order to ensure all this freedom, the *copyleft* licensing concept was developed, adding distribution terms to the conventional copyright practice, which gives anyone the right to use, modify and redistribute the code, since the distribution terms are kept intact.

The licensing is a complex subject in software development and distribution. There are several licensing standards for both free and proprietary software. The following diagram (Figure 16) by (Kuei, 1999) shows the different categories of software licensing. Some of them limit the access to the closed program (proprietary software), some only limit the access to the source code (public domain software without source code) while others only impose distribution terms (copylefted software).

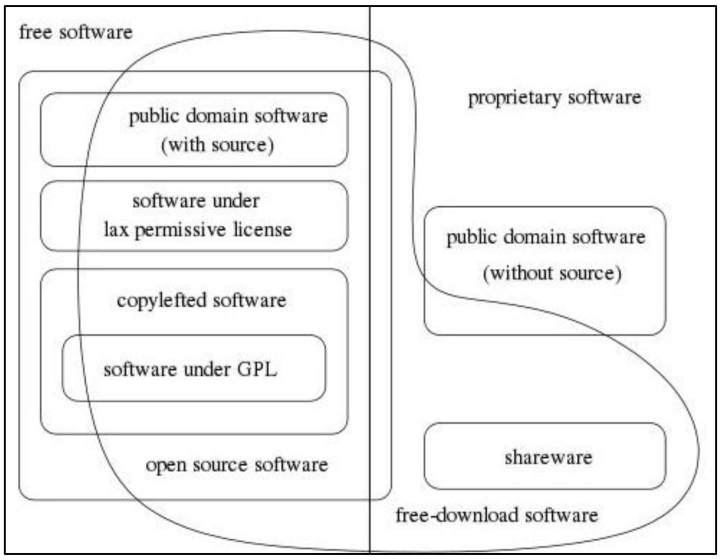


Figure 16 – Overview of types of software licenses. (Kuei, 1999)

The use of software patents instead of copyright in order to protect intellectual property is a big problem for OSS communities. This mechanism can be applied on the compression algorithm used to create file formats such as GIF and MP3. The final users do not need to pay anything to use GIF files, but the developers need to pay a license fee in order to use the compression algorithm in their programs, which is a big obstacle for OSS development. (Bretthauer, 2001)

In order to ensure that a software respects the open source concept, two main characteristics need to be attended: OSS developers are always users and OSS should adhere to the Open Source Definition. The first requirement is self-explanatory and matches the idea that the code developer has a need to develop the software. The second requirement is defined by 10 characteristics that every OSS must have (OSI, 2016). The three most important ones refer back to the free software definition:

- The ability to distribute the software freely;
- The source code’s availability;
- The right to create derived works through modification.

4.3 Open Source Community

The origins of user innovation communities are back before the advent of OSS communities. Innovation communities are not only related to information products such as software code, but are also related to physical products. (von Hippel, 2001) Active OSS projects have a well-defined community. The community organizational structure can vary, but it gathers members with common interests around a project. These members can be involved in developing the project or even only interested in using its results. According to (Gacek and Arief, 2004), people who collaborate with a OSS development always use the produced code. It makes it difficult to distinguish between users and developers, since all users have the potential to become a collaborator. ((Gacek and Arief, 2004); (Yunwen and Kishida, 2003)) Figure 17 shows how the OSS structure is organized, presenting the relation between users and several types of developers and their functions in the community.

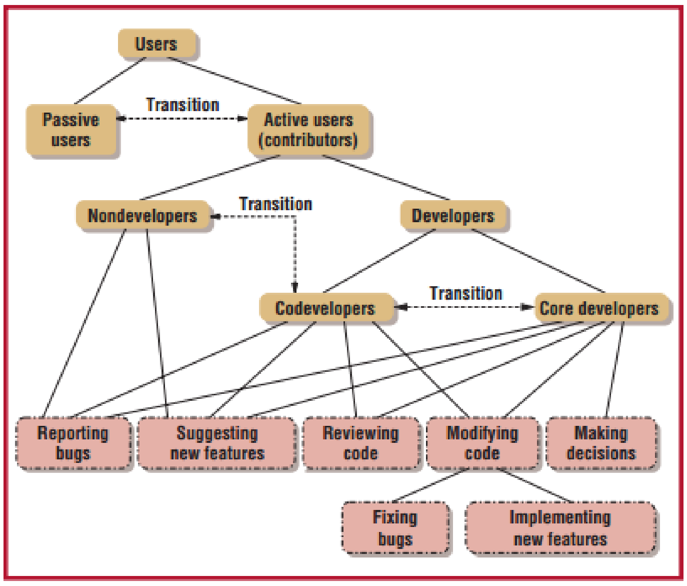


Figure 17 – Types of open source users and developers. (Gacek and Arief, 2004)

The group of developers, users and user-turned developers form a community of practice that is a group of people who work together pursuing a common objective. In order to be successful, an OSS needs a well-structured community to support the collaboration between developer and user. (Yunwen and Kishida, 2003) The members of the community (specially the contributors) need to be motivated to work in order to provide the ideal conditions for the OSS to progress. Most of the contributors are experienced programmers and although most of them act voluntarily, some are members of companies that support their participation. (Lakhani et al., 2002)

In order to have better chance to progress, the OSS community needs to have some users with innovative profile, these users need to be eager to spread their innovation and this diffusion of innovation needs to be able to compete with commercial production. (von Hippel, 2001)

Usually, OSS communities face two major issues related with the community's structure: the balance of centralization and the meritocratic culture. Depending on how strict the OSS community hierarch is, the decision making can be more centralized (for strict hierarchies) or more decentralized (for looser organizational structures). In strict communities, the higher hierarch people have, the more power they have and, consequently, more control over the decision making process. In looser communities, all the developers are in the same level, implying in decision making based on full consensus. OSS communities greatly relay on the contributors' perceived merit, which will increase as the contributor collaborate with the community development. The more relevant are the contributions the bigger is the power owned by the contributor, which can ascend (in some communities) from a passive user to a code developer, having more influence in the official releases of the code. The way this kind of transition happens, if it happens at all, depends on the project's organizational structure. (Gacek and Arief, 2004)

4.4 How it Can Succeed?

According to (von Hippel, 2001) "User Innovation Communities Shouldn't Exist, but they do". There are two main factors playing against the OSS communities when compared to commercial enterprises. The first one refers to the existence of a financial incentive for commercial manufactures. Different from individual developers and innovators who usually only benefit from the use of the developed code, the companies can expect to sell their product and make profit. The second point refers to the companies' infrastructure, which can provide better production, distribution and field-support capabilities. These factors grant great advantages for widespread diffusion of innovation (von Hippel, 2001)

The OSS work arrangement is completely different from the one in the conventional industry style. The OSS systems are built for a potentially bigger number of people, there is no work assignment since people work only with what they find pleasant, there are no conventional organizational elements such project plan, schedule, or list of deliverables and usually the contributors are geographically sparsely distributed. Contrary to what is expected in this kind of work arrangement, the OSS development is often equivalent, or even superior to traditionally developed products. (Mockus et al., 2000)

There is no doubt about the capacity of OSS community to produce good quality products, both regarding the quality and the support users receive, although the reasons for that are not completely understood and proven yet. One possible advantage of OSS is that the quality of the final result tends to be higher, since contributors code with care and creativity due to the satisfaction obtained through the work. Also, in an OSS development, problems and defects are quickly found and fixed, since there are a lot of people involved not only with the code development itself, but also with the use of this code. (Mockus et al., 2000)

According to (Bonaccorsi and Rossi, 2003) the success and diffusion of a OSS is closely related to the positive beliefs that software has among the users. In the presence of well-established commercial software standards, the OSS depends largely on several sources of network externalities. With no established commercial standards and in the presence of well diffused OSS, commercial companies need to heavily invest on R&D aiming to increase quality in order to be able to fight the OSS control of the market.

As also pointed by (Bonaccorsi and Rossi, 2003) simulations, although there is a natural and inevitable competition between OSS and commercial software, they "…are likely to coexist even in the limit.".

4.5 JavaScript as an Open Standard

An OSS needs to be written in a computational language that is not propriety, in other words being owned and controlled by one company. An open computational language can be called an Open Standard (OS). This is the case of JavaScript, which is a particular implementation of the ECMAScript language standard.

An OS like JavaScript is open but not open source, since is it not a program. It is only a document describing expected behaviors for lines of code written in its computational language syntax. Although an OS cannot be open source, an implementation of it can be (and in this case, it is an OSS).

In this section, JavaScript is presented as an OS, showing its background, main features and justifying why it will be chosen for developing the conceptual ship design open source tools library.

4.5.1 Object Oriented Programming in JavaScript

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data and code. In OOP, computer programs are designed by making them out of objects that interact with one another. The most popular and developed model of the OOP is the class-based programming (CBP). In this model, objects are entities that combine data, behavior and identity. The structure and behavior of an object are defined by its class, which includes all objects of a specific type. The objects are created based on classes and are considered instances of them, inheriting some of their properties.

JavaScript does not follow the CBP model. It is structured as a class free language, where objects inherit properties from other objects. This model is called prototype-based since behavior reuse is performed via a process of cloning existing objects that serve as prototypes. (Stefanov, 2010) This approach is powerful, making the inheritance process easier to implement, but it is also way different from what a conventional CBP language is. (Crockford, 2008)

An object, which is the JavaScript's core data type and its only complex one, is an unordered list of primitive (Number, String, Boolean, Undefined, and Null) and complex data types that are stored as a series of key-value pairs. The key property serves as an identifier while the value represents the value of the expression, which can be a primitive or an object value. Each item in the list is called a property or, if an item is a function, it is called method. (Stefanov, 2010) This easy notation inspired JSON, a popular data interchange format. (Crockford, 2008) An example of an object (car) containing both properties (type, model and color) and methods (showColor()) can be seen below.

```
var car = {                                 // car object.
    type:"Fiat",                            // car object's property.
    model:"500",                            // car object's property.
    color:"white",                          // car object's property.
    showColor : function () {               // car object's method.
        alert(this.color)                   // showColor method's description.
    };
}
```

One of the fundamental concepts from OOP is the inheritance concept. Usually in a CBP language, objects are instances of classes, from which they can inherit properties and functions. In JavaScript, this process is a little different since objects inherit from other objects.

Another important concept of OOP is the encapsulation concept. Encapsulation refers to enclosing all the functionalities of an object within that object so that the object's methods and properties are hidden from the rest of the application, making it possible to abstract or localize a specific set of functionalities on objects.

These two concepts will be important in this work, since they allow the construction of applications with reusable code, scalable architecture, and abstracted functionalities.

4.5.2 Object Definition in JavaScript

The objects definition in JavaScript can be done in three different ways, using object literals, using the new keyword or using the Object.create() method. (Flanagan, 2011)

The object literals are the simplest way to define and fully initialize an object in JavaScript. It can be written as a list of key-values pairs, separated by commas and inside curly brackets. Each time an object literal is evaluated it will create and initialize a new object, so the same literal can be used to create a bunch of different objects, if used recursively.

```
var empty = {};                  // An object with no properties.
var point = {x:0, y:0};          // Two properties.
```

The new operator can be used to create and initializes a new object. The new object is created using a function invocation, called constructor. Besides offering built-in constructors for several types of object, in JavaScript it is possible for the user to define specific constructors in order to initialize specific objects.

```
var o = new Object();            // Create an empty object: same as {}.
```

The last implementation of JavaScript OS, ECMAScript 5, brings the Object.create() method, which can be used to create a new object from a given prototype. This method has the powerful capability of creating object that inherit properties from any object.

```
var o1 = Object.create({x:1, y:2}); // o1 inherits properties x and y.
var o2 = Object.create(null);       // o2 inherits no props or methods.
```

This great set of forms to define objects makes JavaScript a really eclectic and powerful standard for dealing with OOP, avoiding the complication brought by the use of classes and also providing great capabilities from an inheritance and code reuse point of view.

4.5.3 Why JavaScript?

The idea in this work is not only to create an open source tools library, but also to create a ship design tool that is simple to use, not computational intensive and requires as minimum effort as possible to share designs and results.

Working with a software that requires to be installed in the computer could make it difficult to share results with clients, team members or other stakeholders. Using an online platform for the vessel design can reduce this information sharing difficulty, as the only thing one needs to access the information about the design is a web browser, and the only thing needed to edit is a text editor, reducing the need for client-side software to a minimum. Also, since all the processing is done in browser, the computational requirements are low.

Web development is not restrict to one operational system or one platform, since internet is universal. When thinking about client-side web development, JavaScript is an obvious choice. JavaScript is so important and popular because it is the language of the web browsers. (Crockford, 2008) It is supported by all modern web browsers without the need of plugins, since each browser has its own built-in JavaScript engine. (Flanagan, 2011) JavaScript composes a triad of web technologies that are essential for web development, together with HTML to specify the content of the web page and CSS to specify the presentation of the web page. JavaScript is responsible for describing the behavior of the web page. (Flanagan, 2011)

The position of JavaScript as the main languages in web browser makes its development fast, with new tools and libraries being developed all the time by the gigantic JavaScript's community.

Since JavaScript is prototype-based with first-class functions, supporting object-oriented, imperative, and functional programming styles (Flanagan, 2011), its prototype and inheritance capabilities make it good choice for dealing with objects to handle the vessel subsystem division and the knowledge base data. It has a simple API for working with text, arrays, dates and regular expressions, which can be completed using third parties' APIs. It has no input-output functionalities, relaying on the environment where it is embedded to handle these operations. (Flanagan, 2011)

The biggest drawback of JavaScript is the fact it is an OOP language that is PBP model. This makes of it an unusual language for most developers, which are accustomed to conventional CBP model. Directly using programming techniques from CBP will not work in JavaScript, which can be frustrating for an unadvised programmer. (Crockford, 2008)

4.5.4 The UML Class Diagram

For modeling the static design view of the vessel prototype, the UML Class Diagram (Booch et al., 1998) is going to employed. It is the most common diagram for modeling object-oriented systems and presents the systems' classes, interfaces, collaborations and their relationships.

In order to read a class diagram, some simple knowledge about its representation pattern is needed. Firstly, the UML provides a graphical representation of class. This notation allows to visualize abstractions created to represent elements of the designed system and lets you emphasize the most important parts of these abstractions: their name, attributes (JavaScript properties), and operations (JavaScript methods). (Figure 18)

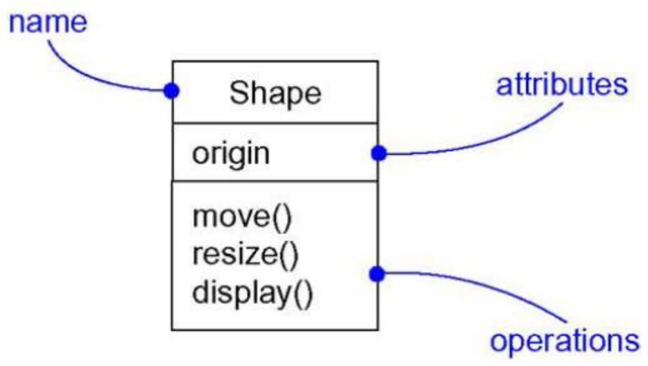


Figure 18 – Graphical representation of a class. (Booch et al., 1998)

In UML diagrams, classes can be related using the notations presented in Figure 19.

The association is the most abstract way to describe the relationship between classes, stating that there is some kind of link or dependency between two classes or more.

The inheritance relation is used to indicate where a class inherits properties and methods from. The arrow points from the subclass (child) to the superclass (or parent).

The realization relation exists between two classes when Class A realizes the behavior specified by Class B.

The dependency relation exists between two elements when changes to the definition of one element may cause changes to the definition of the other element.

The aggregation can be used in cases where there is a part-of relationship between Class A (whole) and Class B (part), but with no strong life-cycle dependency between them.

The composition relation works in cases where there is a part-of relationship between Class A (whole) and Class B (part), but also there is a strong lifecycle dependency between the two and the “whole” class has exclusive ownership of its “part” elements.

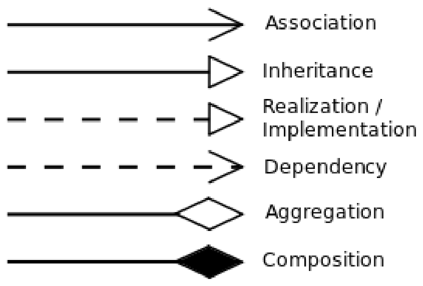


Figure 19 - UML relation notations.

In order to specify how many instances of a class can be connected by a relationship to an instance of another class, the multiplicity concept is used. The multiplicity is placed in the ends of the relation arrows and refers to the classes nearest to each end. The UML multiplicity notation can be found in Table 1.

0	No instances
0..1	No instances, or one instance
1	Exactly one instance
0..*	Zero or more instances
*****	Zero or more instances
1..*	One or more instances

Table 1 - UML multiplicity notation.

The last UML concept to approach here is the visibility concept. The visibility specifies whether an attribute or operator from Class A can or cannot be used by Class B. Table 2 shows the visibilities and their symbols, which are placed at the left side of the attribute or operator’s name. If the visibility is public, any class can use it. If it is private, only the class who owns it can use it. If it is protected, the class who owns it and classes derived from it can use it.

+	Public
−	Private
#	Protected

Table 2 - UML visibility notation

5 Research Approach

In order to build up the open source tools library, I am going to deal with the three main topics of this work: KBD, CSD and OSS. The KBD theory will be responsible for the prototype concept and knowledge-base. The CSD method selected is the SyBSD theory. The open source tools library will be developed using JavaScript as an OS. Selecting the relevant topics in each subject, the work scope is reduced and can be seen in Figure 20.

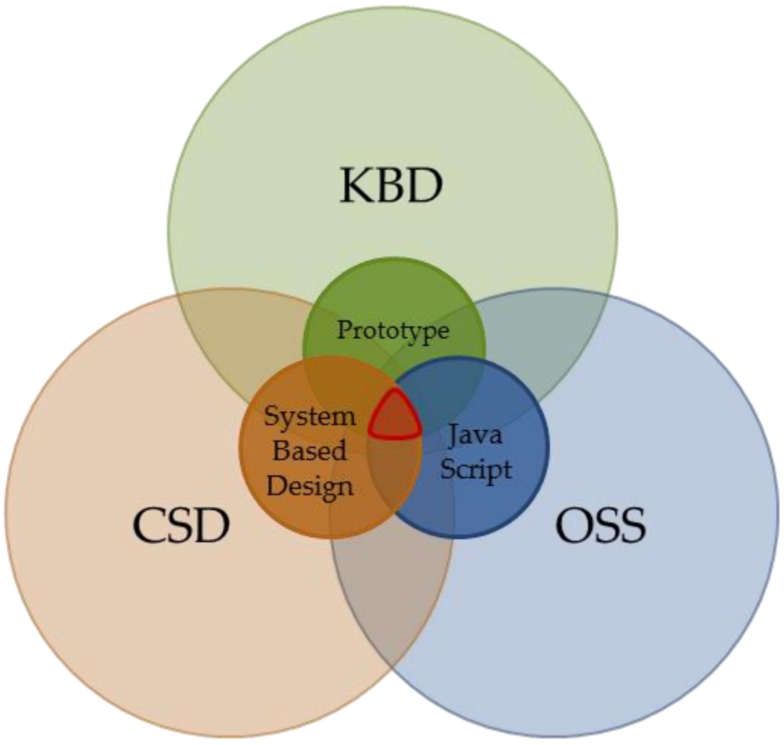


Figure 20 - Narrower scope used on the research approach.

This chapter will be developed following the system architecture presented in Figure 21, where the tools library components are organized and their relations stablished. The User provides inputs to the library and receives outputs from it. The Tools Library is developed using JavaScript, for both Prototype and Knowledge-Base. The Prototype contains the most important KBD elements, namely Function, Behavior and Structure. The User's inputs feed the Function block, while the Structure and Behavior blocks are developed using SyBSD theory. The Structure and Behavior blocks receive information from the Knowledge Base through an Inference Mechanism, respectively from the SyBSD Structure Database and Regression Database blocks.

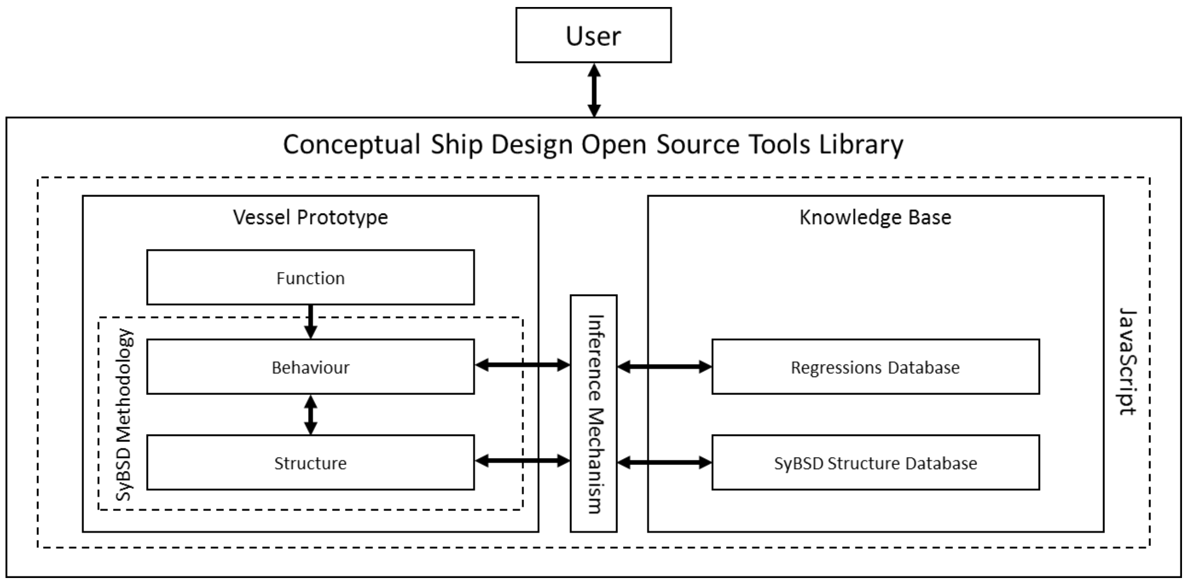


Figure 21 - Conceptual Ship Design Open Source Tools Library architecture.

Following this architecture, I am going to start this chapter with the Knowledge Base creation and organization. Secondly, the way the SyBSD methodology is used to structure the prototype and conduct the design process will be presented. Finally yet importantly, the Vessel prototype and tools library construction using JavaScript is approached, giving special attention to how the OOP capabilities can be used to structure, represent and handle knowledge about the CSD process.

5.1 Knowledge-Base

One important element in KBD is the knowledge-base. It is responsible for storing knowledge about the design process, which can be accessed by some sort of inference mechanism to retrieve facts, knowledge and control whenever the reasoning process requires. The tools library knowledge base was constructed to be the most generic as possible in order to provide knowledge to the design of several kinds of vessels and is composed by two databases: the Regressions Database and the SyBSD Structure Database.

The regression database contains important vessel design coefficients regression and knowledge about previously built vessels of several types, including container carriers, bulk carriers, tankers, ferries, roro and offshore support vessels. As examples of these regressions, it is presented below the relation between midsection, prismatic and waterline coefficients and Froude number.

```
var data_CM = [[0, 0.100, 0.200, 0.300, 0.385, 0.400, 0.454, 0.500],
               [1, 0.998, 0.997, 0.978, 0.900, 0.879, 0.800, 0.740]];

var data_CP = [[0, 0.100, 0.200, 0.256, 0.300, 0.320, 0.401, 0.500],
               [1, 0.914, 0.799, 0.699, 0.624, 0.598, 0.550, 0.583]];

var data_CW = [[0, 0.100, 0.200, 0.219, 0.299, 0.328, 0.400, 0.500],
               [1, 0.964, 0.915, 0.899, 0.822, 0.800, 0.749, 0.715]];
```

The complete regressions database can be found in APPENDIX C.

The SyBSD database contains class definitions based on the SyBSD structure. These classes are used to instantiate vessel elements, which compose the vessel's subsystem, which in turn compose the vessel's systems. Whenever the user instantiates a new element, this database will be accessed and the required class structure will be retrieved. As an example, the SyBSD structure of the outfitting system can be seen below, where the outfitting system is presented with its subsystem structures.

```
outfitting

operational_support
{"area":"", "covered":"", "height":""}

ship_equipment
{"area":"", "covered":"", "height":""}

rescue_firefighting
{"number_units": "", "area_unit":"", "covered":"", "height":""}
```

The complete SyBSD database can be found in APPENDIX B.

5.2 System-Based Design Methodology

The SyBSD methodology was used to define both the vessel structure and the vessel behavior. The structure relays on the system subdivision apply in SyBSD theory, which is based on the SFI grouping system division. The vessel's behavior can be obtained from direct application of the SyBSD methodology. This application, besides providing the vessel's main dimensions and behaviors, also prepares the terrain for the application of more complex tools, which can be used to obtain more complete and complex figures about the vessel's actual behavior.

These two main applications of the SyBSD methodology will be discussed in the following sections.

5.2.1 Vessel’s Structure

The proposed vessel prototype subdivision structure is presented in Figure 22. SyBSD uses a simple division for the physical structure of the vessel. There are two main groups of systems: Task Related Systems [1.1] and Ship System [1.2]. The Task Related Systems group includes any cargo and cargo handling systems and specialized system for offshore support vessels which are not related directly to cargo but are related to the money making capacity of the vessel. The Ship Systems group includes any system required for the vessel to operate safely and considers the Outfitting [1.2.1], Crew [1.2.2], Service [1.2.3] and Machinery Systems [1.2.4]. More information about the SyBSD structure can be found in APPENDIX B.

Besides the vessel physical structure, there are also other important elements to consider in the prototype structure. I am defining two JavaScript objects to store data. The first one is related to the required functions of the vessel, namely the Mission Requirements object [1.4]. The second is responsible for storing the vessel’s main dimensions and behaviors, namely the Main Dimensions object [1.5].

Lastly, the prototype will hold several JavaScript methods (or functions), which will be responsible for data handling, reasoning, knowledge retrieve and knowledge application. The methods present in the vessel prototype are: Prototype [1.3.1], Area [1.3.2], Volume [1.3.3], Light Weight [1.3.4], Dead Weight [1.3.5], Displacement [1.3.6], Main Dimensions [1.3.7], Holtrop [1.3.8], Seakeeping [1.3.9] and Hull Lines [1.3.10]. More information about each method and its application can be found in APPENDIX A.

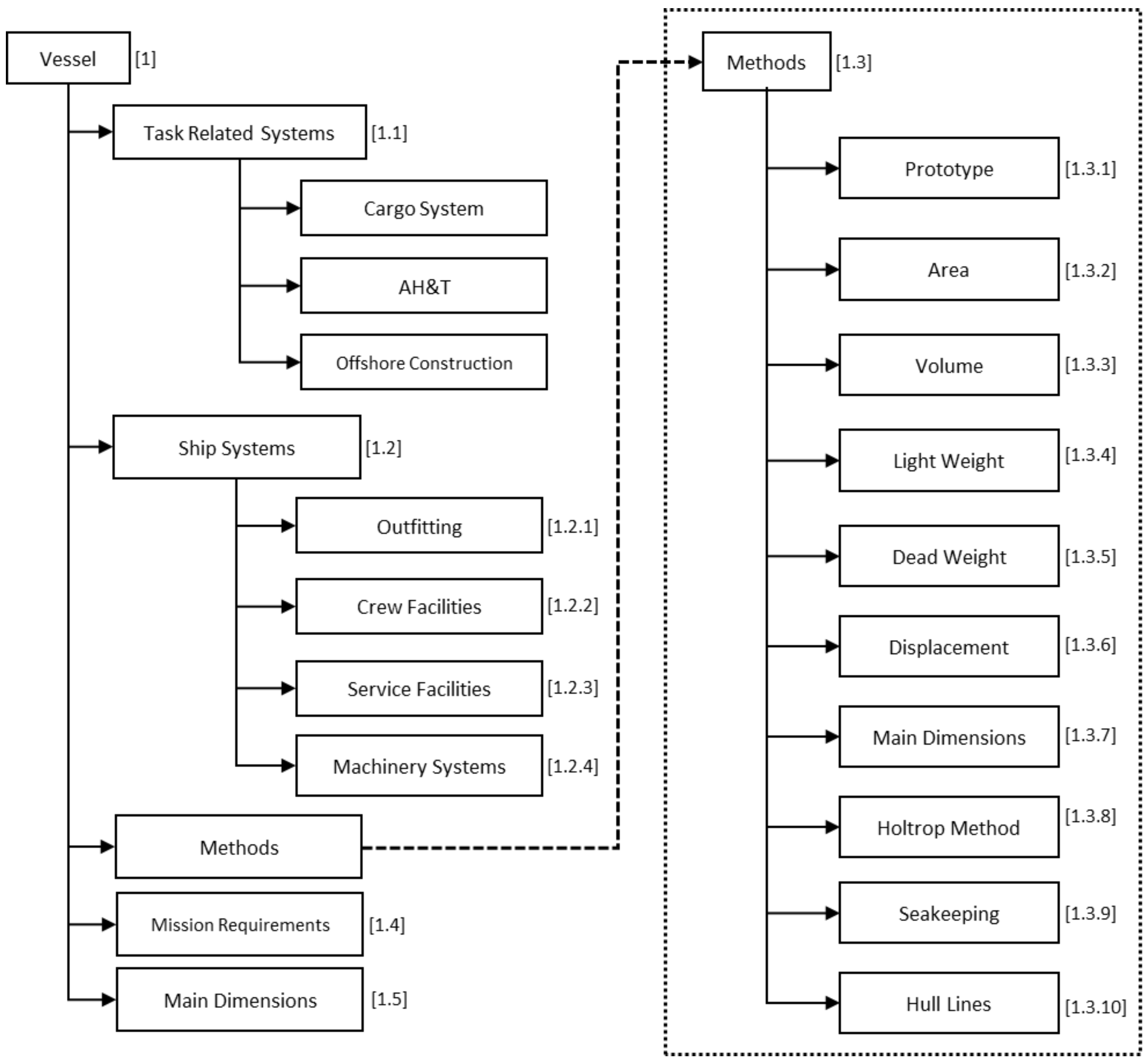


Figure 22 - Vessel prototype subdivision structure.

5.2.2 System-Based Ship Design Process

In order to apply the SyBSD methodology, the workflow presented in Figure 23 was developed. It includes the main design process the user should perform while applying the ship design tools library. Some of the phases relay on the users' knowledge, while others relay on the developed knowledge-base.

The workflow is divided in 8 phases, which will be detailed hereafter. Each phase is represented by a block in Figure 23, and the phase number is identified by the number in the top right corner of each block. The SyBSD spiral stages are indicated by the dashed boxes.

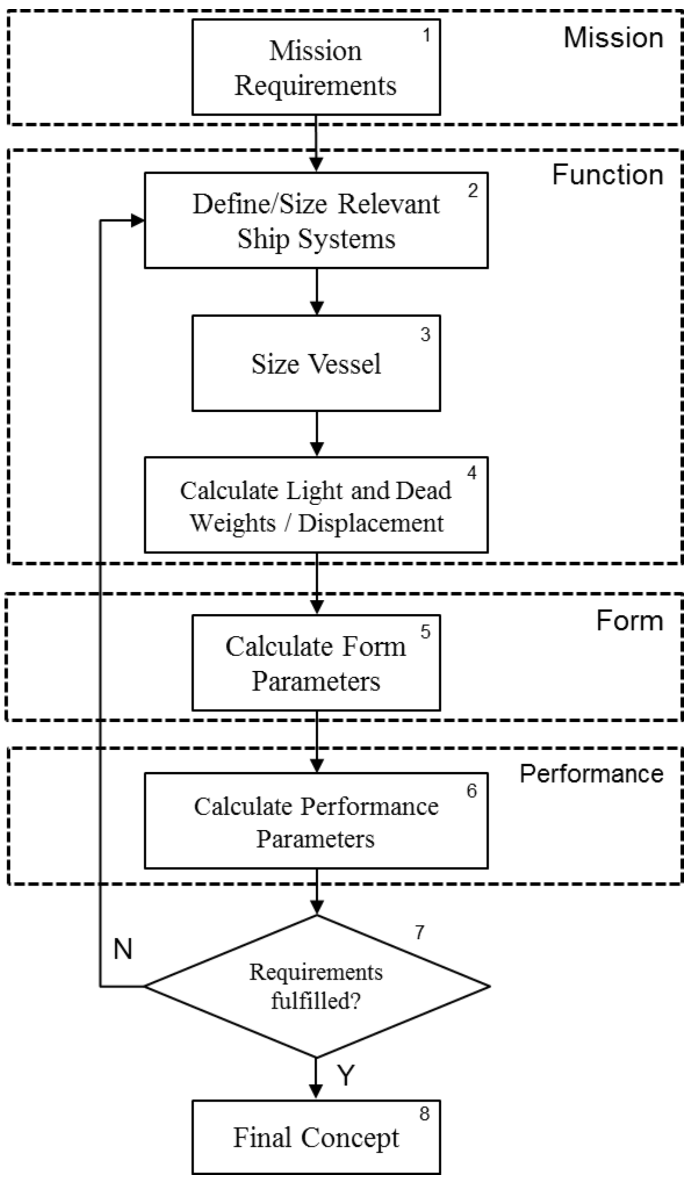

Mission
Mission Requirements
1
Function
Define/Size Relevant Ship Systems
2
Size Vessel
3
Calculate Light and Dead Weights / Displacement
4
Form
Calculate Form Parameters
5
Performance
Calculate Performance Parameters
6
N
Requirements fulfilled?
7
Y
Final Concept
8

Figure 23 – System-based ship design process workflow.

Phase 1. The first phase on the workflow represents the Mission stage on the SyBSD spiral. Its goal is to obtain the mission requirements from the ship owner. This information will be crucial to correctly size the vessel, since it will define parameters such as cargo type and capacity, autonomy, speed, crew size, special functions, etc. These parameters are defined based on the expected tasks, capacities and performance from the vessel.

The next three phases compose the Function stage of the SyBSD spiral.

Phase 2. This phase consists in defining and sizing the relevant systems for the vessel so that it is able to perform and fulfill all the mission requirements. This approach can be characterized as a Bottom-Up (Coyne et al., 1989), approach, since it starts from individual subsystems to compose the vessel, instead of fitting the subsystems inside the vessel.

The basic system division of the SyBSD methodology consists of two main subsystems groups: Task Related (or Payload Related) Systems and Ship Systems. The first one is related to the vessel's money making potential and will enclose all the system's need for performing the vessel's specific tasks. These tasks can be related to cargo or people transportation, offshore construction, anchor handling, etc. The second subsystems group is related to the vessel's needs to perform the required tasks. This group includes subsystems such as machinery, structure, tanks, accommodations, etc. The traditional SyBSD breakdown structure for an offshore support vessel can be seen in Figure 24.

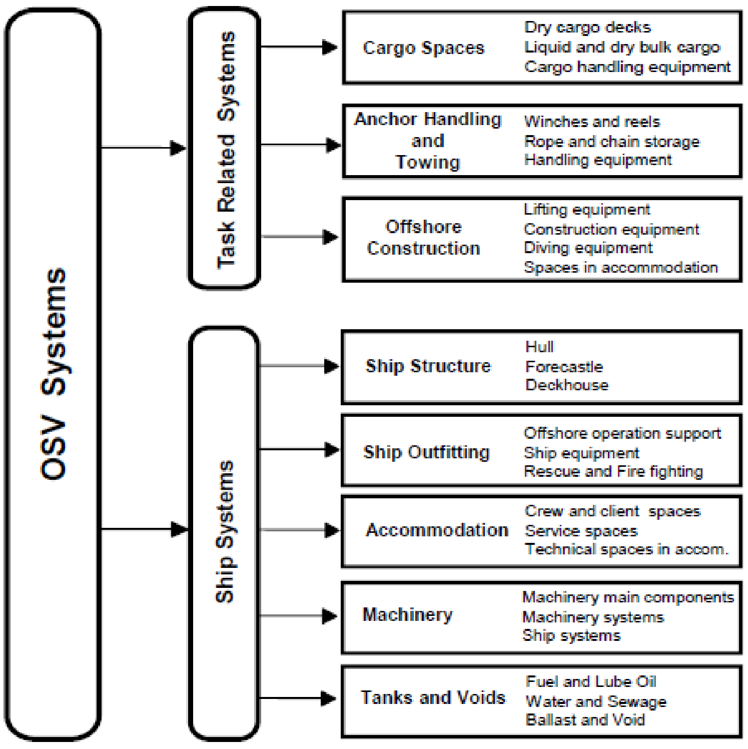
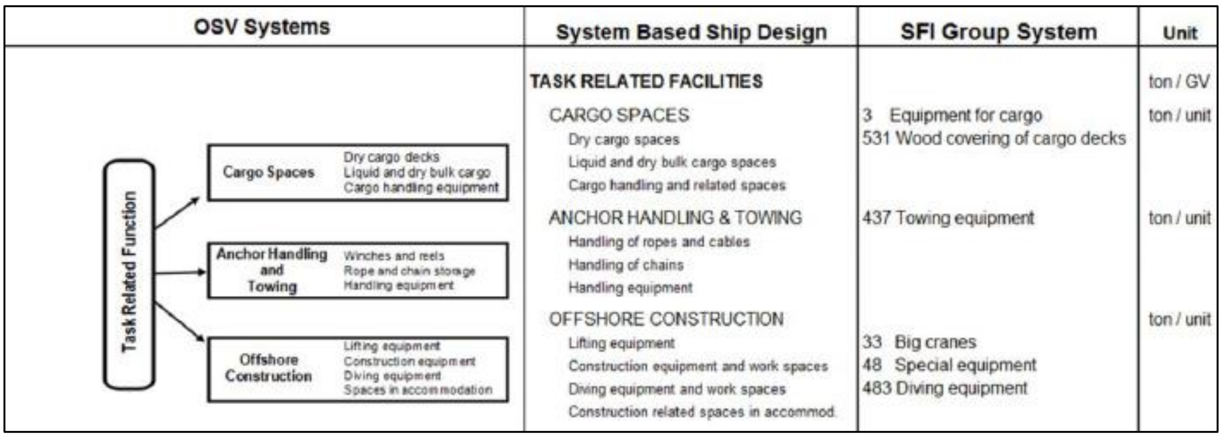


Figure 24 – Example of an OSV function breakdown structure. (Erikstad and Levander, 2012)

The SyBSD methodology uses a grouping system which is closely related to the one present on the SFI Group System. Although they are very similar, the SFI Group System does not distinguish between tasks related systems and ship systems. There are also some elements that are located in different groups in both grouping systems. In this case, some special relations can be made to completely relate the SyBSD theory to SFI Group System. One good example of this is the case of the anchor handling and towing equipment, which is part of the task related systems on the SyBSD methodology but is located at the ship equipment category (Item 437) on the SFI Group System (Figure 25).



OSV Systems	System Based Ship Design	SFI Group System	Unit
Task Related Function	TASK RELATED FACILITIES		ton / GV
Cargo Spaces: Dry cargo decks, Liquid and dry bulk cargo, Cargo handling equipment	CARGO SPACES	3 Equipment for cargo	ton / unit
	Dry cargo spaces	531 Wood covering of cargo decks	
	Liquid and dry bulk cargo spaces		
	Cargo handling and related spaces		
Anchor Handling and Towing: Winches and reels, Rope and chain storage, Handling equipment	ANCHOR HANDLING & TOWING	437 Towing equipment	ton / unit
	Handling of ropes and cables		
	Handling of chains		
	Handling equipment		
Offshore Construction: Lifting equipment, Construction equipment, Diving equipment, Spaces in accommodation	OFFSHORE CONSTRUCTION		ton / unit
	Lifting equipment	33 Big cranes	
	Construction equipment and work spaces	48 Special equipment	
	Diving equipment and work spaces	483 Diving equipment	
	Construction related spaces in accommod.		

Figure 25 – System-based design and SFI Group System special relation example. (Levander, 2012)

Based on the chosen subdivision, the areas and volumes to accommodate all the subsystems need to be calculated. This is made through the information provided by the user. Each class of subsystem requires different parameters to define its area and volume. These parameters are related to the subsystems' capacities and functions. More information about the required parameters for each class of subsystem can be found on APPENDIX B. For some subsystems, area and volume are relevant but there are some subsystems for which only area or volume can be important. For example, for tanks and voids systems, the SyBSD methodology is only interested in the volume of the system. Figure 26 schematizes the systems' sizing process.

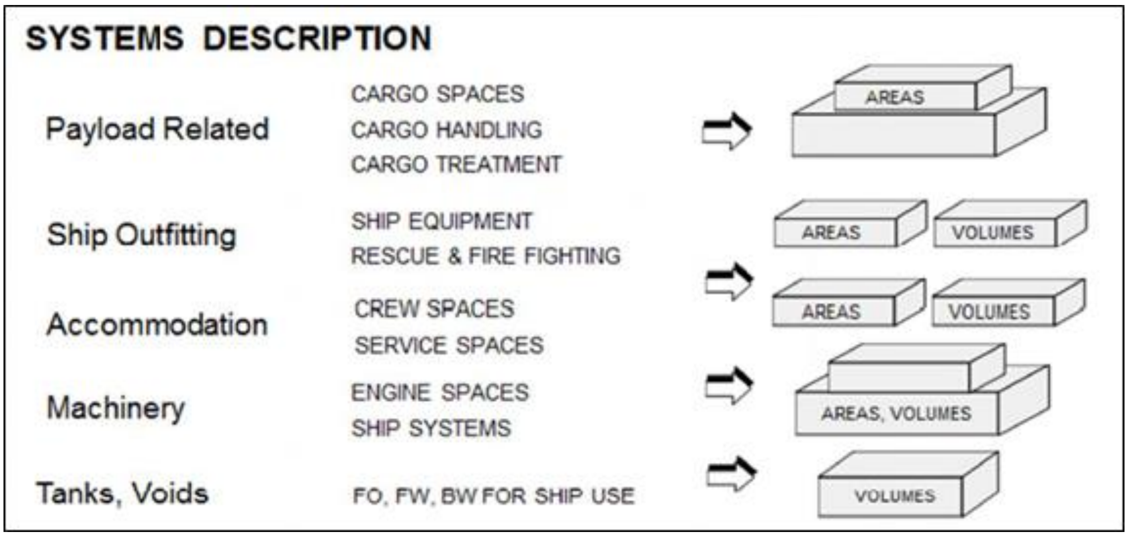


Figure 26 - Systems sizing. Adapted from (Levander, 2012)

Phase 3. In this phase, the vessel is sized based on the areas and volumes of all previously defined subsystems. The operation is automatically done by the library, considering for each subsystem if the area and volume properties are relevant or not and is based on the developed knowledge-base. This phase can be summarized according to Figure 27.

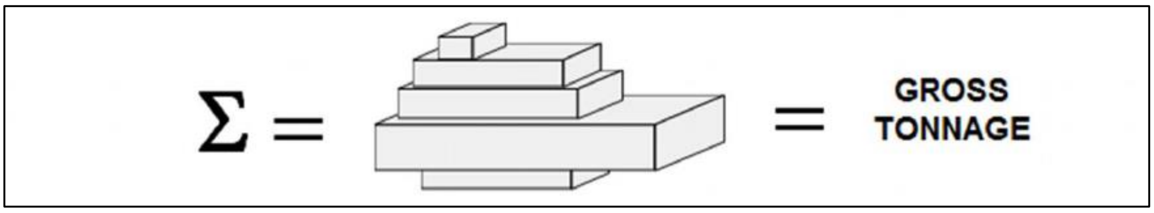


Figure 27 - Vessel sizing. Adapted from (Levander, 2012)

Phase 4. In this phase, the vessel displacement is defined. The total displacement of a vessel can be basically divided in two categories: Lightweight and Deadweight. According to (Parsons, 2003), the first group represents the weight of the vessel that is ready to go to the sea, with neither loads nor cargo. It can be roughly divided in structural weight, machinery weight, outfitting weight and margin weight. The margin weight is responsible for protecting the design from underestimation of the needed displacement. Parsons also states that the Deadweight can be portioned in payload (or cargo deadweight), fuel oil weight, lube oil weight, fresh water weight, crew and their effects weight and provisions weight. In SyBSD approach, the weight estimation is done following the weight groups stablished in Figure 28, which although are portioned in a slightly different way from Parsons' approach, address all the needed weight information.

The estimation of weight at the early parametric stage of design typically involves the use of parametric models that are typically developed from weight information for similar vessels. The lightship weights estimation was conducted using the representative parameters for each item (ship equipment is based on gross volume, accommodation outfitting is based on the accommodation area, total installed power is used for machinery and area for hatch covers) and a collection of coefficients (APPENDIX C) based on statistical data provided by (Levander, 2012). This data is divided by lightweight group and ship type, and is presented as regression curves, which were included in the knowledge-base.

For the deadweight groups calculation (excluding the payload weight) the weight estimation is based on the consumptions of each resource and their densities. For the Provision and Stores and Crew and Their Effects weight groups calculation, coefficients suggested by (Parsons,

2003) were included in the knowledge-base, taking also into account the number of people on board and the vessel endurance.

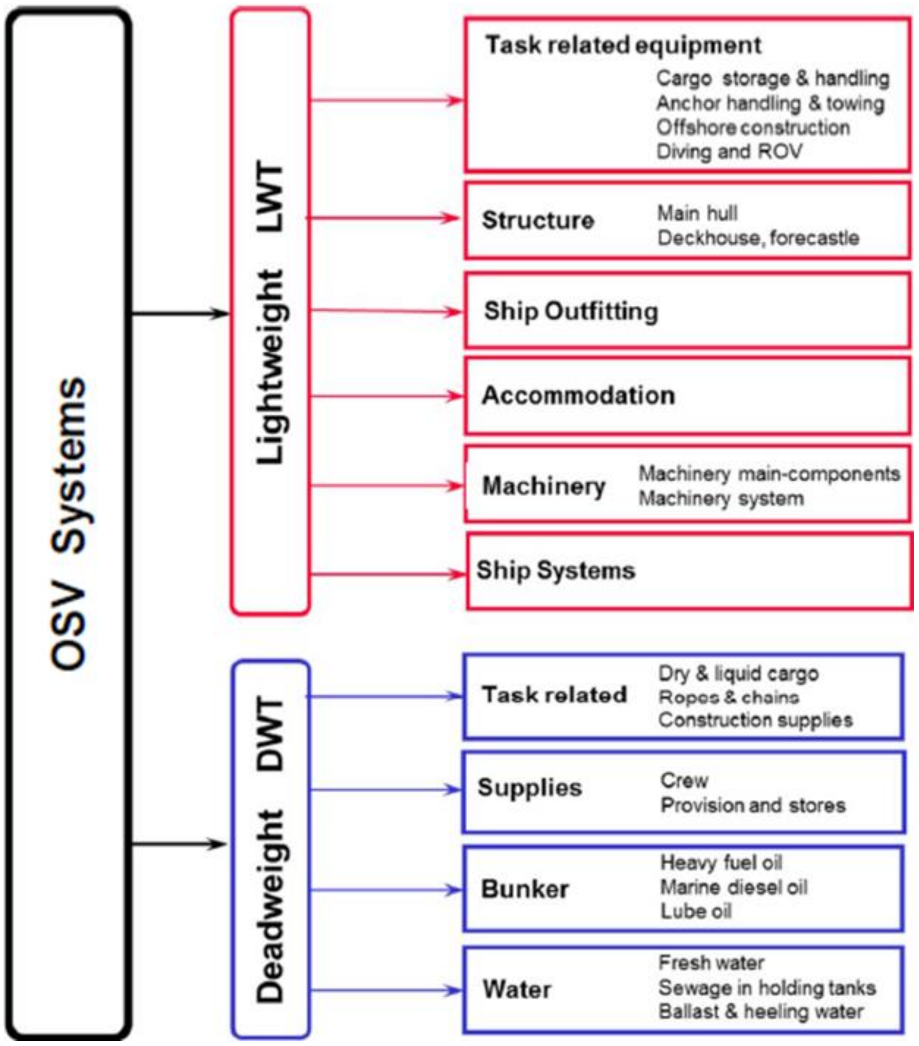


Figure 28 - Example of an OSV light and dead weights main groups. (Erikstad and Levander, 2012)

Phase 5. The methodology had mitigated the decision process about the vessel's main dimension until now, when a rough idea about the required areas, volumes and weights is available. Now, with more knowledge about the system, it is possible to take better decisions about the future of the design.

This phase comprehends the rough estimation of the vessel's forms (Figure 29). In order to specify the vessel's hull, having the area, volume and weight distribution is crucial but, unfortunately, not enough. It is also important to have more specific information such as the vessel's dimension, how the volume is distributed along the hull, where the centers of gravity and buoyancy are located and so on. These parameters can be defined, based on both statistical regressions and empirical formulas, which are mostly implemented as expert's knowledge in the knowledge-base.

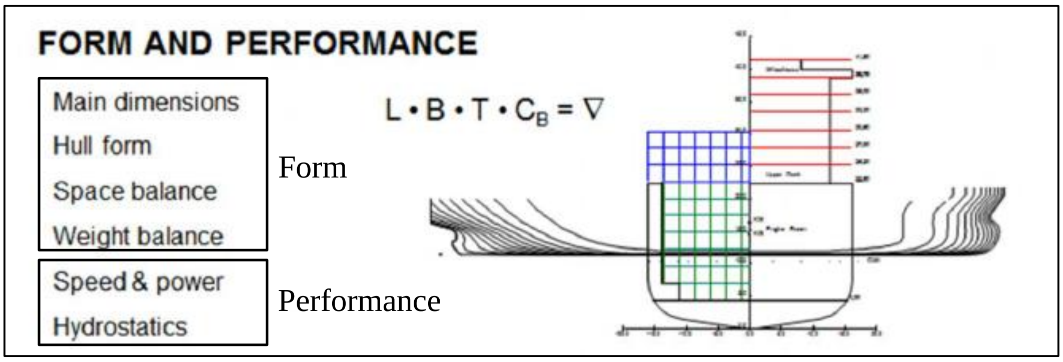


Figure 29 - Form and performance definition stage examples. (Levander, 2012)

For obtaining the main dimensions, regression analyses from similar vessels will be conducted. This approach is useful at a preliminar design stage and its results can be refined at later design stages, when the ship designer has more precise information to work with. The regression implemented in the regressions database (APPENDIX C) can be found at (Levander, 2012) and are presented as separated regression for each kind of vessel, in function of a relevant property of each type of vessel. For example, in the case of platform support vessels, the main dimension regressions are presented in function of the vessel's gross tonnage.

The hull form coefficients (block coefficient, prismatic coefficient, water plane area coefficient and mid-section coefficient) can be obtained from the Froude Number, based on regressions presented by (Levander, 2012). These regressions are not vessel type dependent, since the Froude Number itself already carries enough information about the vessel type. For slender and faster vessels, the Froude Number is higher than for bulkier and slower vessels.

For space and weight balance, several statistical, empirical and numerical expression were implemented in the knowledge-base. For most of the parameters related to space and weight balance, the expression is vessel type sensitive, so the expected results change according with the type of vessel.

Phase 6. This phase consists in evaluating the main performance indicators of the vessel. These indicators can refer to any performance required that the vessel should fulfill, such as maximum speed, wave resistance, seakeeping response and so on. They are closely related to the vessel's actual behaviors and should be compared to the expected behaviors in order to attest the rightness of the design.

This performance evaluation is conducted using JavaScript methods implemented in the tools library. Two performance methods were included in the library so far: the Holtrop and the Seakeeping methods. The first one gives a rough estimate about the vessel's wave resistance and the second gives some information about the vessel's seakeeping capabilities.

Phase 7. This phase consists in an evaluation process, where the user's knowledge and judgment are relevant to decide if the obtained result is good enough, fulfilling all the mission requirement without generating an unfeasible result. If, for any reason, the design process generates an unpractical design or a design that does not completely fulfill all the requirements, the user should return to the Phase 2, and update the subsystems' definition, using the obtained knowledge to make better assumption and decisions.

Phase 8. In case the result obtained in Phase 7 is a satisfactory design, the SyBSD is completed. With a feasible solution, several methods defined in the tools library can be applied to further progress the design. If during the application of any extra method any inconsistency is found, the user can return to the SyBSD process to perform further modifications. As the conventional PBD methodology, the SyBSD is also an iterative process and revisiting past stages of the design is a common place on the ship design process.

5.3 Vessel Prototype

As already stated by (Lee et al., 1996), it is difficult to obtain knowledge to compose a knowledge-base. Once the knowledge base is constructed, another problem needs to be overcome: how to efficiently handle all this knowledge. One efficient way of doing so is using OOP to handle the main elements of the KBD methodology.

The tools library is going to represent the vessel using an open prototype and in order to construct it, I am going to implement the tools library using object oriented programming concepts and JavaScript language as an OS. JavaScript was not designed as a conventional CBP language, but its object concept can be used to work around this issue.

5.3.1 Object Oriented Programming Implementation in JavaScript: Encapsulation and Inheritance

In order to implement the OOP in JavaScript, I am going to use two different techniques. The first one will be the encapsulation, for creating objects with specialized functionalities. The second one will be inheritance, for code reuse.

The encapsulation concept basically means to putting all the inner workings of an object inside that object. To do so, it is needed to identify and define the properties and methods of that object, so the encapsulation pattern to construct the object can be applied. Implementing inheritance in this application will allow to inheriting functionality from parent functions so that the code can be easily reused in the application and extend the functionality of objects, which can make use of their inherited functionalities and still have their own specialized methods and properties.

The best encapsulation mechanism in JavaScript is the Combination Constructor/Prototype Pattern. (Zakas, 2009) This method is not only capable of dealing with the encapsulation matter, but it is also possible to use it in order to implement inheritance through Prototypical Inheritance.

The use of encapsulation makes no sense if you only want to store data inside an object. For this kind of task, writing the object by using object literal is sufficient. But when you need to create several objects with similar properties and methods, it makes sense to encapsulate all these properties and methods inside a function and use it to construct these objects.

In order to exemplify the use of Combination Constructor/Prototype Pattern technique in JavaScript, I am going to present the implementation of the *systemPrototype* method, which is held by the vessel object and is used to instantiate new system objects. Each system object will contain subsystem objects instantiated by the user, following the defined SyBSD structure database (APPENDIX B). The *systemPrototype* method will need to write down the instantiated subsystems as pairs key-property inside the system object. In addition, each system object will have the following method: *add*, *area*, *delete*, *input* and *volume*. Since all system objects will have the same methods, the Prototypal Inheritance will be used to make the child system objects inherit the methods from the parent system object. Each system object will be afterward specified with the relevant properties addressed by the user.

Since I want all vessel systems to have the same methods, I can use a constructor function (class in OOP) to encapsulate these methods. In order to create the constructor function, the Combination Constructor/Prototype Pattern technique will be used.

The first step of the creation of the constructor function is to initialize the instance properties. These properties will be defined on each System instance that is created. The object doesn't have default properties, but it has a code routine responsible for getting the constructor input, searching in the SyBSD structure database for the subsystem classes and instantiating the required classes as properties. The properties values will be different for each System, depending on the user’s input. The use of the keyword *this* inside the function specifies that these properties will be unique to every instance of the System object.

```
// constructor function definition.
"systemPrototype": function(subSystem) {
    // loop for repeating operation according to size of the input.
    for(i = 0; i < subSystem.length; i++){
        // using input to select subsystem’s structures in knowledge 
base.
        var splited_subSystem 
=knowledge_base.split(subSystem[i][0]+"\n");
        if (splited_subSystem.length > 1){
            // further subsystem’s structure selection in knowledge 
base.
            var splited_subSystem2 = splited_subSystem[1].split("\n");
            // creating temporary object.
            var obj = {};
            // filling temporary object with subsystem classes.
            obj[subSystem[i][1]] = JSON.parse(splited_subSystem2[0]);
            // transfering properties from temp obj to vessel’s system.
            if(typeof this[subSystem[i][0]] == 'undefined'){
                 // if the subsystem does not exist yet.
                this[subSystem[i][0]] = obj;
            }else{
                // if the subsystem already exists.
                $.extend(this[subSystem[i][0]],obj);
            }
        }else{
           // error message for incorrect input.
            alert(subSystem[i][0] + "is not a valid subsystem name.");
        }
    }
}
```

After the constructor is defined, the next step is to overwrite the prototype property with an object literal, where all the methods that will be inherited by all the System instances are defined. By overwriting the prototype with a new object literal, all the methods are organized in one place, effectively implementing the encapsulation.

```
systemPrototype.prototype = {                   // overwriting prototype 
method.
    constructor: systemPrototype,               // overwriting constructor.
    this.input = function (vector){…},          // defining input method.
    this.add = function (subSystem){…},         // defining add method.
    this.delete = function (vector){…},         // defining delete method.
    this.area = function (){…},                 // defining area method.
    this.volume = function (){…},               // defining volume method.
}
```

The constructor property of the prototype was overwritten in order to simplify the access to the methods, since after overwriting the prototype, the function constructor does not point to the correct prototype anymore. Due to this problem, the user would need to specify the prototype property each time a method is called inside it. To avoid this, the new constructor needs to be set manually, as it was done. See the example below:

```
// method call without overwriting the constructor property manually.
systemPrototype.prototype.area();

// method call overwriting the constructor property manually.
systemPrototype.area();
```

When overwriting the prototype property, the function is prepared to provide Prototypal Inheritance. The properties and methods added on the prototype property will be inherited by each instance of the System object, so they can use them and also receive new properties and methods.

5.3.2 Object-Oriented Representation of Ship Design Knowledge

Object-orientation is usually both a language feature and a design methodology. (Meyer, 1988) It will be used to represent elements from the KBD theory. This representation will be made using some key features from Object-Oriented Design (OOD) (Booch, 1982), such as the concept of objects, encapsulation and inheritance. Object-orientation provides a flexible platform for developing conceptual design problems. (Zhang et al., 2001) In the next sections, the ship design knowledge representation is presented and explained. The system description is made using a Class Diagram from Unified Modeling Language (UML). (Booch et al., 1998)

5.3.2.1 Tools Library Structural Subdivision

In order to help you to better understand how the tools library is organized, in relation to the class structure in the object-oriented design methodology, Figure 30 is presented. This figure is helpful to understand how the vessel hierarchy is constructed in this prototype. Different kinds of subdivisions and organizations can create different hierarchical structures, but the one presented here aims to be as close as possible to the original SyBSD one. Since JavaScript's object is the platform used to model the vessel, it is easy to use the objects functionalities to organize the system hierarchy.

After instantiated, the Vessel is an object, containing properties and methods. The properties can be primitive values (such integers or strings) or other objects, while methods are functions (which is JavaScript are also objects, but which can perform tasks and operations).

When instantiated, the Systems are also objects, which contain properties and methods. Since they are objects, they can be stored as property of the Vessel object.

Once instantiated, the Subsystems are also objects, which contain properties. Since they are objects, they can be stored as properties of the System objects.

Lastly, the elements are also instantiated as objects, which contain properties. They can be stored as properties of the Subsystem objects. This hierarchical structure can be seen in Figure 30. One example of this organizational structure for and individual element would be:

Vessel(CargoSystem(CargoHoldDryBulk(IronOreHold))))

It is important to point out here that the SyBSD does not properly show the complex set of connections and interfaces between elements, even if they are part of the same system or subsystem. These connections and interfaces start to gain more importance in the next stage of the design process and cannot be ignored.

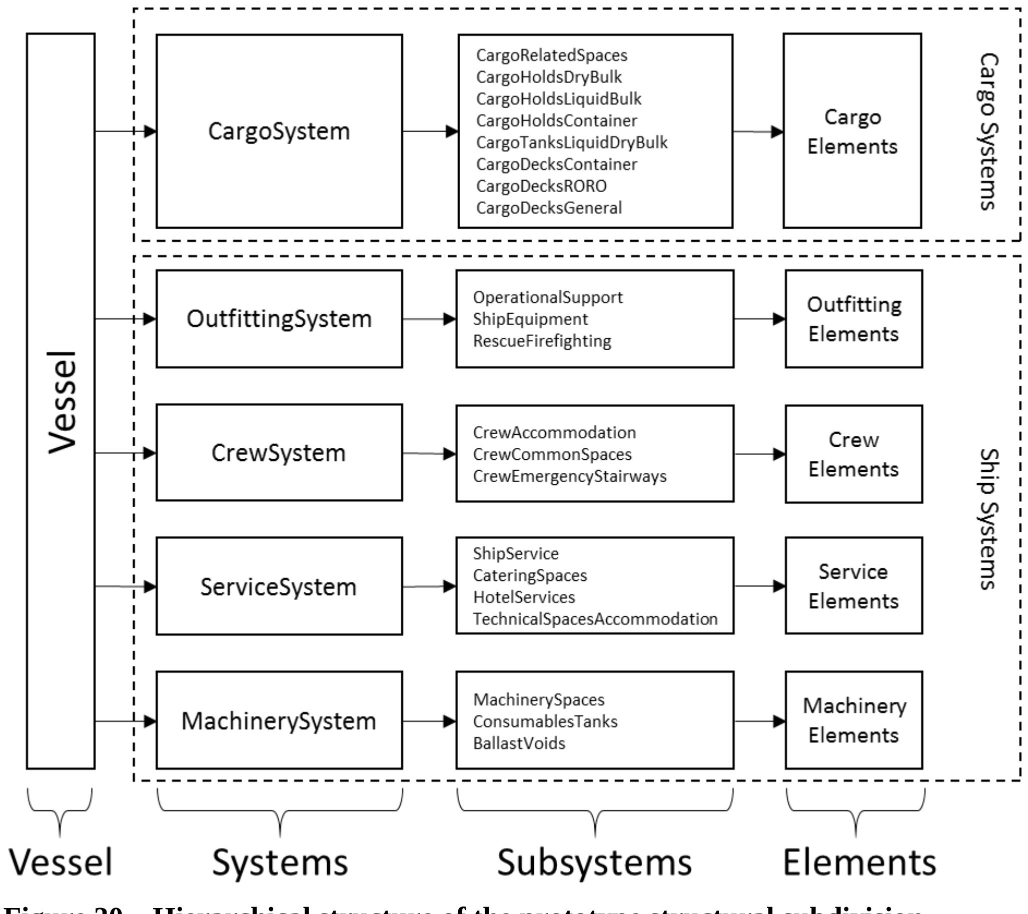


Figure 30 – Hierarchical structure of the prototype structural subdivision.

5.3.2.2 Class Diagram Representation of the Vessel Prototype

Using the UML class diagram presented in Chapter 4, the object-orient vessel prototype is modeled. The representation is crucial for providing such a complex relational structure of all the classes composing the vessel prototype.

The representation done here is a simplified one, where some less relevant aspects of the class diagram were neglected, such as methods' arguments, relations' labeling and responsibilities' definition. The neglected aspects can be useful in several situations, but for the reason the diagram is used here (mostly to make the relations between classes clearer) they are not needed. The main objective with this simplification was to make the diagram more readable in the limited space provided by this report's pages.

The vessel prototype class diagram can be seen in Figure 31. The main class of the vessel prototype is the *Vessel* class. Its attributes and operations are specified, although the input parameters for the operations were omitted.

The *Vessel* class is composed by *System* classes. This *System* classes are related to the *Vessel* class by means of composition relations of multiplicity zero or one, since each *Vessel* class can have zero or only one of each *System* class.

The *System* classes do not have any default attribute or operation, but they inherit operations from the class *VesselSystem*, which they are connected to by inheritance relations (which has no multiplicity).

The attributes of the *System* classes are represented by the *Subsystem* classes, which are connected to their respective *System* class by means of composition relations of multiplicity zero or more. This happens because System classes can contain any number of their respective Subsystem classes.

All the attributes and operations of the vessel prototype were considered public, since they all need to be manipulated by the *Vessel* class.

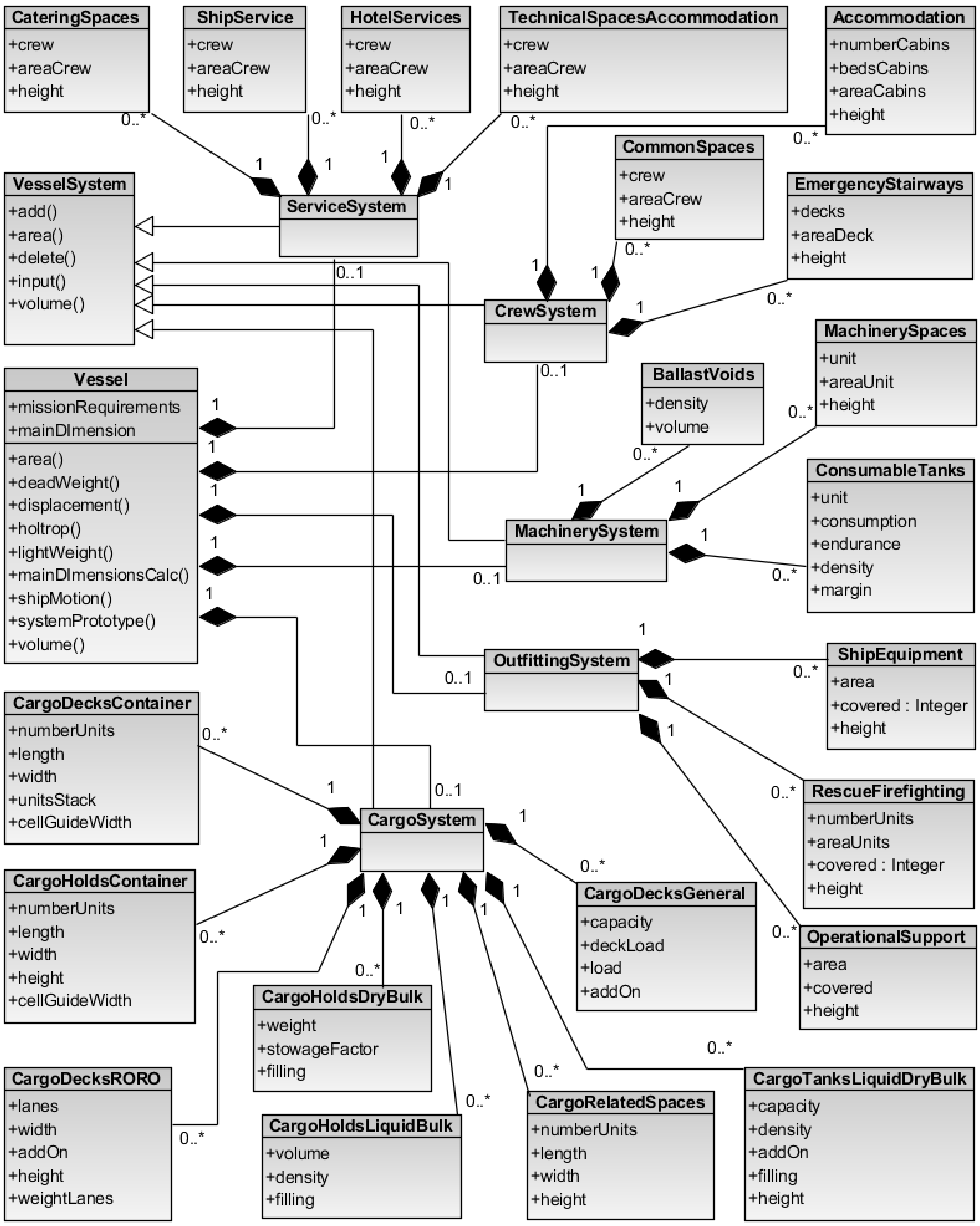


Figure 31 – Vessel prototype class diagram.

5.3.2.3 The Vessel Class

This vessel prototype is constructed as a class called Vessel. As it can be seen from Figure 31, it works as a container to receive all the vessel's systems and subsystems. It also stores and organizes all the vessel's properties and methods.

The Vessel class has two default properties, *missionRequirements* and *mainDimensions*. Both of these properties are JavaScript objects, which store all the vessel's properties and its values. The *missionRequirements* object will store the data used to instantiate the vessel class and the *mainDimensions* object will store the results from the conceptual design process, after the application of the *mainDimensionsCalc* method.

The Vessel class has several default methods, which are *area*, *deadWeight*, *displacement*, *holtrop, lightWeight*, *mainDimensionsCalc, shipMotion*, *systemPrototype* and *volume*. The methods *area*, *volume*, *deadWeight*, *lightweight* and *displacement* all self-explanatories and will provide, according to the systems and subsystems defined by the user, the vessel area, volume, deadweight, lightweight and displacement, respectively. The *mainDimensionsCalc* method calculates the vessels main dimensions and stores the results in the Vessel's *mainDimensions* property. The *holtrop* method applies the Holtrop methodology to the vessel, after the main dimensions are calculated. The *shipMotion* method calculates an approximate response for the seakeeping motion of the vessel, also after the main dimensions are calculated. The *systemPrototype* method is responsible for instantiating new systems inside the vessel object created using the Vessel class.

5.3.2.4 The VesselSystem Class

The *VesselSystem* is a superclass from where the *System* classes inherit (Figure 31). It has no default properties, but presents several default methods, which are *add*, *area*, *delete*, *input* and *volume*. The *area* and *volume* methods provide a way to calculate the system objects area and volume, which are important in order to size the vessel. The other three methods are responsible for data handling inside the system objects. The *add* method provides a way of adding subsystem property values once the subsystems are instantiated. The *delete* method removes one instantiated element inside a subsystem object or even a whole subsystem object from the current system object. The *input* method provides a way of changing some already defined subsystem's properties, without the need to reinsert all the properties for a subsystem's element using the add method.

5.3.2.5 The System Classes

The *System* classes are sub classes that inherit from the superclass *VesselSystem* (Figure 31). For this tools library, I defined five *System* classes, aiming to cover the main vessel systems: *CargoSystem*, *OutfittingSystem*, *CrewSystem*, *MachinerySystem* and *ServiceSystem*. These classes inherit their methods from their parent class (*VesselSystem*), which were already defined in the previous section. These classes do not have any default properties, but they are used to store subsystem object instantiated by the user.

The *System* classes are instantiated using the method *systemPrototype* (from the *Vessel* class), which uses the *VesselSystem* class as a constructor. The *systemPrototype* argument is used to specify which elements (and the subsystems these elements belong to) the user wants to instantiate inside each system object.

As it is possible to see in Figure 31, each instance of the *Vessel* class can have up to one instance of each *System* class. It is possible for a design not to have one or more of the *System* classes, once these classes are not applicable for the type of vessel being designed.

5.3.2.6 The Subsystem Classes

The *Subsystem* classes are used to instantiate subsystem objects containing subsystem elements. The number of subsystem classes are quite extensive and each *System* class has its related *Subsystem* classes. The *System* classes and their related *Subsystem* classes can be seen in Figure 31 or in the following list:

CargoSystem**:** *CargoRelatedSpaces, CargoHoldsDryBulk, CargoHoldsLiquidBulk, CargoHoldsContainer, CargoTanksLiquidDryBulk, CargoDecksContainer, CargoDecksRORO and CargoDecksGeneral*;

OutfittingSystem: *OperationalSupport, ShipEquipment and RescueFirefighting*;

ServiceSystem**:** *TechnicalSpacesAccommodation, ShipService and CateringSpaces*;

HotelServices**:** *CrewSystem, Accommodation, CommonSpaces and EmergencyStairways*;

MachinerySystem: *MachinerySpaces, ConsumablesTanks and BallastVoids*.

By using *Subsystem* classes, it is possible to define all sorts of elements that a vessel can contain. These classes contain no default methods, but have their own properties (Figure 31 or APPENDIX B), which need to be filled by the user, providing the structure of the subsystems' elements. *Subsystem* classes are called by the *systemPrototype* method, which create instances of them as subsystem objects, which in turn are stored as *System* objects properties.

As stated in Figure 31, all *System* classes can have any number of *Subsystem* classes elements instances, according to the user’s needs. These instances will be grouped according the subsystems they belong to.

5.3.2.7 Elements

Elements are the elementary instances in the tools library. They are instantiated based on the *Subsystem* classes definitions. From the same *Subsystem* class, it is possible to instantiate as many elements as needed. The knowledge base is used in the instantiation process through the use of the inheritance concept, where the elements inherit the properties from the *Subsystems* classes.

5.4 Application Example: Deck Barge

As a simple example of the research approach application, a simple deck barge conceptual design is going to be developed. From the cargo capacity requirement, the barge’s light and dead weights and main dimensions will be calculated.

5.4.1 Introduction

Barges can be self-propelled or pushed/pulled by a towboat. For this example to be as simple as possible, the barge will be considered as non-self-propelled. It won’t have ballast control capabilities either. This way the machinery, crew and service systems can be disregarded. Deck barges do not have cargo holds. All their cargo (usually carried in the form of containers or general cargo) is kept on the barge weather deck, which will limit the cargo system analysis to only consider the deck cargo system. Outfitting will be considered in order to take into account some parts of the barge, such as mooring decks and rope storage. An example of this kind of barge can be seen in Figure 32.

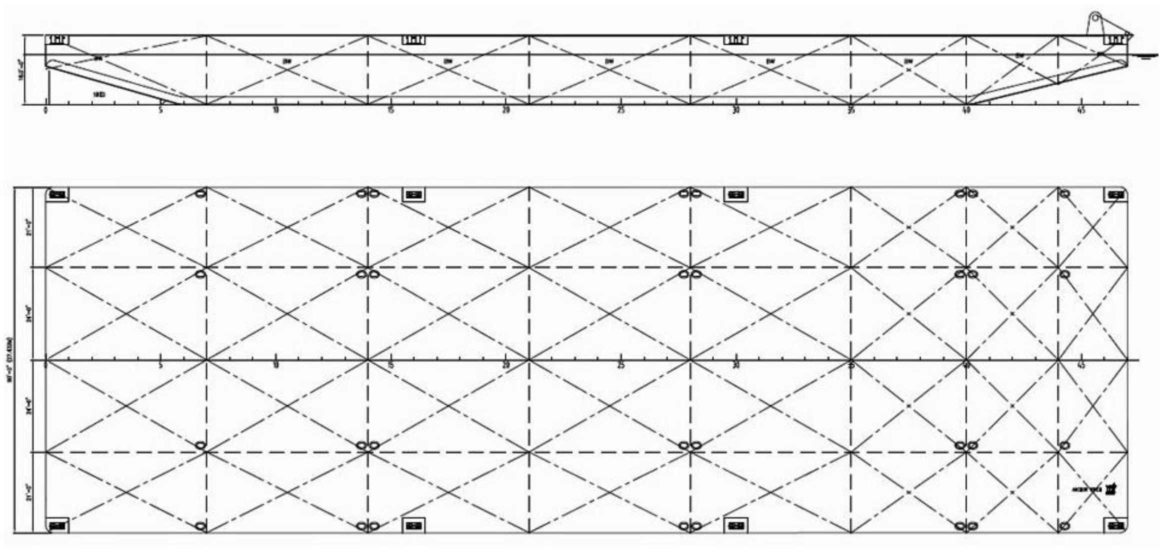


Figure 32 - Deck barge example. (Brokers, 2016)

5.4.2 Tools Library Application

In this example, the deck barge will not have either any kind of machinery or any crew and service facilities. The deck barge will be divided considering only a deck cargo system, an outfitting system and the hull. (Figure 33)

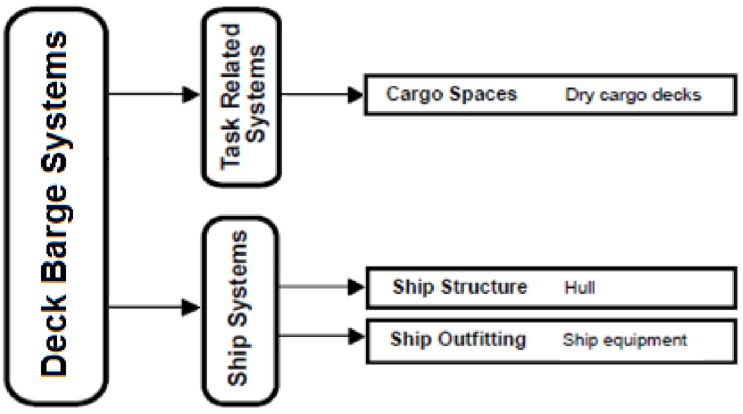


Figure 33 - Deck barge system breakdown.

This system breakdown can be implemented using the tools library based on the class structure presented in the class diagram in Figure 34. This representation was constructed using Figure 31 as a reference and excising all unnecessary classes to represent this simplified example.

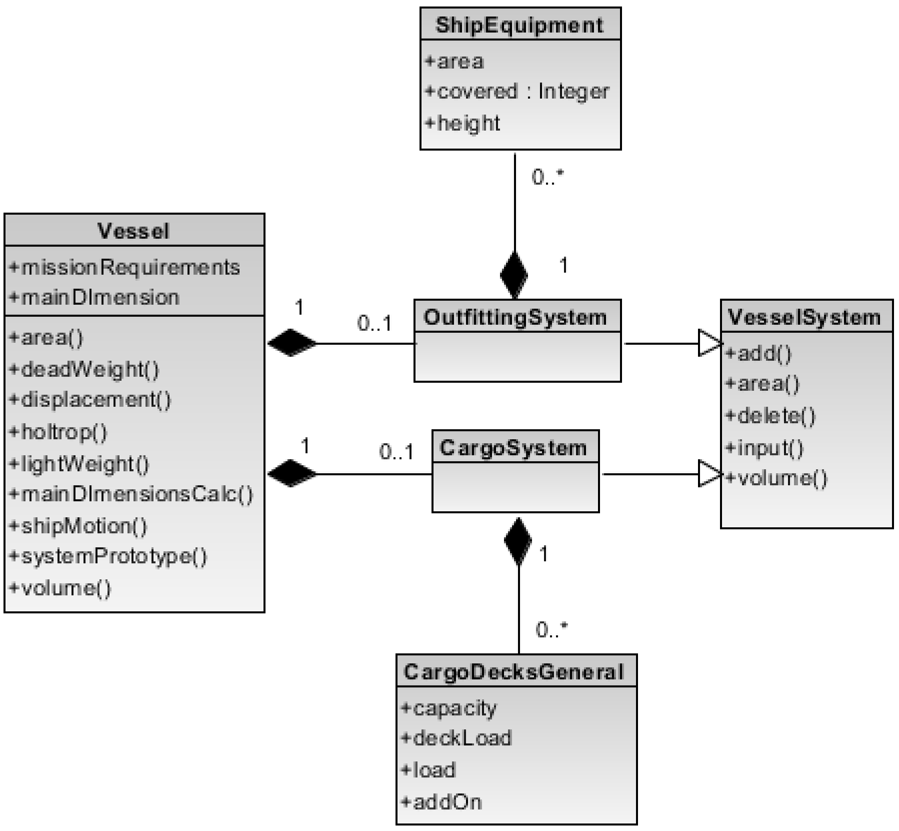


Figure 34 - Class Diagram - Deck Barge.

In order to construct this barge structure, the design process begins by instantiating a new vessel object using the tools library *Vessel* constructor. This vessel object will hold all information about the barge and its systems. The input parameters are vessel type (“deck_barge”), cargo hold capacity (0), cargo deck capacity (4000 ton), crew size (0), vessel speed (0), installed power (0), autonomy (0) and operational area (“inland waterway”):

```
Simple_Barge = new vessel("deck_barge",0,4000,0,0,0,0,"Inland
Waterway");
```

The next step consists in instantiating a cargo system object, to hold cargo subsystem object instances. In order to do so, a vector containing the subsystems to be instantiated (subsystems category-subsystems name pairs) in the cargo system is defined. This vector is used as the input of the *systemPrototype* method, which will define empty subsystems elements according to the specified subsystem categories. For more information about subsystems categories, please check APPENDIX B.

```
var cargo_system = [["cargo_decks_general", "open_cargo_deck"]];

Simple_Barge.cargo_system = new Simple_Barge.systemPrototype(
cargo_system);
```

After the cargo system and its subsystems are specified, it is time to define the subsystems' properties. This can be done by using the *input* method, held by any system instance. The parameters to be input depend on the subsystem categories. For more information about which parameters need to be input for each subsystems categories, please check APPENDIX B. In this case, the expected cargo capacity of the deck was set to 4000 tons, the expected deck load was set to 5 ton/m^2, the actual deck load was set to 4000 tons and an additional margin of 5% was included for safety reasons.

```
Simple_Barge.cargo_system.input([["open_cargo_deck",["capacity", 4000,
"deck_load", 5, "load", 4000, "add_on", 5]]]);
```

For the outfitting system definition, the same process used for the cargo system needs to be repeated. Firstly, a vector containing the subsystem category – subsystem name pairs that will be instantiated is defined. After that, this vector is used as an input for the *systemPrototype* constructor for instantiating the new outfitting system.

```
var outfitting = [["ship_equipment", "mooring_deck_forward"],
                  ["ship_equipment", "mooring_deck_aft"]];

Simple_Barge.outfitting = new Simple_Barge.systemPrototype(
outfitting);
```

The next step will be fulfilling the outfitting subsystems with their properties values. Here, the *input* method was used again to define the subsystem's occupied areas (m^2), covered area (%) and element height (m).

```
Simple_Barge.outfitting.input([["mooring_deck_forward", ["area", 75,
"covered", 100, "height", 1.5]],
                                ["mooring_deck_aft", ["area", 10,
"covered", 100, "height", 1.5]]]);
```

The hull subsystem doesn't need to be instantiated. It will be defined according to the required area and volume for the vessel. Once all the required systems and subsystems are instantiated, the last step in the conceptual design phase for obtaining the vessel's main dimensions is using the *mainDimensionCalc* method, held by the *Vessel* object. This method does not require any input, since all need information is gathered from the tools library database (see APPENDIX C) and from the instantiated systems.

```
Simple_Barge.mainDimensionsCalc();
```

5.4.3 Conceptual Design Result

The results obtained from this design example can be seen on the *mainDimensions* object, held by the V*essel* object (Table 3). Figure 35 presents a simplified model of the designed deck barge, which shows the blocks which compose the design (on red the hull, on blue the main deck and on green the mooring decks).

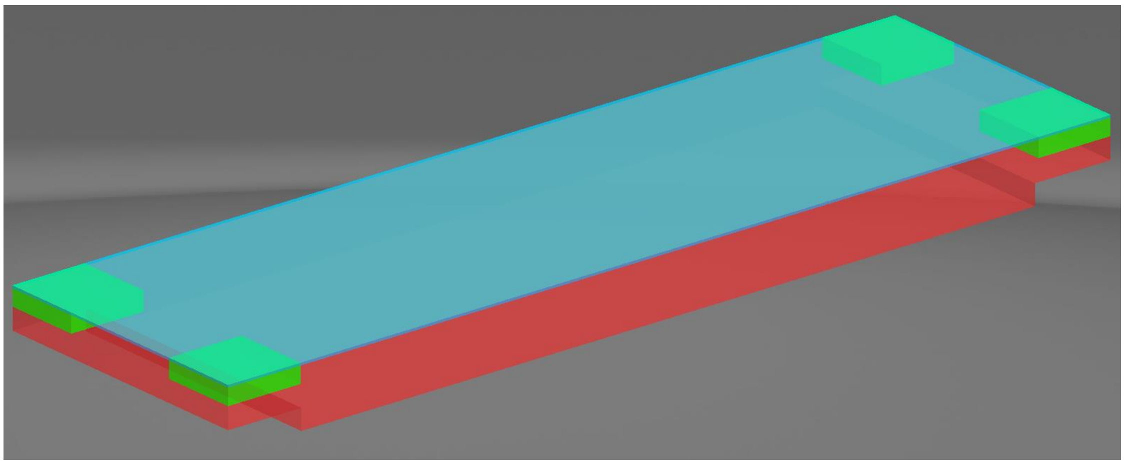


Figure 35 – Deck barge model using blocks.

Table 3 – Conceptual design results – main parameters.

Beam (m)	22.356
Metacentric height above center of buoyancy (m)	14.207
Beam/Depth	4.826
Block coeff	0.838
Center of gravity coeff	0.775
Mid ship coeff	1
Prismatic coeff	0.838
Vertical prismatic coeff	0.838
Waterline coeff	1
Depth (m)	4.632
Froude number	0
Metacentric height above center of gravity (m)	12.483
Vertical center buoyancy(m)	1.865
Vertical center gravity (m)	3.589
Length/Beam	3.339
Length/Depth	16.117
LCB (%)	0.088
Length perpendiculars (m)	72.473
Length waterline (m)	74.647
Draft (m)	3.427
Draft/Depth	0.740
Deadweight (ton)	4000
Displacement (ton)	4770.757
Lightweight (ton)	770.757
Slenderness ratio	4.471

6 Case Study

This case study is used as a benchmark to verify how fair a result is obtained by applying the developed tools library. The idea is to apply the tools library to a real design problem and verify how it behaves and how it can be used to handle a bigger amount of data and requirements.

6.1 Case Description

The case study is based on the PSV NAO FIGHTER (Figure 36). The vessel belongs to the PX121 product family (which is a medium-size class) and was designed by Ulstein Design & Solutions AS, constructed by Ulstein Verft AS and owned by Nordic Amercian Offshore (NAO).

In this case study, the tools library is applied aiming to attend the NAO FIGHTER's mission requirements (Table 4). The results obtained from the tools library application are compared to the real vessel parameters in order to verify how realistic (or unrealistic) the final concept is. Since the result of the tools library is only a preliminary concept, it is not expected to obtain a perfect matchup between the results, but instead, a deviance of about 10% to more or less is expected and considered fine. The library application will be done considering a subsystem division that is common for PSV vessels but can be a little ~~bit~~ different from the NAO FIGHTER's subdivision since its exact subdivision is unknown, but close results are expected anyway.

Table 4 – NAO FIGHTER's mission requirements (Ulstein, 2016)

NAO FIGHTER Requirements	
Tunnel thruster	1
Retractable thruster	1
Azimuth thruster	2
Speed (max)	15.9 kn
Accommodation	24 POB
Deck area	850 m^2
Fuel Oil (MDO)	1474 m^3
Fresh Water	1033 m^3
Ballast water/Drill water	1676 m^3
Liquid mud (sp. gr.2,8 t/m3)	1307 m^3
Brine (sp. gr.2,5 t/m3)	1307 m^3
Cement (4 tanks)	254 m^3
LFL* (4 tanks)	153 m^3
Base oil	259 m^3

Figure 36 - Ulstein's NAO FIGHTER (Ulstein, 2016)

6.2 Tool Library Application

In this case study, the vessel has all the ship systems. In the task related systems, the only one present is the cargo system, since this vessel is a PSV whose only attribution is to transport cargo. The system breakdown structure can be seen in (Figure 37).

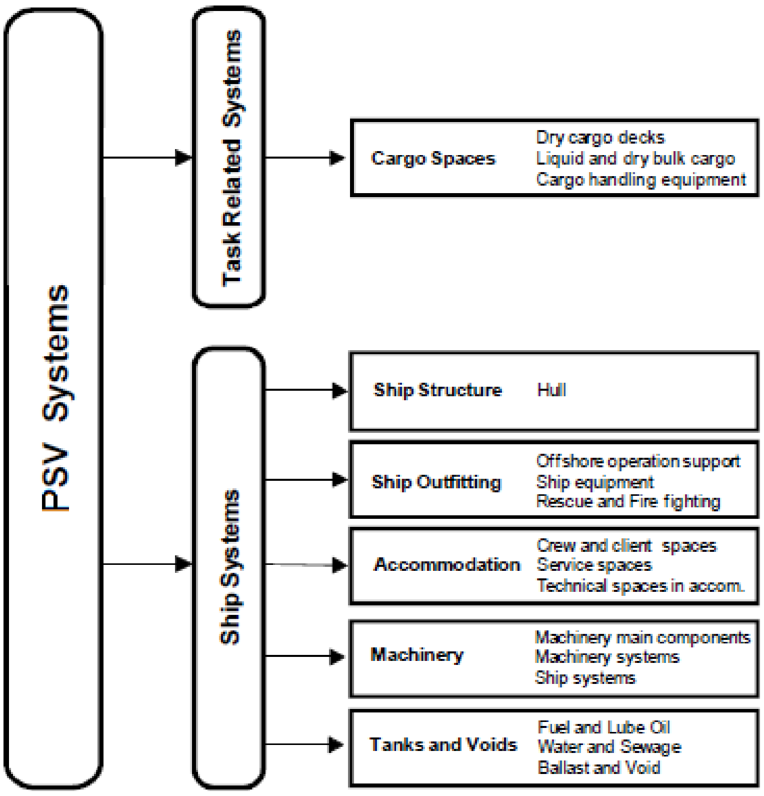


Figure 37 – NAO Fighter case study system breakdown.

This system breakdown can be implemented using the tools library based on the class structure presented in the class diagram in Figure 38. This representation was constructed using Figure

31 as a reference and excising all unnecessary classes to represent this specific case study. The class diagram is not needed in order to apply the tools library, but it is used here to better illustrate the prototype to be developed.

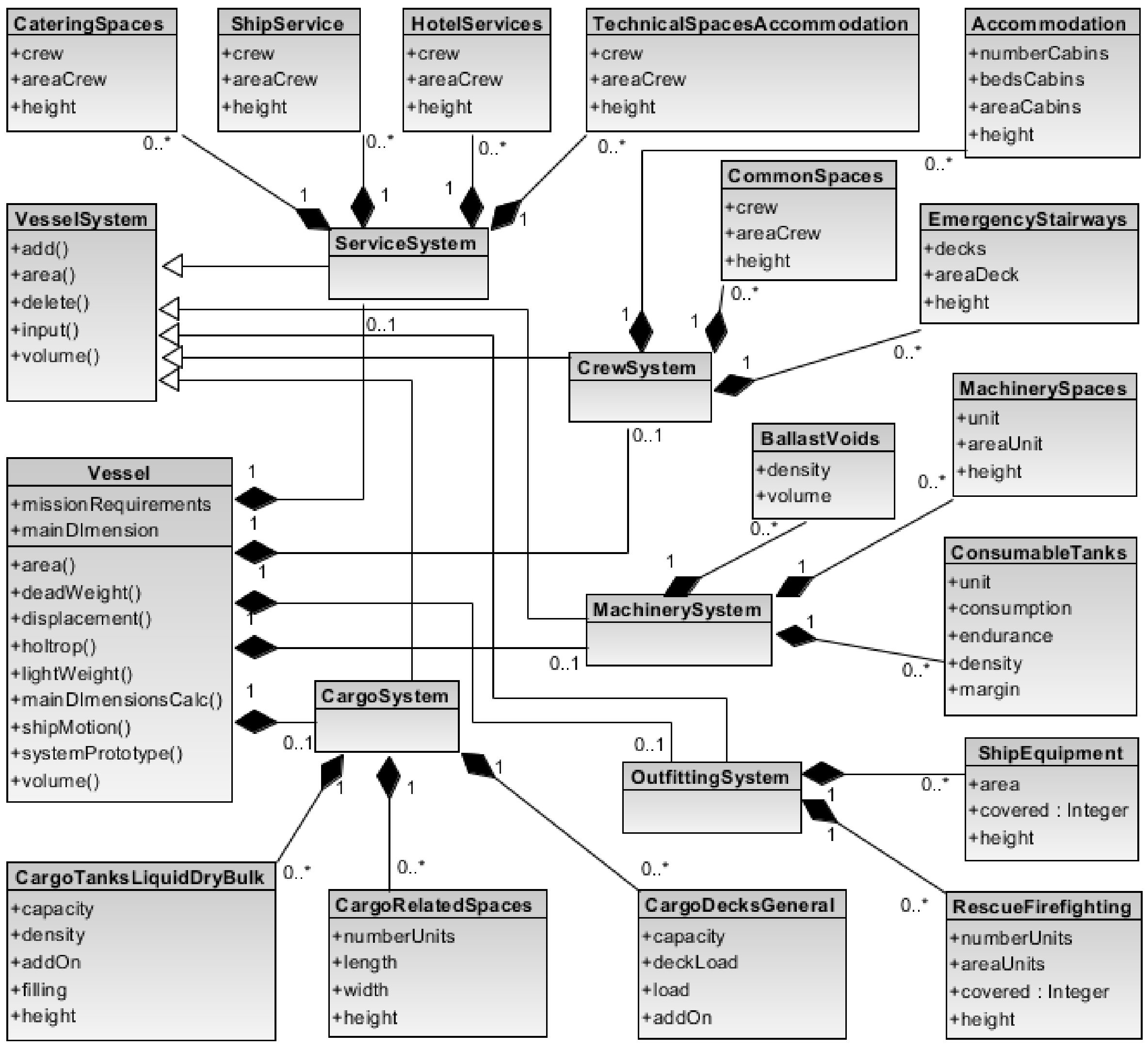


Figure 38 - Class Diagram – NAO Fighter case study.

In order to construct this PSV structure, a new vessel object needs to be instantiated using the tools library *Vessel* constructor. This vessel object will hold all information about the ship and its systems. The input parameters are vessel type ("PSV"), cargo hold capacity (4000 ton), cargo deck capacity (2025 ton), crew size (24), vessel speed (15.85 knots), installed power (6000 kW), autonomy (1000 km) and operational area ("North Sea"):

```
NAO_Fighter = new vessel("PSV",4000,2025,24,15.85,6000,1000,"North
Sea");
```

The next step consists in instantiating a cargo system object, to hold cargo subsystem object instances. In order to do so, a vector containing the subsystems to be instantiated (subsystems category-subsystems name pairs) in the cargo system is defined. This vector is used as the input of the *systemPrototype* method, which will define empty subsystems elements according to the specified subsystem categories. For more information about subsystems categories, please check APPENDIX B.

```
var cargo_system = [
    ["cargo_decks_general", "open_cargo_deck"]
    ["cargo_tanks_liquid_and_dry_bulk","brine_and_mud"],
    ["cargo_tanks_liquid_and_dry_bulk","fresh_water"],
    ["cargo_tanks_liquid_and_dry_bulk","lfl"],
    ["cargo_tanks_liquid_and_dry_bulk","base_oil"],
    ["cargo_tanks_liquid_and_dry_bulk","cement"],
    ["cargo_related_spaces","transfer_pumps_and_piping"]
];

NAO_Fighter.cargo_system = new NAO_Fighter.systemPrototype(
cargo_system);
```

After the cargo system and its subsystems are specified, it is time to define the subsystems properties. This can be done using the *input* method, held by any system instance. The parameters to be input depend on the subsystem categories. For more information about which parameters need to be input for each subsystems categories, please check APPENDIX B. In this case, the expected cargo capacity of the deck was set to 2025 tons, the expected deck load was set to 2.5 ton/m^2, the actual deck load was set to 0 tons and an additional margin of 5% was included for safety reasons. Since the vessel cannot have full deck and tank loads at the same time, the actual load in the deck was set to zero, but the space needed still included in the areas and volumes calculation. The other tanks were set as specified by the requirements and can be seen below.

```
NAO_Fighter.cargo_system.input([
    ["open_cargo_deck",["capacity", 2025, "deck_load", 2.5, "load", 0,
"add_on", 5]],
    ["brine_and_mud",["capacity",1300, "density",1.1, "add_on",5,
"filling",100, "height",3.9]],
    ["fresh_water",["capacity",800,"density", 1, "add_on",5,
"filling",100, "height",3.9]],
    ["lfl",["capacity",150, "density",0.75, "add_on",5, "filling",100,
"height",3.9]],
    ["base_oil",["capacity",260, "density", 0.88, "add_on",5,
"filling",100, "height",3.9]],
    ["cement",["capacity",250, "density",1.2, "add_on",5,
"filling",100, "height",3.9]],
    ["transfer_pumps_and_piping", ["number_ units",3,"length", 3,
"width",2,"height",3.9]]
]);
```

For the outfitting system definition, the same process used for the cargo system needs to be repeated. Firstly, a vector containing the subsystem category – subsystem name pairs that will be instantiated is defined. After that, this vector is used as an input for the *systemPrototype* constructor for instantiating the new outfitting system.

```
var outfitting = [
    ["ship_equipment","tunnel_thrusters"],
    ["ship_equipment","retractable_thrusters"],
    ["ship_equipment","steering_gear"],
    ["ship_equipment", "mooring_deck_forward"],
    ["ship_equipment", "mooring_deck_aft"],
    ["ship_equipment","incinerator_plant"],
    ["ship_equipment","decks_stores"],
    ["ship_equipment","rope_stores"],
    ["rescue_firefighting","fast_rescue_boats"],
    ["rescue_firefighting","life_saving_appliances"],
    ["rescue_firefighting","fire_monitors"]
];

NAO_Fighter.outfitting = new NAO_Fighter.systemPrototype(outfitting);
```

The next step will be fulfilling the outfitting subsystems with their property values. Here, the *input* method was used again to define the subsystem’s occupied areas (m^2), covered area (%), element height (m), number of unit of a certain element and area per unit of this element (m^2).

```
NAO_Fighter.outfitting.input([
    ["tunnel_thrusters",["area",20, "covered",100, "height",6]],
    ["retractable_thrusters",["area",10, "covered",100, "height",9.5]],
    ["steering_gear",  ["area",50, "covered",100, "height",3.9]],
    ["mooring_deck_forward", ["area", 75, "covered", 100, "height",
2.9]],
    ["mooring_deck_aft", ["area", 10, "covered", 100, "height", 3]],
    ["incinerator_plant",["area",10, "covered",100, "height",3]],
    ["decks_stores",["area",40, "covered",100, "height",3]],
    ["rope_stores",["area",40, "covered",100, "height",3]],

["fast_rescue_boats",["number_units",1,"area_unit",25,"covered",100,
"height",4]],
    ["life_saving_appliances",["number_units",40, "area_unit",0.5,
"covered",100, "height",2]],
    ["fire_monitors",["number_units",2, "area_unit",4, "covered",0,
"height",3]]
]);
```

For the crew facilities system definition, the same process used for instantiating new systems needs to be repeated. Firstly, a vector containing the subsystem category – subsystem name pairs that will be instantiated is defined. After that, this vector is used as an input for the *systemPrototype* constructor for instantiating the new crew system.

```
var crew_facilities = [
    ["crew_accommodation","captain_class_suite"],
    ["crew_accommodation","officer_cabin"],
    ["crew_accommodation","crew_single"],
    ["crew_accommodation","crew_double"],
    ["crew_accommodation","cabin_corridors_wall_lining"],
    ["crew_common_spaces","mess_room"],
    ["crew_common_spaces","officers_dayroom"],
    ["crew_common_spaces","crew_dayroom"],
    ["crew_common_spaces","duty_mess"],
    ["crew_common_spaces","gymnesium"],
    ["crew_common_spaces","laundry_linen"],
    ["crew_common_spaces","change_room"],
    ["crew_common_spaces","toilets"],
    ["crew_common_spaces","corridors"],
    ["crew_emergency_stairways","main_stair"],
    ["crew_emergency_stairways","service_stairs_fore"],
    ["crew_emergency_stairways","service_stairs_aft"]
];

NAO_Fighter.crew_facilities = new
NAO_Fighter.systemPrototype(crew_facili ties);
```

The next step will be fulfilling the crew facilities subsystems with their properties values. Here, the *input* method was used again to define the subsystem's number of cabins of each type, number of bed per cabin of each type, area occupied by each cabin (m^2), height of each cabin (m), number of crew member to use certain facility, area required per crew member (m^2), height of facility (m), number of decks occupied by a certain installation, area required per deck (m^2) and height of each deck (m).

```
NAO_Fighter.crew_facilities.input([
    ["captain_class_suite",["number_cabins",2, "beds_cabins",1,
"area_cabins",30, "height",2.9]],
    ["officer_cabin",["number_cabins",4, "beds_cabins",1,
"area_cabins",20, "height",2.9]],
    ["crew_single",["number_cabins",8, "beds_cabins",1,
"area_cabins",15, "height",2.9]],
    ["crew_double",["number_cabins",5, "beds_cabins",2,
"area_cabins",15, "height",2.9]],
    ["cabin_corridors_wall_lining",["number_cabins",1, "beds_cabins",0,
"area_cabins",70, "height",2.9]],
    ["mess_room",["crew",20, "area_crew",3, "height",2.9]],
    ["officers_dayroom",["crew",10, "area_crew",3, "height",2.9]],
    ["crew_dayroom",["crew",20, "area_crew",3, "height",2.9]],
    ["duty_mess",["crew",10, "area_crew",2.5, "height",2]],
    ["gymnesium",["crew",24, "area_crew",1, "height",3]],
    ["laundry_linen",["crew",24, "area_crew",0.3, "height",2.9]],
    ["change_room",["crew",24, "area_crew",0.5, "height",3]],
    ["toilets",["crew",24, "area_crew",0.3, "height",3]],
    ["corridors",["crew",24, "area_crew",1, "height",3]],
    ["main_stair",["decks",7, "area_deck",10, "height",2.9]],
    ["service_stairs_fore",["decks",3, "area_deck",6, "height",3]],
    ["service_stairs_aft",["decks",3, "area_deck",6, "height",3]]
]);
```

For the service facilities system definition, the same process used for instantiating new systems needs to be repeated. Firstly, a vector containing the subsystem category – subsystem name pairs that will be instantiated is defined. After that, this vector is used as an input for the *systemPrototype* constructor for instantiating the new service system.

```
var service_facilities = [
    ["ship_service","wheelhouse"],
    ["ship_service","ship_offices"],
    ["ship_service","iscp_office"],
    ["ship_service","conference_room"],
    ["ship_service","hospital"],
    ["catering_spaces","galleys"],
    ["catering_spaces","galley_provision_store"],
    ["catering_spaces","dry_provision_store"],
    ["catering_spaces","cold_provision_store"],
    ["catering_spaces","scullery"],
    ["hotel_services","linen_store"],
    ["hotel_services","ship_laundry"],
    ["hotel_services","storage_spaces_in_the_accommodation"],
    ["hotel_services","cleaning_lockers"],
    ["technical_spaces_accommodation","ac_rooms_and_ducting"],
    ["technical_spaces_accommodation","electric_substations"],

["technical_spaces_accommodation","instrument_room_under_wheelhouse"],
    ["technical_spaces_accommodation","void_spaces_in_deckhouse"]
];
```

```
NAO_Fighter.service_facilities = new
NAO_Fighter.systemPrototype(service _facilities);
```

The next step will be fulfilling the service facilities subsystems with their property values. Here, the *input* method was used again to define the subsystem's number of crew members to use a certain facility, area required per crew member (m^2), height of facility (m).

```
NAO_Fighter.service_facilities.input([
    ["wheelhouse",["crew",24, "area_crew", 8, "height", 3.1]],
    ["ship_offices",["crew",24, "area_crew", 1.5, "height", 2.9]],
    ["iscp_office",["crew",24, "area_crew", 0.2, "height", 2.9]],
    ["conference_room",["crew",24, "area_crew", 1, "height", 2.9]],
    ["hospital",["crew",24, "area_crew", 0.6, "height", 2.9]],
    ["galleys",["crew",24, "area_crew", 0.8, "height", 2.9]],
    ["galley_provision_store",["crew",24, "area_crew", 0.2, "height",
2.9]],
    ["dry_provision_store",["crew",24, "area_crew", 2, "height", 3]],
    ["cold_provision_store",["crew",24, "area_crew", 1, "height", 3]],
    ["scullery",["crew",24, "area_crew", 0.2, "height", 2.9]],
    ["linen_store",["crew",24, "area_crew", 0.3, "height", 3]],
    ["ship_laundry",["crew",24, "area_crew", 0.6, "height", 3]],
    ["storage_spaces_in_the_accommodation",["crew",24, "area_crew",
1.5, "height", 3]],
    ["cleaning_lockers",["crew",24, "area_crew", 0.5, "height", 3]],
    ["ac_rooms_and_ducting",["crew",24, "area_crew", 2, "height", 2]],
    ["electric_substations",["crew",24, "area_crew", 0.2, "height",
2.9]],
    ["instrument_room_under_wheelhouse",["crew",24, "area_crew", 4,
"height", 2]],
    ["void_spaces_in_deckhouse",["crew",24, "area_crew", 2, "height",
2]]
]);
```

For the machinery system definition, the same process used for instantiating new systems needs to be repeated. Firstly, a vector containing the subsystem category – subsystem name pairs that will be instantiated is defined. After that, this vector is used as an input for the *systemPrototype* constructor for instantiating the new machinery system.

```
var machinery = [
    ["machinery_spaces","main_and_auxiliary_engine_rooms"],
    ["machinery_spaces","shaftlines_propellers_propulsion_thrusters"],
    ["machinery_spaces","emergency_generator_and_battery_room"],
    ["machinery_spaces","pump_rooms_and_equipment_spaces"],
    ["machinery_spaces","workshops_and_stores"],
    ["machinery_spaces","ecr_and_switchboard_room"],
    ["machinery_spaces","fire_fighting_system_and_co2_room"],
    ["machinery_spaces","engine_casing"],
    ["machinery_spaces","air_intakes"],
    ["machinery_spaces","funnel"],
    ["consumables_tanks","fuel_oil"],
    ["consumables_tanks","lub_oil"],
    ["consumables_tanks","fresh_water"],
    ["consumables_tanks","sewage_and_grey_water"],
    ["ballast_and_voids","ballast_water"]
];

NAO_Fighter.machinery = new NAO_Fighter.systemPrototype(machinery);
```

The next step will be fulfilling the machinery subsystems with their property values. The installed power defined when creating the vessel object, was a rough guess. This value can be changed by the tools library in the next steps of the design process, but for now it is used to define the machinery system's subsystems properties. The value is stored in a variable just to reduce the code to be typed. The *input* method was used again to define the subsystem's number of units used by a certain element (these units can be kW or number of decks), the area per unit occupied by this element (m^2), the height (m), for tanks of consumables, the reference unit (installed power or number of hours per day used), the consumption, the endurance (days), the density (ton/m^3) and a safety margin and, for tanks of non-consumables, the density (ton/m^3) and volume (m^3). Note that the ballast water tanks were specified with density zero, to account for the volume but without considering the weight of the full tank, since the vessel is being defined in a loaded condition were the ballast tanks do not need to be filled.

```
var installedPower = NAO_Fighter.missionRequirements.installedPower;
NAO_Fighter.machinery.input([
    ["main_and_auxiliary_engine_rooms",["unit",installedPower,
"area_unit",0.035, "height",4.3]],

["shaftlines_propellers_propulsion_thrusters",["unit",installedPower,
"area_unit",0.01, "height",4.1]],
    ["emergency_generator_and_battery_room",["unit",installedPower,
"area_unit",0.02, "height",2.9]],
    ["pump_rooms_and_equipment_spaces",["unit",installedPower,
"area_unit",0.003, "height",3.9]],
    ["workshops_and_stores",["unit",installedPower, "area_unit",0.003,
"height",3.9]],
    ["ecr_and_switchboard_room",["unit",installedPower,
"area_unit",0.003, "height",3.9]],
    ["fire_fighting_system_and_co2_room",["unit",installedPower,
"area_unit",0.002, "height",3.45]],
    ["engine_casing",["unit",6, "area_unit",40, "height",3]],
    ["air_intakes",["unit",3, "area_unit",15, "height",3]],
    ["funnel",["unit",1, "area_unit",8, "height",5]],
    ["fuel_oil",["unit",installedPower, "consumption",0.0002,
"endurance",2, "density",0.95, "margin",20]],
    ["lub_oil",["unit",installedPower, "consumption",0.000001,
"endurance",2, "density",0.89, "margin",20]],
    ["fresh_water",["unit",24, "consumption",0.0085, "endurance",2,
"density",1, "margin",20]],
    ["sewage_and_grey_water",["unit",24, "consumption",0.0085,
"endurance",2, "density",1, "margin",20]],
    ["ballast_water",["density",0, "volume",2000]]
]);
```

The hull subsystem does not need to be instantiated. It will be defined according to the required area and volume for the vessel. Once all the required systems and subsystems are instantiated, the last step in the conceptual design phase for obtaining the vessel's main dimensions is using the *mainDimensionCalc* method, held by the *Vessel* object. This method does not require any input, since all needed information is gathered from the tools library database (see APPENDIX C) and from the instantiated systems.

```
NAO_Fighter.mainDimensionsCalc();
```

With the application of the *mainDimensionCalc* method, a new estimate was made for the installed power, and the initial guess was substituted in the mission requirements. With the new value, it is important to define again the properties of the machinery system and afterwards that reapply the *mainDimensionCalc* method. This is important since the installed power is used to define a lot of properties in the machinery system.

```
var installedPower = NAO_Fighter.missionRequirements.installedPower;
NAO_Fighter.machinery.input([
    ["main_and_auxiliary_engine_rooms",["unit",installedPower,
"area_unit",0.035, "height",4.3]],

["shaftlines_propellers_propulsion_thrusters",["unit",installedPower,
"area_unit",0.01, "height",4.1]],
    ["emergency_generator_and_battery_room",["unit",installedPower,
"area_unit",0.02, "height",2.9]],
    ["pump_rooms_and_equipment_spaces",["unit",installedPower,
"area_unit",0.003, "height",3.9]],
    ["workshops_and_stores",["unit",installedPower, "area_unit",0.003,
"height",3.9]],
    ["ecr_and_switchboard_room",["unit",installedPower,
"area_unit",0.003, "height",3.9]],
    ["fire_fighting_system_and_co2_room",["unit",installedPower,
"area_unit",0.002, "height",3.45]],
    ["engine_casing",["unit",6, "area_unit",40, "height",3]],
    ["air_intakes",["unit",3, "area_unit",15, "height",3]],
    ["funnel",["unit",1, "area_unit",8, "height",5]],
    ["fuel_oil",["unit",installedPower, "consumption",0.0002,
"endurance",2, "density",0.95, "margin",20]],
    ["lub_oil",["unit",installedPower, "consumption",0.000001,
"endurance",2, "density",0.89, "margin",20]],
    ["fresh_water",["unit",24, "consumption",0.0085, "endurance",2,
"density",1, "margin",20]],
    ["sewage_and_grey_water",["unit",24, "consumption",0.0085,
"endurance",2, "density",1, "margin",20]],
    ["ballast_water",["density",0, "volume",2000]]
]);

NAO_Fighter.mainDimensionsCalc();
```

6.3 Results and Analyses

The results obtained from this design example can be seen on the *mainDimensions* object, held by the V*essel* object (Table 5). The obtained results do not show anything that raises any worry about the feasibility of the design. All the parameters are quite normal for a PSV of this size.

Table 5 – Conceptual design results – main parameters.

Beam (m)	19.39
Metacentric height above center of buoyancy (m)	6.52
Beam/Depth	2.44
Block coeff.	0.675
Center of gravity coeff.	0.74
Mid ship coeff.	0.985
Prismatic coeff.	0.685
Vertical prismatic coeff.	0.807
Waterline coeff.	0.836
Depth (m)	7.94
Froude number	0.28
Metacentric height above center of gravity (m)	1.19
Vertical center buoyancy(m)	2.73
Vertical center gravity (m)	8.06
Length/Beam	4.48
Length/Depth	10.93
LCB (%)	-0.022
Length perpendiculars (m)	84.29
Length waterline (m)	86.82
Draft (m)	4.93
Draft/Depth	0.62
Deadweight (ton)	2997.06
Displacement (ton)	5568.23
Installed Power (kW)	7789.15
Lightweight (ton)	2571.17
Slenderness ratio	4.94

It is also possible to apply the *shipMotion* method to obtain an estimative of the vessel response for a specific sea state. In the case below, the method evaluates the vessel response (in its center of gravity) for a sea state of wave in beam sea, with 2 m amplitude and natural period of 6.5 s.

```
NAO_Fighter.shipMotion(NAO_Fighter.mainDimensions.Lwl,NAO_Fighter.mainD
imensions.B,NAO_Fighter.mainDimensions.Cb,NAO_Fighter.mainDimensions.T,
NAO_Fighter.missionRequirements.speed,90,0,2,NAO_Fighter.mainDimensions
.Cwl, NAO_Fighter.mainDimensions.GM,6.5,20,0.6,'main_graphic');
```

Figure 39 shows an example of movement plot from the *shipMotion* method.

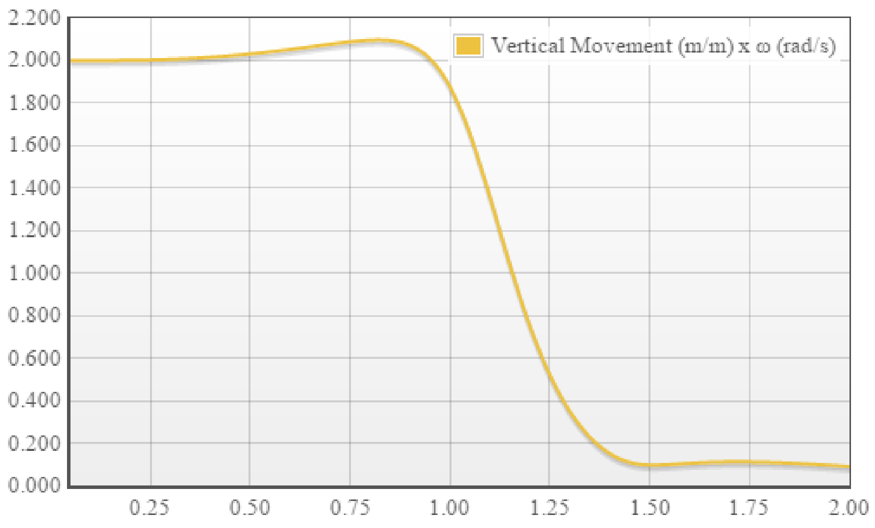


Figure 39 - Vertical motion (m/m) as function of wave frequency. Combined movement from the pitch and heave at the desired location.

Figure 40 shows an example of acceleration plot from the *shipMotion* method.

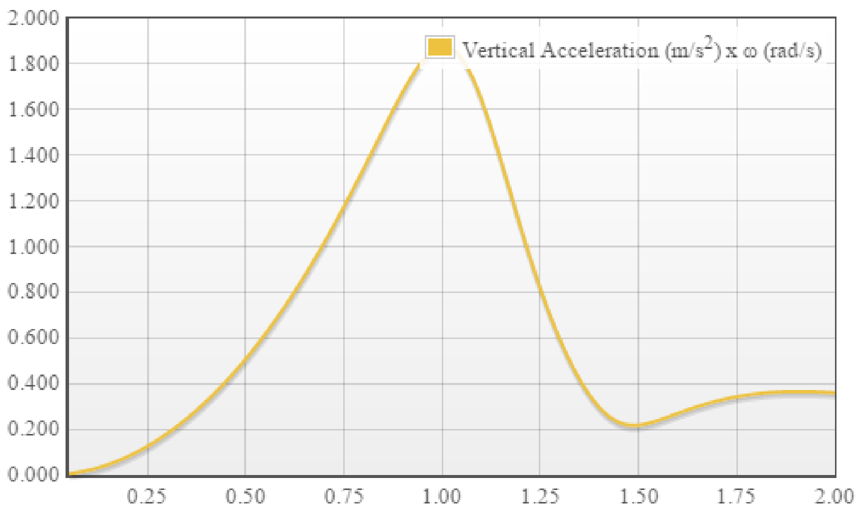


Figure 40 - Vertical acceleration (m/s2) as function of wave frequency. Combined movement from the pitch and heave at the desired location.

Other method implemented in the library and that can be applied in this case is the *holtrop* method. It can give a rough guess about the resistance the vessel would experience while cruising in a specific speed. The details about this method application can be found in APPENDIX A.

```
NAO_Fighter.holtrop(NAO_Fighter.mainDimensions.Lwl,NAO_Fighter.mainDime
nsions.B,NAO_Fighter.mainDimensions.T,NAO_Fighter.mainDimensions.T,NAO_
Fighter.mainDimensions.Lcb,NAO_Fighter.mainDimensions.Cb,NAO_Fighter.ma
inDimensions.Cm,NAO_Fighter.mainDimensions.Cwl,NAO_Fighter.missionRequi
rements.speed,1,1,3,[0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0],'ma
in_graphic2');
```

Figure 41 shows the expected curve of resistance for the vessel for different Froude numbers (and consequently, different speeds), although it is possible to also obtain the resistance value for a specific speed.

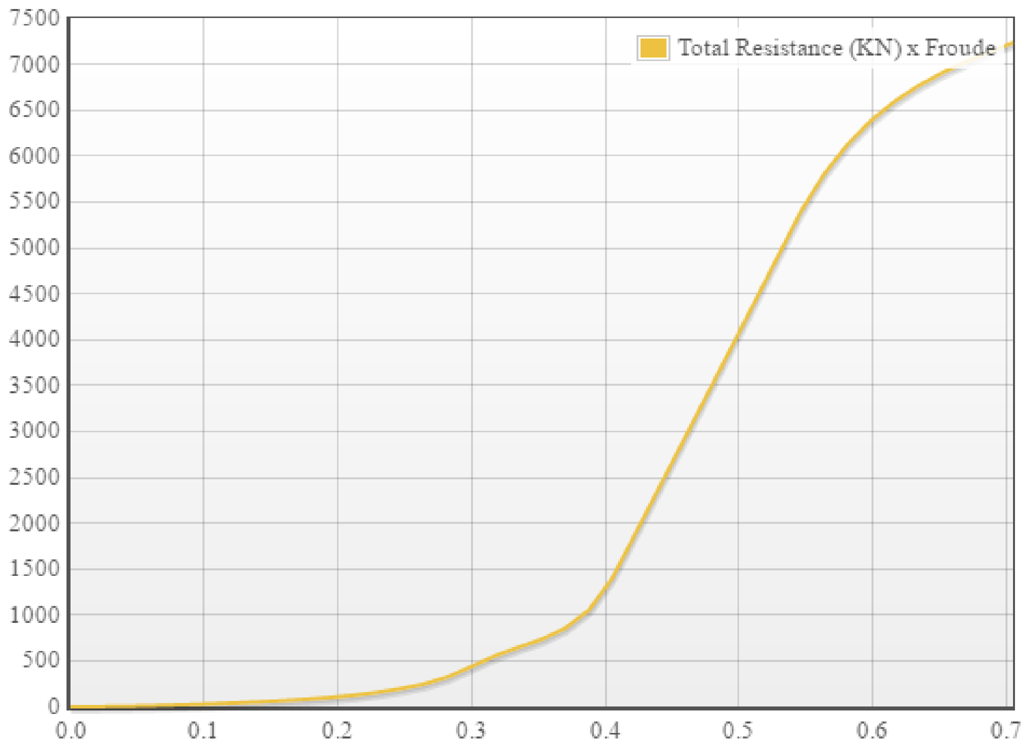


Figure 41 – Total resistance (kN) in function of Froude Number.

For the special case of OSV, a method which can draw the hull lines of a X-BOW inspired vessel was developed. This method is not fully implemented in the library, but one example of its application in this case study can be seen in Figure 42 and Figure 43 .

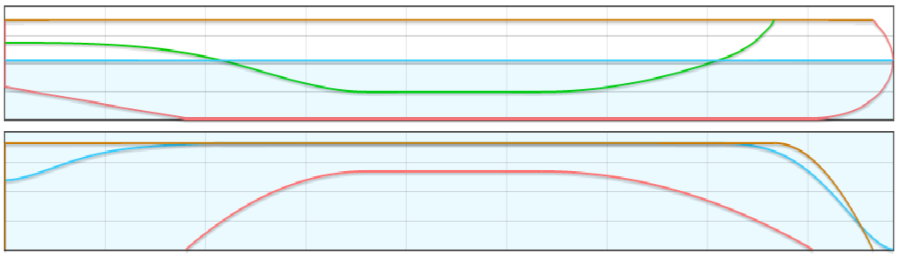


Figure 42 – Lateral and Top views of the case study's hull lines.

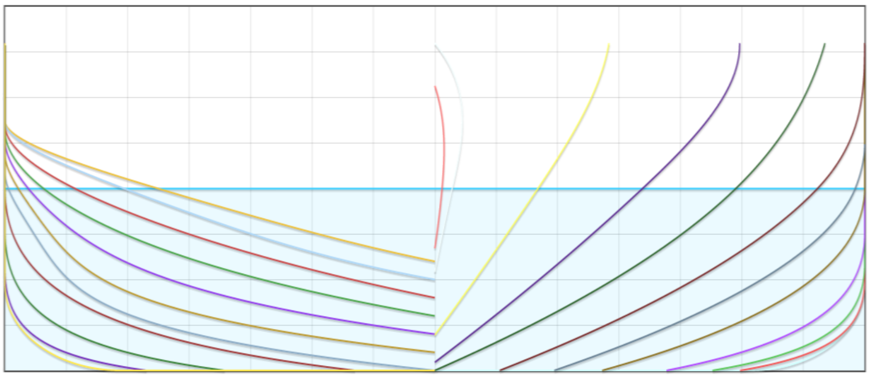


Figure 43 – Cross sectional curves for the case study vessel.

The results obtained from the application of the tools library can be compared with the original vessel's parameters. Table 6 presents these figures and also the deviance of the obtained values from the original design.

Table 6 – Comparison between case study and NAO FIGHTER parameters.

	Case Study	NAO FIGHTER	Deviation
Length	84.20 m	83.40 m	1%
Beam	19.30 m	18.00 m	7%
Dead weight	2997 tonnes	3300 tonnes	-9%
Draught (max)	6.53 m	6.00 m	9%
Speed (max)	15.85 kn	15.85 kn	0%
Accommodation	24 POB	24 POB	0%
Deck area	850 m^2	850 m^2	0%

Some parameters have no deviance at all, since they were set as required values, from which the design was built around. For the other main parameters, it is possible to verify cases where the deviation goes almost 10% up and almost 10% down.

Luckily enough, the obtained values are all inside the required range of ± 10%, with the only iteration being made to adjust the installed power value. If the obtained results were not as close to the real vessel as these ones, it would not necessarily indicate a failed design. As long as none of the parameters makes the design unfeasible for any reason, it can be considered as a new design solution.

7 Concluding Remarks

After applying the open source conceptual ship design tools library in a real design problem, I concluded that the tools library can be successfully used to handle the conceptual design phase of a vessel. The quality of the results is strongly connected to the designer's knowledge about the vessel structure, subdivision and elements. The library also relays on an extensive database that is constructed using information from previous vessels. The quality of this database also has the potential to greatly impact the final concept, so it should be kept updated and organized in order to ensure the most trustable results.

It is possible to argue that the tools library is not only a collection of ship design function as it was expected in the beginning of this study. This is due to the fact that the design decisions taken during the tools library development have defined a stiff structure for the vessel object, which needs to be followed by the user in order to correctly apply the design functions. This stiff structure makes the tools library be more similar to a framework than to a conventional library of functions.

While doing the problem statement, I have defined 3 main resource questions to investigate through this study. They were:

- **Question 1**: How can the vessel be efficiently subdivided in order to allow better design control and knowledge handle in the initial stages of the design process?
- **Question 2:** Which aspects of open source technology can improve the conceptual ship design process?
- **Question 3:** How to develop a tools library that has a structure flexible enough to contemplate the design of any kind of vessel but at the same time is capable of providing results that are good enough to compose a solid ground for the continuation of the design process?

Regarding the first research question, the use of OOP and the concept of classes to represent the vessel's systems and subsystems proved to be an efficient way of handling the conceptual design knowledge. Having the vessel divided in subsystems and elements, which can be attributed to classes in a OOP language, works very well to approach a complex design problem by dividing it in smaller and simpler problems, that can be solved individually.

At the moment, the biggest drawback of the tools library is that it applies the SyBSD theory directly, which does not provide a mechanism to consider the systems, subsystems and elements interdependencies and interfaces. In a product with such a complex structure as a vessel, neglecting this type of relation will, inevitably, lead to design problems, especially in the more advanced design stages. Since these effects are less relevant in the conceptual design phase, the tools library can be applied to the conceptual design but should not be applied, in its current development stage, to further design phases.

When evaluating the second research question, I have noticed that the freedom and power provided by the open source technology make the tools library possibilities almost endless. Since the tools library is developed in an OSS, it is free and, consequently, more accessible to end users, which can be constrained by high costs related to licensing fees of commercial software packages. Since the code is open, the tools library can be continuously improved by interested users. There is no limitation to its functionality, since when a limitation is found, it can be corrected or extinguished by the adding of new tools.

One of the strongest points in the conceptual ship design tools library is its modularity. As presented in Chapter 5, the library in composed by several design tools organized under the *Vessel* object's methods. As the need for different functionalities appears, new tools can be added to the library without the need to modify the functions that are already there. The new functions just need to be developed having in mind the *Vessel* object's hierarchy and structure. New systems and subsystem structures can also be easily added by just updating the SyBSD Structure Database. The modular characteristic makes the tools library a very flexible and powerful ship design tool.

The choice for JavaScript as an OSS was not aleatory. At first, it is a web-based programming language which is present in all the web browsers. It is easy and intuitive to code, what collaborates with its fast development and spread. Developing the library to be accessed via web browsers reduces the needs for client side software to a minimum, also reducing the computational requirements for the user's hardware. When developed for web, the tools library became accessible for any operational system and device, which also contributes to increase the application reach. The web can also be used as a platform for cooperative development, where design teams can store, share and discuss the design task. Having a tools library that support natively the internet as a platform makes the development of this collaborative design environment easier.

Finally, while investigating the third research question, the way I found to make the tools library the most generic as possible was by SyBSD theory to simplify the systems, subsystems and elements definitions as much as possible. Having design elements being represented by a handful set of parameters, considering only which is highly relevant, makes these definitions as simple and generic as possible. That way, the same element structure can be reused for different elements, which also works well with the OOP philosophy. When using a simplified definition for systems, subsystems and elements, the generic design tools library just need a well-structured knowledge base (more specifically the SyBSD Structure database) to handle the design task without any complication.

7.1 Future Work

The tools library is suitable enough for addressing the conceptual design phase, but lacks support for further and more detailed stages of design. The capacity to store and handle data can be increased in order to make detail design a reality, especially considering the systems communication and interfaces, implementing a mix of SyBSD and Holistic design processes. This improvement in the library can be difficult to implement, since its whole structure is based on isolating systems, subsystems and elements, but definitely worth investigating in a further study.

One of the greatest points of the OSS is its fantastic community, which can work together in the direction of a common goal, by helping improving the original code, suggesting and implementing changes and new features, finding and correcting bugs and so on. In this work, this aspect of the OSS was not investigated and can be a field for future study in order to further investigate how the OSS technology can impact the ship design task.

The library offers a great set of ship design functions, which can be further expanded, but misses a native support for a graphic user interface (GUI). Right now, a user needs to hardcode a GUI and link it with the vessel object and its properties and methods. Having a GUI library which has this integration already done and giving the user the opportunity to customize it if he or she wants, could make it more appealing for people who are not very experienced with JavaScript or with coding at all.

8 References

ANDERSON, D. M. 1997. *Agile product development for mass customization : how to develop and deliver products for mass customization, niche markets, JIT, build-to-order, and flexible manufacturing,* Chicago, Irwin Professional Pub.

BERNSTEIN, J. I. 1998. *Design methods in the aerospace industry : looking for evidence of set-based practices.* M s, Massachusetts Institute of Technology, Technology and Policy Program.

BONACCORSI, A. & ROSSI, C. 2003. Why Open Source software can succeed. *Research Policy,* 32**,** 1243-1258.

BOOCH, G. 1982. Object-oriented design. *ACM SIGAda Ada Letters,* I**,** 64-76.

BOOCH, G., RUMBAUGH, J. & JACOBSON, I. 1998. *Unified Modeling Language User Guide, The,* Addison Wesley

BRETTHAUER, D. 2001. Open Source Software: A History. *UConn Libraries Published Works*.

BRINATI, H. L., AUGUSTO, O. B. & DE CONTI, M. B. 2007. Learning Aspects of Procedures for Ship Conceptual Design Based on First Principles. *International Conference on Engineering Education*.

BROKERS, D. 2016. *86 m (282-feet) Ballastable Deck Barge* [Online]. Available: http://www.workbargebrokers.com/product/1275/130503-bd [Accessed 1 June 2016].

CHO, K. N. & ŽANIĆ, V. Design principles and criteria. 16th International Ship and Offshore Structures Congress, 2006.

CHOU, Y.-C. 2004. Applying Neural Networks in Quality Function Deployment Process for Conceptual Design. *Journal of the Chinese Institute of Industrial Engineers,* 21**,** 587-596.

COYNE, R. D. D., ROSENMAN, M. A., RADFORD, A. D., BALACHANDRAN, M. & GERO, J. S. 1989. *Knowledge-Based Design Systems,* Addison-Wesley Longman Publishing Co., Inc.

CROCKFORD, D. 2008. *JavaScript: The Good Parts,* " O'Reilly Media, Inc.".

DIEROLF, D. A. & RICHTER, K. J. 1989. Computer-Aided Group Problem Solving for Unified Life Cycle Engineering (ULCE). Alexandria: Institute for Defense Analyses.

ERIKSTAD, S. O. 1996. *A decision support model for preliminary ship design.* PhD Thesis, NTNU.

ERIKSTAD, S. O. & LEVANDER, K. System Based Design of offshore support vessels. Proceedings 11th International Marine Design Conference—IMDC201, 2012.

EVANS, J. H. 1959. Basic Design Concepts. *Journal of the American Society for Naval Engineers,* 71**,** 671-678.

FISHER, R. A. 1971. The design of experiments.

FLANAGAN, D. 2011. *JavaScript: The definitive guide*, " O'Reilly Media, Inc.".

GACEK, C. & ARIEF, B. 2004. The many meanings of open source. *IEEE Software,* 21**,** 34-40.

GALE, P. A. & SLUTSKY, J. 2014. Ship design. Available: http://www.accessscience.com/content/ship-design/619500.

GASPAR, H. M. 2013. *Handling Aspects of Complexity in Conceptual Ship Design.* PhD Thesis, NTNU.

GASPAR, H. M., RHODES, D., ROSS, A. & ERIKSTAD, S. O. 2012a. Handling complexity aspects in conceptual ship design. *International Maritime Design Conference.* Glasgow, UK.

GASPAR, H. M., RHODES, D. H., ROSS, A. M. & ERIKSTAD, S. O. 2012b. Addressing Complexity Aspects in Conceptual Ship Design: A Systems Engineering Approach. *Journal of Ship Production and Design,* 28**,** 145-159.

GERO, J. S. 1990. Design prototypes: a knowledge representation schema for design. *AI magazine,* 11**,** 26.

GERO, J. S. & ROSENMAN, M. A. 1990. A conceptual framework for knowledge-based design research at Sydney University's design computing unit. *Artificial Intelligence in Engineering,* 5**,** 65-77.

HOLTROP, J. 1984. A Statistical Reanalysis of Resistance and Propulsion Data. *International Shipbuilding Progress,* Vol 31.

HOLTROP, J. & MENNEN, G. 1982. An approximate power prediction method. *International Shipbuilding Progress,* Vol 29.

HUBKA, V. & EDER, W. 1988. *Theory of Technical Systems,* New York, Spring.

HYDRONSHIP. 2016. *FREE!ship* [Online]. Available: http://www.hydronship.net [Accessed April 2016].

JENSEN, J. J., MANSOUR, A. E. & OLSEN, A. S. 2004. Estimation of ship motions using closed-form expressions. *Ocean Engineering,* 31**,** 61-85.

KALYANARAM, G. & KRISHNAN, V. 1997. Deliberate Product Definition: Customizing the Product Definition Process. *Journal of Marketing Research,* 34**,** 276.

KRISHNAN, V., EPPINGER, S. D. & WHITNEY, D. E. 1991. *Towards a cooperative design methodology : analysis of sequential decision strategies,* Cambridge, Mass., Sloan School of Management, Massachusetts Institute of Technology.

KUEI, C. 1999. Categories of Free and Non-Free Software in Open Sources. *In:* DIBONA, C., OCKMAN, S. & STONE, M. (eds.) *Open sources : voices from the open source revolution.* 1st ed. Beijing ; Sebastopol: O'Reilly.

LAKHANI, K., WOLF, B., BATES, J. & DIBONA, C. 2002. The Boston Consulting Group Hacker Survey.

LEE, K.-H., LEE, D. & HAN, S.-H. 1996. Object-oriented approach to a knowledge-based structural design system. *Expert Systems with Applications,* 10**,** 223-231.

LEVANDER, K. 1991. System-based passenger ship design. *4th Int. Marine Systems Design Conference (IMSDC'91).* Kobe.

LEVANDER, K. 2012. *System Based Ship Design Kompendium*.

LIKER, J. K., SOBEK, D. K., WARD, A. C. & CRISTIANO, J. J. 1996. Involving suppliers in product development in the United States and Japan: evidence for set-based concurrent engineering. *IEEE Transactions on Engineering Management,* 43**,** 165-178.

MANCHINU, A. & MCCONNELL, F. The SFI coding and classification system for ship information. REAPS Technical Symposium, 1977.

MEYER, B. 1988. *Object-oriented software construction,* Prentice hall New York.

MOCKUS, A., FIELDING, R. T. & HERBSLEB, J. 2000. A case study of open source software development: the Apache server. 263-272.

MONTEIRO, T. G., ANDRADE, S. L. & GASPAR, H. M. 2015. Product Life-Cycle Management In Ship Design: From Concept To Decommission In A Virtual Environment. 178-184.

NOWACKI, H. 2010. Five decades of Computer-Aided Ship Design. *Computer-Aided Design,* 42**,** 956-969.

OSI. 2016. *The Open Source Definition* [Online]. Available: https://opensource.org/osd [Accessed 1 June 2016].

PAPANIKOLAOU, A. 2010. Holistic ship design optimization. *Computer-Aided Design,* 42**,** 1028-1044.

PARSONS, M. G. 2003. Parametric Design. *In:* LAMB, T. (ed.) *Ship Design and Construction.* Jersey City, NJ: Society of Naval Architects and Marine Engineers.

PARSONS, M. G., SINGER, D. J. & SAUTER, J. A. 1999. A Hybrid Agent Approach For Set-Based Conceptual Ship Design. *International Conference on Computer Applications in Shipbuilding.* Cambridge, MA.

SIMON, E. 1996. Innovation and intellectual property protection: the software industry perspective. *The Columbia Journal of World Business,* 31**,** 30-37.

SINGER, D. J., DOERRY, N. & BUCKLEY, M. E. 2009. What Is Set-Based Design? *Naval Engineers Journal,* 121**,** 31-43.

STALLMAN, R. 1999. The GNU Operating System and the Free Software Movement. *In:* DIBONA, C., OCKMAN, S. & STONE, M. (eds.) *Open sources : voices from the open source revolution.* 1st ed. Beijing ; Sebastopol: O'Reilly.

STEFANOV, S. 2010. *JavaScript Patterns,* O'Reilly Media.

ULSTEIN. 2016. *NAO Fighter* [Online]. Available: http://ulstein.com/references/blue-fighter [Accessed 1 June 2016].

URKE, T. 1976. SFI GROUP SYSTEM - A CODING SYSTEM FOR SHIP INFORMATION.

UTNE, I. B. 2009. Life cycle cost (LCC) as a tool for improving sustainability in the Norwegian fishing fleet. *Journal of Cleaner Production,* 17**,** 335-344.

VESTBØSTAD, Ø. 2011. System Based Ship Design for Offshore Vessels. Institutt for industriell økonomi og teknologiledelse.

VON HIPPEL, E. 2001. Innovation by User Communities: Learning from Open-Source Software *MIT Sloan School of Management*.

VON HIPPEL, E. & VON KROGH, G. 2003. Open Source Software and the “Private-Collective” Innovation Model: Issues for Organization Science. *Organization Science,* 14**,** 209-223.

WARD, A., LIKER, J. K., CRISTIANO, J. J. & SOBEK, D. K. I. 1995. The second Toyota paradox: How delaying decisions can make better cars faster. *Long Range Planning,* 28**,** 129.

WOMACK, J. P., JONES, D. T. & ROOS, D. 1991. *The machine that changed the world : how Japan's secret weapon in the global auto wars will revolutionize western industry,* New York, NY, HarperPerennial.

YUNWEN, Y. & KISHIDA, K. 2003. Toward an understanding of the motivation of open source software developers. 419-429.

ZAKAS, N. C. 2009. *Professional javascript for web developers,* John Wiley & Sons.

ZHANG, W. Y., TOR, S. B. & BRITTON, G. A. 2001. A Prototype Knowledge-Based System for Conceptual Synthesis of the Design Process. *The International Journal of Advanced Manufacturing Technology,* 17**,** 549-557.

APPENDIX A: API Documentation

These methods are available for the Vessel-object. When applicable, some methods are identified using the code convention from Figure 22.

Vessel prototype

```
vessel(vesselType, cargoHoldCapacity, cargoDeckCapacity, crewSize,
          serviceSpeed, installedPower, autonomy, operationalArea) [1]
```

Creates a new vessel. This method should be used as an object constructor, using the keyword *new*. The constructor will create a vessel object containing the standard objects *missionRequirements* [1.4] and *mainDimensions* [1.5] and some methods. The first object stores the mission requirements used to create the vessel object and the second one stores all calculated parameters for the vessel. This constructor requires the following inputs: vessel type (string, for accepted values refer to APPENDIX B), cargo hold capacity (number), cargo deck capacity (number), crew size (number), service speed (number, in knots), initial guess for installed power (number, in kilo Watts), vessel autonomy (number, in nautical miles) and operational area (string). Example usage:

```
Vessel = new vessel("PSV",50000,0,24,15.85,6000,5000,"North Sea");
```

Subsystems prototype method

```
systemPrototype(subSystem) [1.3.1]
```

Create a new system as an object inside the main object, vessel. This method should be used as an object constructor, using the *new* keyword. The input *subsystem* is a vector containing pairs “subsystem category – subsystem element”. Subsystems’ categories are pre-defined and specified at the knowledge base (check the APPENDIX B for further information). The specification contains the required information that need to be provided for each category. The subsystem categories define to which category the element which is being defined belongs.

Subsystems' elements are specified by the user according to what he or she needs to specify in order to define each subsystem present in the vessel. It is important to couple each subsystem element with a subsystem category which allows the correct specification of the element. The subsystem created will be have all the subsystem's element properties empty and they need to be filled afterwards. Example usage:

```
var cargo_system = [["cargo_decks_general", "open_cargo_deck"],
                    ["cargo_holds_dry_bulk", "iron_ore"]];

Vessel.cargo_system = new Vessel.systemPrototype(cargo_system);
```

input(*vector*)

input is a method owned by any subsystem created using the method *systemPrototype*. It is responsible for inserting values in the subsystems' elements parameters. The input *vector* is a vector containing pairs "subsystem element – parameters values". The subsystem elements are the ones specified by the user when he or she creates each subsystem. The parameters values are vectors containing the properties and properties values of each element specified by the user. The required parameters for each category of element can be check at the knowledge-base. Example usage:

```
Vessel.cargo_system.input([["open_cargo_deck",["capacity",2025,
"deck_load ",2.5, "load",0, "add_on",5]],
                           [["iron_ore",["weight",50000,
"stowage_factor", 0.31, "filling",50]]]);
```

add(*subSystem*)

add is a method owned by any subsystem created using the method *prototype*. It is responsible for inserting new elements in a subsystem already created. The input *subSystem* is a vector containing pairs "subsystem category – subsystem element". For further references about the input *subSystem*, check the "prototype(*subSystem*)" reference. Example usage:

```
Vessel.cargo_system.add([["cargo_related_spaces","crane"]]);
```

`delete(vector)`

delete is a method owned by any subsystem created using the method *systemPrototype*. It is responsible for removing elements in a subsystem already created or deleting the entire subsystem. The input *vector* containing pairs “item to be deleted – item class”. The item to be deleted is the name of the item the user wants to delete. It can be one specified element or a subsystem containing one or more elements. The item class identify the class of the item that will be deleted. It accepts two values: “element” for deleting individual elements and “sub_system” for deleting entire subsystems. Example usage:

```
Vessel.cargo_system.delete([["crane","element"]]);

Vessel.cargo_system.delete([["cargo_related_spaces","sub_system"]]);
```

`area()`

area is a method owned by any subsystem created using the method *systemPrototype*. It is responsible for calculating the total area of the system to which the method belongs. The total area is based on the area of each defined subsystem, which are composed by the areas of each element defined by the user inside each subsystem. This method does not take any input. Example usage:

```
var cargoSystemArea = Vessel.cargo_system.area();
```

`volume()`

volume is a method owned by any subsystem created using the method *systemPrototype*. It is responsible for calculating the total area of the system to which the method belongs. The total volume is based on the volume of each defined subsystem, which are composed by the volumes of each element defined by the user inside each subsystem. This method does not take any input. Example usage:

```
var cargoSystemVolume = Vessel.cargo_system.volume();
```

Vessel’s Total Area, Volume and Weights methods

`area() [1.3.2]`

Besides being a method present on any subsystem created using the method *systemPrototype, area* is also a method owned by the vessel itself. It is responsible for calculating the total area of the vessel based on the area of each vessel’s system. This method does not take any input. Example usage:

```
var vesselArea = Vessel.area();
```

`volume() [1.3.3]`

Besides being a method present on any subsystem created using the method *systemPrototype, volume* is also a method owned by the vessel itself. It is responsible for calculating the total volume of the vessel based on the volume of each vessel’s system. This method does not take any input. Example usage:

```
var vesselVolume = Vessel.volume();
```

`lightWeight() [1.3.4]`

lightWeight is a method owned by the vessel. It is responsible for estimating the vessel’s light weight, based on the areas and volumes information of each vessel’s system and weight coefficients present in the knowledge-base. The weight coefficients vary according to the vessel type. Check the knowledge base for further information on weight coefficients. This method does not take any input. Example usage:

```
var vesselLightWeight = Vessel.methods.lightWeight();
```

`deadWeight()  [1.3.5]`

deadWeight is a method owned by the vessel. It is responsible for estimating the vessel's dead weight, based on the amount and type of cargo, installed power and crew size. and weight coefficients present in the knowledge-base. Check the knowledge base for further information on weight coefficients. This method does not take any input. Example usage:

```
var vesselDeadWeight = Vessel.deadWeight();
```

`displacement()  [1.3.6]`

displacement is a method owned by the vessel. It is responsible for calculating the vessel's total displacement based on the vessel's light and dead weight. This method does not take any input. Example usage:

```
var vesselDisplacemente = Vessel.displacement();
```

Calculation methods

`mainDimensionsCalc()  [1.3.7]`

mainDimensionsCalc is a method owned by the vessel. It is responsible for calculating the vessel's main dimensions, coefficients and dimensionless parameters, based on several empirical expressions and information from previous vessels present in the knowledge-base. Check the knowledge base for further information on coefficients and regressions. This method does not take any input. Example usage:

```
Vessel.methods.mainDimensionsCalc();

// accessing calculated properties
var vesselLengthPP = Vessel.mainDimensions.Lpp;
var vesselBeam = Vessel.mainDimensions.B;
```

A list of all properties calculated by the method *mainDimensionsCalc* with their respective keys to identify each property inside the object *mainDimensions* can be found on Table A1.

Table A1 – List of properties calculated by the method main_dimensions_calc.

Parameter	object key
Beam	B
Transversal metacentric height	BM
B/D ratio	B_D
Block coefficient	Cb
Center of gravity coefficient	Ckg
Main section coefficient	Cm
Prismatic coefficient	Cp
Vertical prismatic coefficient	Cpv
Waterline coefficient	Cwl
Depth	D
Froude number	Fn
Transversal Metacentric Height	GM
Vertical position of center of buoyance	KB
Vertical position of Center of gravity	KG
L/B ratio	L_B
L/D ratio	L_D
Longitudinal center of buoyance	Lcb
Length between perpendiculars	Lpp
Length at the waterline	Lwl
Draught	T
T/D ratio	T_D
Dead weight	dead_weight
Displacement	displacement
Installed Power	installed_power
Light weight	light_weight
Slenderness ratio	slenderness

```
holtrop(LWL, BREADTH, TF, TA, LCB, CB, CSM, CWL, VS, B, TR, CSTERN,
          APP, AREA_APP, div_id) [1.3.8]
```

holtrop is a method owned by the vessel. It is responsible for calculating the vessel's hydrodynamic resistance, based on the Holtrop Method. ((Holtrop and Mennen, 1982), (Holtrop, 1984)) The Holtrop method based on statistical regression of model tests and results from ship trials. The method is used to estimate the resistance of displacement ships and can result in inaccurate values for other types of vessels. This method takes as input the following parameters: waterline length, beam, draught forward, draught afterward, longitudinal center of buoyance, block coefficient, main section coefficient, waterline coefficient, service speed (in knots), vessel has bulbous bow (binary), vessel has transom stern (binary), stern type coefficient (Table A2), vector describing presence of different appendages – zero if the appendages does not exist or the coefficient value otherwise (Table A3) - (11x1 vector), vector describing area of present appendages (11x1 vector)

and the div id where the resistance graphic will be plotted. For further information about the Holtrop method refer to ((Holtrop and Mennen, 1982), (Holtrop, 1984)). Example usage:

```
var vesselResistance = Vessel.holtrop(Vessel.mainDimensions.Lwl,
Vessel.mainDimensions.B, Vessel.mainDimensions.T, Vessel.mainDimensions
.T, Vessel.mainDimensions.Lcb, Vessel.mainDimensions.Cb, Vessel.mainDim
ensions.Cm, Vessel.mainDimensions.Cwl, Vessel.missionRequirements.spe
ed, 1, 1, 3, [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0], 'plot_holtrop');
```

The following tables contain relevant information about the types and coefficients for different sterns and appendages.

Table A2 - Stern coefficients types: and values.

Stern Coefficient	
Pram with Gondola	1
V-Shaped Sections	2
Normal Section Shape	3
U-Shaped Sections with Hognes Stern	4

Table A3 - Appendages types and values.

Appendage	
Name	Value Range
Rudder behind Skeg	1.5 - 2.0
Rudder behind Stern	1.3 - 1.5
Twin-Screw Balance Rudders	2.8
Shaft Brackets	3.0
Skeg	1.5 - 2.0
Strut Bossings	3.0
Hull Bossings	2.0
Shafts	2.0 - 4.0
Stabilizer Fins	2.8
Dome	2.7
Bilge Keel	1.4

```
 shipMotion(LWL, BREADTH, CB, T, VS, heading, Position, Wave_Amp
litude, Cwp, GM, natural_period, critical_damping_percentage,delta,
div_id) [1.3.9]
```

shipMotion is a method owned by the vessel. This method provides a graphical representation of estimated motion responses and bending moment for ships, based on geometric characteristics and operational profile (Jensen et al., 2004). This method takes as input the following parameters: waterline length, beam, block coefficient, draught, service speed (in knots), vessel heading, longitudinal position on the vessel where the user wishes to study the motion in relation to the center of gravity, wave amplitude, waterline coefficient, metacentric height, wave natural period, critical damping percentage, prismatic length ratio and the div id where the motion graphics will be plotted. For further information about the method refer to (Jensen et al., 2004). Example usage:

```
Vessel.shipMotion(Vessel.mainDimensions.Lwl, Vessel.mainDimensions.B,
Vessel.mainDimensions.Cb, Vessel.mainDimensions.T, Vessel.Requirements.
speed, 90, 0, 2, Vessel.mainDimensions.Cwl, 4.2, 6.5, 20, 0.6,
'plot_ship_motion');
```

APPENDIX B: System-Based Design Structure Database

This database is responsible for storing the structural organization of the vessels systems and subsystems. Each subsystem is represented by one class, which contains the required properties for that category of element. Any subsystem present inside the vessel can be instantiated using this SyBSD structure database. The systems are identified using the reference number from Figure 22.

```
CargoSystem [1.1]

CargoRelatedSpaces
{"number_units": "", "length": "", "width": "", "height": ""}

CargoHoldsDryBulk
{"weight":"", "stowage_factor":"", "filling":""}

CargoHoldsLiquidBulk
{"volume":"", "density":"", "filling":""}

CargoHoldsContainer
{"number_units": "", "length": "", "width": "", "height": "",
"cell_guide_width": ""}

CargoTanksLiquidDryBulk
{"capacity":"", "density":"", "add_on":"", "filling":"",
"height":""}

CargoDecksContainer
{"number_units": "", "length": "", "width": "", "units_in_stack":
"", "cell_guide_width": ""}

CargoDecksRORO
{"lanes":"", "width":"", "add_on":"", "height":"",
"weight_lanes":""}

CargoDecksGeneral
{"capacity":"", "deck_load":"", "load":"", "add_on":""}
```

```
OutfittingSystem [1.2.1]

OperationalSupport
{"area":"", "covered":"", "height":""}

ShipEquipment
{"area":"", "covered":"", "height":""}

RescueFirefighting
{"number_units": "", "area_unit":"", "covered":"", "height":""}
```

CrewSystem [1.2.2]

CrewAccommodation

```
{"number_cabins": "", "beds_cabins": "", "area_cabins":"",
"height":""}
```

CrewCommonSpaces

```
{"crew": "", "area_crew":"", "height":""}
```

CrewEmergencyStairways

```
{"decks": "", "area_deck":"", "height":""}
```

ServiceSystem [1.2.3]

ShipService

```
{"crew": "", "area_crew":"", "height":""}
```

CateringSpaces

```
{"crew": "", "area_crew":"", "height":""}
```

HotelServices

```
{"crew": "", "area_crew":"", "height":""}
```

TechnicalSpacesAccommodation

```
{"crew": "", "area_crew":"", "height":""}
```

MachinerySystem [1.2.4]

MachinerySpaces

```
{"unit": "", "area_unit":"", "height":""}
```

ConsumablesTanks

```
{"unit": "", "consumption": "", "endurance":"", "density":"",
"margin":""}
```

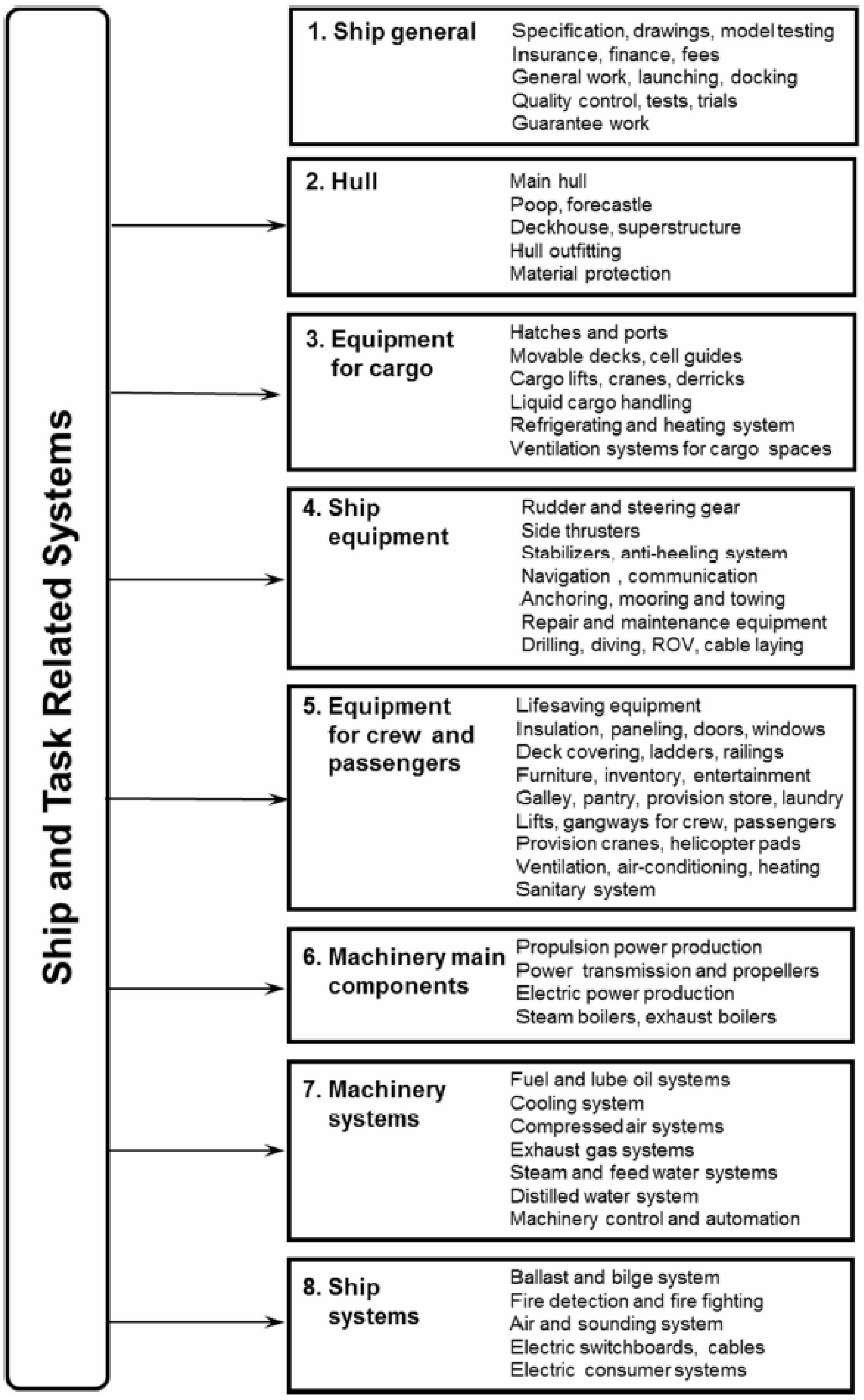
BallastVoids

```
{"density":"", "volume": ""}
```

APPENDIX C: Regressions Database

```
var data_CM = [[0.000, 0.100, 0.200, 0.300, 0.385, 0.400, 0.454, 0.500],
               [1.000, 0.998, 0.997, 0.978, 0.900, 0.879, 0.800, 0.740]];

var data_CP = [[0.000, 0.100, 0.200, 0.256, 0.300, 0.320, 0.401, 0.500],
               [1.000, 0.914, 0.799, 0.699, 0.624, 0.598, 0.550, 0.583]];

var data_CW = [[0.000, 0.100, 0.200, 0.219, 0.299, 0.328, 0.400, 0.500],
               [1.000, 0.964, 0.915, 0.899, 0.822, 0.800, 0.749, 0.715]];

var data_CB = [[0.000, 0.100, 0.197, 0.253, 0.300, 0.385, 0.400, 0.500],
               [1.000, 0.914, 0.799, 0.699, 0.611, 0.498, 0.484, 0.431]];

var data_Ckg = {
    "cargo_container": {"small":[[100, 6000],
                                  [0.6, 0.8]],
                        "big":[[6001, 18000],
                                [0.58, 0.64]]},
    "roro": [[500, 4100],
             [0.6, 0.8]],
    "ropax": [[1000, 75000],
              [0.6, 0.8]],
    "bulk_carrier": [[500, 400000],
                     [0.55, 0.58]],
    "tanker": [[500, 415000],
               [0.52, 0.54]],
    "PSV": [[100, 5500],
            [0.6, 0.8]],
    "AH&T": [[500, 9000],
             [0.6, 0.8]],
    "OSCV": [[3500, 18000],
             [0.6, 0.8]],
    "deck_barge": [[272.156, 4535.925],
                   [0.6, 0.8]]
};

var data_payload_related_outfitting = {
    "cargo_container": [[7019.099, 15765.824, 25173.590, 29356.983,
50154.849, 59708.826, 75153.481, 100157.113],
                        [0.0140, 0.0120, 0.0105, 0.0100, 0.0084, 0.0080,
0.0075, 0.0070]],
    "roro": [[4174.502, 10446.697, 25096.470, 49965.473],
             [0.0070, 0.0060, 0.0048, 0.0040]],
    "ropax": [[4017.237, 12777.122, 24947.364, 50081.152, 75089.653],
              [0.0050, 0.0040, 0.0035, 0.0029, 0.0027]],
    "bulk_carrier": [[6098.93, 24811.68, 49946.26, 75215.99, 100095.52,
124975.05, 150116.47],
                     [0.0040, 0.0031, 0.0026, 0.0024, 0.0024, 0.0024,
0.0024]],
    "tanker": [[7922.685, 25066.465, 50071.150, 75209.938, 100155.336,
125034.735, 150110.947],
               [0.0033, 0.0026, 0.0021, 0.0019, 0.0020, 0.0020, 0.0020]],
    "PSV": [[994.27, 5000.66, 5918.93, 9959.92, 14981.43, 19008.14],
            [0.0184, 0.0107, 0.0100, 0.0075, 0.0056, 0.0045]],
    "AH&T": [[4014.047, 5024.011, 6922.088, 9983.741, 10534.853],
             [0.0422, 0.0412, 0.0401, 0.0386, 0.0384]],
    "OSCV": [[958.82, 3049.46, 5010.78, 9978.57, 14993.01, 19004.74],
             [0.0369, 0.0312, 0.0279, 0.0238, 0.0215, 0.0201]],
    "deck_barge": [[272.156 , 544.311, 1315.418, 1859.729, 4535.925],
                   [0, 0, 0, 0, 0]]
};

var data_accommodation_weight = {
    "cargo_container": [[1010.989, 1802.198, 3714.286, 6065.934],
                        [0.206, 0.200, 0.187, 0.175]],
    "roro": [[1010.989, 1802.198, 3714.286, 6065.934],
             [0.206, 0.200, 0.187, 0.175]],
    "ropax": [[2153.846, 2989.011, 9934.066, 20000.000, 30065.934,
39956.044, 50021.978],
              [0.204, 0.200, 0.170, 0.155, 0.151, 0.151, 0.151]],
    "bulk_carrier": [[1010.989, 1802.198, 3714.286, 6065.934],
                     [0.206, 0.200, 0.187, 0.175]],
    "tanker": [[1010.989, 1802.198, 3714.286, 6065.934],
               [0.206, 0.200, 0.187, 0.175]],
    "PSV": [[798.771, 990.783, 1996.928, 2995.392, 3993.856, 4496.928],
            [0.210, 0.203, 0.185, 0.176, 0.169, 0.167]],
    "AH&T": [[798.771, 990.783, 1996.928, 2995.392, 3993.856, 4496.928],
             [0.210, 0.203, 0.185, 0.176, 0.169, 0.167]],
    "OSCV": [[798.771, 990.783, 1996.928, 2995.392, 3993.856, 4496.928],
             [0.210, 0.203, 0.185, 0.176, 0.169, 0.167]],
    "deck_barge": [[272.156 , 544.311, 1315.418, 1859.729, 4535.925],
                   [0, 0, 0, 0, 0]]
};

var data_machinery_weight = {
    "cargo_container": [[6026.98, 9986.95, 14904.27, 19995.65, 30091.38,
40100.09, 49978.24],
                        [0.100,0.089, 0.080, 0.074, 0.067, 0.064, 0.063]],
    "roro": [[3981.723, 9943.429, 20039.164, 30091.384, 35400.348],
             [0.0500, 0.0415, 0.0375, 0.0359, 0.0353]],
    "ropax": [[4068, 6462.1, 10030.5, 19995.7, 30091.4, 40056.6, 49978.2],
              [0.0652, 0.0603, 0.0553, 0.0490, 0.0470, 0.0460, 0.0452]],
    "bulk_carrier": [[3938.2, 7941.7, 10030.5, 19299.4, 30091.4, 37576.2],
                     [0.1381, 0.1201, 0.1144, 0.1001, 0.0918, 0.0880]],
    "tanker": [[3938.21, 7941.69, 10030.46, 19299.39, 30091.38, 37576.15],
               [0.1381, 0.1201, 0.1144, 0.1001, 0.0918, 0.0880]],
    "PSV": [[2922.37, 6255.71, 9954.34, 18082.19, 20000.00, 27990.87],
            [0.0450, 0.0377, 0.0339, 0.0298, 0.0292, 0.0272]],
    "AH&T": [[2922.37, 6255.71, 9954.34, 18082.19, 20000.00, 27990.87],
             [0.0450, 0.0377, 0.0339, 0.0298, 0.0292, 0.0272]],
    "OSCV": [[2922.37, 6255.71, 9954.34, 18082.19, 20000.00, 27990.87],
             [0.0450, 0.0377, 0.0339, 0.0298, 0.0292, 0.0272]],
    "deck_barge": [[272.16 , 544.31, 1315.42, 1859.73, 4535.93],
                   [0, 0, 0, 0, 0]]
};

var data_ship_system_weight = {
    "cargo_container": [[5903.56, 10530.48, 25065.13, 49764.04, 74986.72,
100076.02],
                        [0.0070, 0.0060, 0.0042, 0.0029, 0.0024, 0.0022]],

    "roro": [[3917.365, 12643.157, 24799.158, 32328.567, 49762.702],
             [0.0080, 0.0060, 0.0045, 0.0040, 0.0032]],
    "ropax": [[3915.59, 5105.75, 17000.39, 24794.93, 49758.03, 74980.88],
              [0.0083, 0.0080, 0.0060, 0.0053, 0.0040, 0.0035]],
    "bulk_carrier": [[5773.959, 8549.736, 24934.201, 49764.980, 74987.663,
100076.914, 125032.899, 149923.086],
                     [0.0065, 0.0060, 0.0040, 0.0027, 0.0023, 0.0020,
0.0020, 0.0020]],
    "tanker": [[5773.959, 8549.736, 24934.201, 49764.980, 74987.663,
100076.914, 125032.899, 149923.086],
               [0.0065, 0.006, 0.004, 0.003, 0.0023, 0.002,0.002, 0.002]],
    "PSV": [[1013.825, 1351.767, 3379.416, 4976.959, 8448.541, 10015.361,
15023.041, 18986.175],
            [0.0127, 0.012, 0.01, 0.0091, 0.008, 0.0076, 0.0067, 0.0062]],
    "AH&T": [[1013.825, 1351.767, 3379.416, 4976.959, 8448.541, 10015.361,
15023.041, 18986.175],
             [0.0127, 0.012, 0.01, 0.009, 0.008, 0.0076, 0.0067, 0.0062]],
    "OSCV": [[1013.825, 1351.767, 3379.416, 4976.959, 8448.541, 10015.361,
15023.041, 18986.175],
             [0.0127, 0.012, 0.01, 0.009, 0.008, 0.0076, 0.0067, 0.0062]],
    "deck_barge": [[272.156 , 544.311, 1315.418, 1859.729, 4535.925],
                   [0, 0, 0, 0, 0]]
};

var data_ship_outfitting = {
    "cargo_container": [[6039.173, 11099.021, 25136.017, 50108.814,
75408.052, 100054.407],
                        [0.0092, 0.0080, 0.0060, 0.0043, 0.0038, 0.0035]],
    "roro": [[4080.522, 12894.450, 24972.797, 50108.814],
             [0.0100, 0.0080, 0.0062, 0.0046]],
    "ropax": [[4080.522, 6202.394, 16648.531, 24972.797, 43090.316,
49945.593, 75081.610],
              [0.0106, 0.0100, 0.0080, 0.0070, 0.0060, 0.0058, 0.0054]],
    "bulk_carrier": [[7997.824, 9303.591, 24972.797, 49945.593, 75081.610,
100054.407, 125027.203, 150000.000],
                     [0.0083, 0.0080, 0.0057, 0.0040, 0.0036, 0.0033,
0.0031, 0.0030]],
    "tanker": [[7997.824, 9303.591, 24972.797, 49945.593, 75081.610,
100054.407, 125027.203, 150000.000],
               [0.0083, 0.0080, 0.0057, 0.0040, 0.0036, 0.0033, 0.0031,
0.0030]],
    "PSV": [[1003.040, 1641.337, 3677.812, 5015.198, 5516.717],
            [0.0132, 0.0119, 0.0100, 0.0093, 0.0090]],
    "AH&T": [[4012.158, 4468.085, 5015.198, 7264.438, 10000.000],
             [0.0146, 0.0140, 0.0135, 0.0120, 0.0107]],
    "OSCV": [[3009.119, 4984.802, 6322.188, 9969.605, 14984.802,
18996.960],
             [0.0078, 0.0065, 0.0060, 0.0052, 0.0045, 0.0041]],
    "deck_barge": [[272.156 , 544.311, 1315.418, 1859.729, 4535.925],
                   [0.01, 0.09, 0.08, 0.07, 0.06]]
};

var data_hull_weight = {
    "cargo_container": [[5934.938, 24904.064, 49651.143, 75161.347,
99799.407, 124764.521],
                        [0.103, 0.087, 0.080, 0.077, 0.074, 0.072]],

    "roro": [[3863.597, 13021.106, 24904.064, 49760.160, 75052.329],
             [0.116, 0.100, 0.091, 0.082, 0.081]],
    "ropax": [[3863.597, 13021.106, 24904.064, 49760.160, 75052.329],
              [0.116, 0.100, 0.091, 0.082, 0.081]],
    "bulk_carrier": [[6043.956, 14220.304, 24904.064, 41910.867,
49760.160, 74943.311, 99799.407, 124655.503, 149947.671],
                     [0.117, 0.100, 0.090, 0.080, 0.078, 0.073, 0.070,
0.069, 0.067]],
    "tanker": [[6043.956, 17599.860, 24795.046, 49760.160, 75161.347,
99799.407, 124764.521, 149838.653],
               [0.122, 0.100, 0.094, 0.083, 0.079, 0.076, 0.074, 0.072]],
    "PSV": [[1003.135, 4137.931, 4952.978, 10000.000, 14984.326,
19028.213],
            [0.1794, 0.1598, 0.1569, 0.1473, 0.1416, 0.1387]],
    "AH&T": [[1003.135, 4137.931, 4952.978, 10000.000, 14984.326,
19028.213],
             [0.1794, 0.1598, 0.1569, 0.1473, 0.1416, 0.1387]],
    "OSCV": [[1003.135, 4137.931, 4952.978, 10000.000, 14984.326,
19028.213],
             [0.1794, 0.1598, 0.1569, 0.1473, 0.1416, 0.1387]],
    "deck_barge": [[272.156 , 544.311, 1315.418, 1859.729, 4535.925],
                   [0.333, 0.250, 0.224, 0.200, 0.190]]
};

var data_deck_house_weight = {
    "cargo_container": [[3863.597, 25013.082, 49869.178, 75052.329,
99908.425],
                        [0.078, 0.060, 0.057, 0.056, 0.056]],
    "roro": [[3863.597, 25013.082, 49869.178, 75052.329, 99908.425],
             [0.078, 0.060, 0.057, 0.056, 0.056]],
    "ropax": [[3863.597, 25013.082, 49869.178, 75052.329, 99908.425],
              [0.078, 0.060, 0.057, 0.056, 0.056]],
    "bulk_carrier": [[3863.597, 25013.082, 49869.178, 75052.329,
99908.425],
                     [0.078, 0.060, 0.057, 0.056, 0.056]],
    "tanker": [[3863.597, 25013.082, 49869.178, 75052.329, 99908.425],
               [0.078, 0.060, 0.057, 0.056, 0.056]],
    "PSV": [[1003.135, 4984.326, 10000.000, 15015.674, 19028.213],
            [0.1038, 0.0870, 0.0798, 0.0765, 0.0751]],
    "AH&T": [[1003.135, 4984.326, 10000.000, 15015.674, 19028.213],
             [0.1038, 0.0870, 0.0798, 0.0765, 0.0751]],
    "OSCV": [[1003.135, 4984.326, 10000.000, 15015.674, 19028.213],
             [0.1038, 0.0870, 0.0798, 0.0765, 0.0751]],
    "deck_barge": [[272.156 , 544.311, 1315.418, 1859.729, 4535.925],
                   [0, 0, 0, 0, 0]]
};

var data_similar_vessels = {
    "cargo_container": {
        "Lpp": [[355.45, 1408.62,2022.77, 2658.56, 4039.45, 4411.86,
6011.07, 7149.84, 8025.70, 9996.15, 12010.12, 13980.15,15993.95,8007.54],
                [98.855, 150.763, 177.481, 199.618, 242.366, 250.000,
283.588, 300.382, 311.069, 330.916, 346.183, 358.397, 370.611, 379.008]],
        "Beam": [[162.284, 703.892, 2000.700, 4022.842, 6015.397,
8008.253, 10001.304, 11994.440, 14017.875, 16026.290, 18034.747],
                 [12.854, 20.131, 27.696, 34.408, 39.051, 42.696, 45.699,
48.417, 50.849, 52.996, 55.000]],
        "Draught": [[178.896, 2013.891, 4031.253, 6018.840, 8021.567,
10024.376, 12042.286, 13999.976, 16017.906, 18005.736],
                    [4.292, 9.764, 11.415, 12.264, 12.925, 13.396, 13.774,
14.104, 14.434, 14.717]],
        "Depth": [[151.03,276.90,654.50,1258.65,1988.67,3045.94, 4279.42,
6016.36, 8017.62, 10018.88, 12787.91, 14021.39, 16035.24, 18023.91],
                  [5.788, 7.523, 10.124, 12.883, 15.090, 17.770, 20.056,
22.815, 25.338, 27.545, 30.068, 31.014, 32.590, 34.009]],
        "Aux_Power": [[0.00, 1990.81, 3996.93, 5957.12, 7993.87, 10000.00,
11975.49, 14012.25, 16033.69, 18009.18],
                      [0.092, 3.159, 5.872, 8.585, 11.180, 14.128, 16.370,
18.729, 21.324, 23.683]],
        "Prop_Power": {
            "17-20": [[490.046, 1102.603, 1990.812, 2450.230],
                      [5.990, 10.000, 14.718, 16.841]],
            "21-23": [[2633.997, 3981.623, 5972.435, 7993.874, 10015.314,
11975.498, 13996.937, 16018.377, 18024.502],
                      [24.862, 30.996, 38.073, 43.971, 49.633, 54.115,
58.598, 62.608, 66.501]],
            "24-26": [[3705.972, 5972.435, 8024.502, 10015.314, 11975.498,
12986.217, 13996.937, 14977.029],
                      [40.668, 52.464, 61.782, 69.567, 76.527, 80.066,
83.368, 86.435]]
        }
    },

    "roro": {
        "Lpp": [[657.12, 1000.664, 1508.072, 2007.605, 2507.093, 3006.541,
3509.884, 4052.228],
                [97.66, 114.62, 134.50, 150.585, 164.327, 176.02, 187.427,
198.830]],
        "Beam": [[661.506, 1008.716, 1506.288, 2007.653, 2501.270,
3002.571, 3507.721, 4062.977],
                 [17.89,19.87, 21.99, 23.60, 24.93, 26.05, 27.14, 28.13]],
        "Draught": [[650.980, 996.078, 1494.118, 2000.000, 2498.039,
2996.078, 3494.118, 4043.137],
                    [5.60, 5.93, 6.25, 6.44, 6.60, 6.76, 6.85, 6.99]],
        "Depth": [[652.031, 835.518, 1005.898, 1281.127, 1500.655,
1779.161, 2001.966, 2500.000, 3001.311, 3496.068, 4039.974],
                  [12.345, 12.933, 13.405, 14.033, 14.484, 14.975, 15.289,
15.996, 16.526, 17.016, 17.487]],
        "Depth_main_deck": [[645.48, 842.07, 999.35, 1307.34, 1661.21,
2001.97, 2500.00, 2998.03, 3496.07, 4033.42],
                            [6.731, 7.084, 7.359, 7.712, 8.026, 8.379,
8.694, 8.968, 9.282, 9.557]],
        "Aux_Power": [[656.447, 1006.289, 1501.572, 2000.786, 2500.000,
2999.214, 3498.428, 4044.811],
                      [0.98,1.30, 1.64, 2.028, 2.38, 2.690, 2.97, 3.246]],
        "Prop_Power": {
            "15-18": [[652.55,1006.29,1501.57, 2000.79, 2500.00, 2908.81],
                      [4.749, 6.120, 7.842, 9.387, 10.687, 11.775]],
            "19-21": [[1257.862, 1505.503, 2000.786, 2500.000, 2999.214,
3498.428, 4001.572, 4500.786, 5000.000],
                      [11.470, 12.172, 13.472, 14.560, 15.507, 16.349,
17.120, 17.856, 18.556]],

            "22-26": [[1757.075, 2000.786, 2500.000, 2999.214, 3498.428,
3997.642, 4500.786, 5000.000],
                      [19.143, 19.951, 21.496, 22.831, 24.025, 25.113,
26.130, 27.113]]
        }
    },

    "ropax": {
        "Lpp": [[2449.58, 9917.84, 19895.44, 29992.53, 39970.12, 49947.72,
59925.31, 69902.91, 74981.32],
                [70.45, 113.11, 143.79, 165.65, 183.28, 197.74, 210.43,
222.07, 228.06]],
        "Beam": [[117.99, 1887.90, 7492.62, 9970.50, 20000.00, 29970.50,
40000.00, 49970.50, 60058.99, 69970.50, 75162.24],
                 [9.192, 14.987, 19.933, 21.205, 24.502, 26.716, 28.412,
29.778, 30.956, 31.992, 32.510]],
        "Draught": [[2477.876, 3775.811, 10029.499, 20000.000, 29970.501,
40000.000, 50029.499, 60000.000, 70029.499, 75162.242],
                    [3.58, 4.00, 5.09, 5.82, 6.27, 6.60, 6.86, 7.07, 7.23,
7.308]],
        "Depth": [[2510.76,5267.69, 9993.84, 14720.00, 20233.84, 25452.30,
30080.000, 40024.61, 49969.23, 60012.308, 69956.923,75175.385],
                  [11.563, 12.628, 13.457, 13.971, 14.426, 14.723, 14.881,
15.318, 15.616, 15.835, 16.035, 16.154]],
        "Depth_main_deck": [[2412.30,3889.23, 6350.76, 10092.30, 14818.46,
20036.92,24369.23,30080.00,36184.61,40024.61,44947.69, 49969.23, 60012.30,
69956.92, 75076.92],
                            [5.29,6.00,6.71, 7.46, 8.05, 8.45, 8.72, 9.04,
9.28, 9.44, 9.68, 9.83, 10.11, 10.31, 10.43]],
        "Aux_Power": [[2295.80, 10007.35, 19955.85, 29963.20, 40029.43,
50095.65, 59985.28, 70051.50, 75172.92],
                      [0.86,2.27,3.59,4.81,6.13,7.26, 8.39, 9.43, 10.00]],
        "Prop_Power": {
            "15-20": [[2472.40, 10007.35, 19955.85, 29904.34, 40029.43],
                      [3.499, 8.022, 12.073, 15.559, 18.479]],
            "20-25": [[2060.338, 10007.358, 19955.850, 29963.208,
40029.433, 50036.792, 60103.017, 70051.508, 75172.921],
                      [5.666,12.638,18.291,23.001, 26.864, 30.350, 33.553,
36.474, 37.981]],
            "25-30": [[14716.70, 20014.71, 30022.07, 40029.43, 50095.65,
55393.67],
                      [28.466, 32.894, 39.960, 45.707, 50.794, 53.244]],
            "30-32": [[25136.130, 29963.208, 40029.433, 50095.659],
                      [47.308, 51.548, 60.027, 67.187]]
        }
    },

    "bulk_carrier": {
        "Lpp": [[5301.04, 11780.10, 23560.20, 55955.49, 112500, 168160.99,
224410.99, 280955.49, 336910.99, 393455.49],
                [96.980, 121.856, 150.422, 195.131, 241.711, 274.012,
299.407, 320.658, 339.605, 356.250]],
        "Beam": [[4400.96,11149.10,24352.00, 39902.07, 48997.40, 98581.60,
148459.19, 197163.20, 247334.19, 296918.39, 346209.1988, 375548.9614],
                 [15.100, 19.880, 25.618, 30.080, 31.992, 40.120, 46.096,
50.478, 54.303, 57.809, 60.677, 62.351]],
        "Draught": [[5336.00,15285.90,29895.83,49460.76,99081.90,120968.95
,148399.33,198003.44,247311.72, 296910.43, 346214.56, 374804.96],
                    [5.912, 8.191, 10.014, 11.610, 14.202, 15.028, 15.969,
17.393, 18.533, 19.587, 20.442, 20.926]],
        "Depth": [[4015.05, 7528.23, 15558.34,26599.74,39648.68, 50188.20,
70263.48, 99874.52,127979.92, 149058.97, 199749.05, 249435.383, 299623.58,
350062.73, 380928.48],
                  [8.828, 10.081, 12.307, 14.534, 16.273, 17.455, 19.333,
21.453, 23.017, 24.268, 26.421, 28.227, 29.754, 31.211, 31.974]],
        "Aux_Power": [[4954.67, 22150.33, 38180.17, 49255.34, 98510.68,
148640.38, 198187.17, 247151.06, 296406.41, 345953.20, 375972.73],
                      [1.005, 1.405, 1.565, 1.644, 2.004, 2.204, 2.403,
2.523, 2.683, 2.763, 2.803]],
        "Prop_Power": [[5246.13, 15155.49, 32934.05, 49255.34, 74903.09,
98802.13, 119495.21, 148640.38,197895.73,247151.07,296406.4112, 345953.20,
375972.74],
                       [2.88,5.04, 7.63, 9.39, 11.71, 13.66, 14.98, 16.90,
19.58, 22.05, 24.25, 26.25, 27.45]]
    },

    "tanker": {
        "Lpp": [[330.39, 5947.13, 23127.75, 49889.86, 99779.73, 150330.39,
199559.471, 250110.132, 300000.000, 349889.868, 407048.458],
                [49.054, 100.135, 150.270, 191.892, 236.351, 268.041,
292.162, 313.446, 331.419, 347.500, 364.054]],
        "Beam": [[1358.76,9919.91,34176.04,50549.29, 100958.81, 151026.50,
200761.34, 250491.70, 300546.40, 350272.28, 400323.40, 409156.03],
                 [9.178, 19.863, 30.055, 34.000, 42.466, 48.219, 52.904,
56.767, 60.137, 63.178, 65.890, 66.384]],
        "Draught": [[324.44,1946.65,27901.95,49315.07,99927.90,149891.85,
199855.80, 250144.20, 299783.71, 350072.10, 400684.93, 408795.96],
                    [3.031, 5.000, 10.028, 11.821, 14.529, 16.357, 17.764,
19.030, 20.014, 20.963, 21.807, 21.983]],
        "Depth": [[523.865, 2095.460, 5238.650, 11001.164, 22002.328,
36146.682, 50291.036, 72293.364, 100058.207, 149563.446, 200116.414,
249883.586, 300436.554, 350203.725, 407043.073],
                  [4.92, 6.96, 8.92, 11.26, 13.75, 16.176, 17.913, 20.102,
22.217, 25.275, 27.654, 29.617, 31.429, 32.977, 34.601]],
        "Aux_Power": [[647.48, 9064.75, 25251.80, 50179.86, 100683.45,
150215.83, 200395.68,250575.54,300107.91, 350287.77,399820.14, 407913.67],
                      [0.101, 1.655, 2.268, 2.645, 3.069, 3.257, 3.445,
3.587, 3.681, 3.775, 3.869, 3.869]],
        "Prop_Power": [[0.00,6151.8,13920.86,29784.17,50179.86, 100683.45,
150215.83,200395.68,250899.28,30011.91,350611.51,399820.14,408237.41],
                       [1.090, 3.116, 5.000, 7.826, 10.511, 15.834,
19.838, 23.419, 26.622, 29.448, 32.227, 34.630, 35.101]]
    },

    "PSV": {
        "Lpp": [[24.615, 98.462, 295.385, 787.692, 1427.692, 1969.231,
2560.000, 3963.077, 4972.308],
                [17.037, 25.000, 35.563, 49.865, 61.891, 69.204, 75.217,
87.568, 94.718]],
        "Beam": [[21.671, 117.271, 357.920, 985.158, 1612.991, 2023.601,
2990.034, 3980.866, 4971.807],
                 [5.168, 7.514, 9.973, 13.158, 15.114, 16.176, 18.077,
19.531, 20.761]],
        "Draught": [[12.422, 198.758, 633.540, 1416.149, 2000.000,
2819.876, 4012.422, 4956.522],
                    [1.84, 3.02, 3.99, 4.99, 5.47, 6.01, 6.63, 7.03]],
        "Depth": [[27.530, 103.779, 255.536, 683.887, 1212.477, 1703.113,
2495.443, 3438.420, 4016.706, 5009.775],
                  [2.10,2.86,3.74,4.81,5.64,6.24,7.00, 7.67, 8.02, 8.55]],
        "Installed_Power": [[38.810, 595.084, 1371.281, 1992.238,
3208.279, 3984.476, 5019.405],
                            [439.269, 2217.478, 3716.988, 4938.842,
6826.451, 7936.639, 9212.96]],
        "Prop_Power": [[12.91,1097.48,1962.56,3214.98, 3963.85, 4544.87],
                       [477.80,2013.73,3247.14,4992.48,6086.30, 6900.80]]
    },

    "AH&T": {
        "Lpp": [[393.846, 873.846, 1403.077, 1969.231, 3064.615, 3987.692,
5981.538, 7975.385, 8344.615],
                [38.814, 49.540, 58.640, 65.141, 75.217, 81.880, 92.768,
101.706, 103.007]],
        "Beam": [[526.88, 912.88, 1709.75, 2821.13, 4005.27, 5431.26,
7026.65, 8307.82],
                 [10.64,12.60,15.06, 17.29, 19.084, 20.85, 22.38, 23.56]],
        "Draught": [[459.627, 857.143, 1229.814, 1975.155, 2993.789,
3987.578, 5975.155, 7975.155, 8385.093],
                    [3.80,4.50,5.014,5.72, 6.46, 6.99, 7.88, 8.57, 8.69]],
        "Depth": [[507.955, 608.722, 885.817, 1313.666, 1930.010,
2797.690, 3614.933, 4557.752, 5412.525, 6594.037, 7549.234, 8391.286],
                  [4.759, 4.991, 5.615, 6.240, 6.911, 7.651, 8.229, 8.761,
9.201, 9.733, 10.103, 10.404]],
        "Installed_Power": [[478.655, 905.563, 1448.900, 1966.365,
2302.717, 3130.660, 3984.476, 4993.532, 5976.714, 7037.516, 8331.177],
                            [3274.53, 4997.48, 7109.43, 8943.36, 9999.158,
12499.637, 15055.661, 17722.487, 20055.682, 22722.365, 25777.722]],
        "Prop_Power": [[477.728, 761.782, 1149.128, 1949.645, 2943.835,
3976.759, 5151.711, 5990.962, 7682.376, 8327.954],
                       [2667.854, 3786.073, 4997.291, 7326.428, 9958.143,
12356.667, 14918.031, 16664.327, 20017.027, 21227.644]]
    },

    "OSCV": {
        "Lpp": [[3766.15, 4652.31, 5932.31, 7015.39, 8000.00, 9981.538,
10756.923, 11987.692, 13981.538, 16000.000, 17501.538, 18116.923],
                [84.642, 91.468, 100.244, 106.419, 111.945, 121.696,
124.946, 130.146, 137.947, 145.098, 149.973, 152.086]],
        "Beam": [[3763.32,4585.16,5310.35,5987.26,6760.87,7655.34,
8912.55,9988.46,11088.55,11995.21,13240.42, 13989.97, 15972.67, 18124.65],
                 [19.307, 20.146, 20.817, 21.321, 21.881, 22.553, 23.281,
23.841, 24.401, 24.877, 25.410, 25.718, 26.475, 27.260]],
        "Draught": [[3788.820, 4608.696, 5975.155, 7987.578, 10024.845,
11987.578, 13031.056, 14000.000, 16012.422, 18099.379],
                    [5.660, 6.002, 6.439, 6.990, 7.427, 7.825, 7.996,
8.176, 8.489, 8.764]],
        "Depth": [[3827.23, 4682.06, 5587.14, 6743.55, 7598.22, 9031.01,
10539.12, 11795.87, 12926.92, 13932.23, 15163.78, 17526.25],
                  [7.119, 7.605, 8.090, 8.646, 8.993, 9.548, 10.045,
10.450, 10.809, 11.052, 11.399, 11.977]],
        "Installed_Power": [[3803.364, 4734.799, 5976.714, 6649.418,
7943.079, 10000.000, 11979.301, 13997.413, 15989.651, 18137.128],
                            [6324.240, 7545.231, 9099.064, 10042.688,
11596.376, 14037.820, 16284.818, 18531.709, 20695.246, 22969.585]],
        "Prop_Power": [[3783.086, 4893.480, 6003.873, 7320.852, 7992.253,
9954.810, 11994.835, 14009.038, 15958.683, 18089.090],
                       [3266.210, 3939.612, 4613.014, 5449.105, 5843.811,
6888.187, 7979.003, 8999.949, 9951.115, 10971.790]]
    },

    "deck_barge": {
        "Lpp": [[272.156, 544.311, 1315.418, 1859.729, 4535.925],
                [30.48, 45.72, 59.436, 60.96, 76.2]],
        "Beam": [[272.156, 544.311, 1315.418, 1859.729, 4535.925],
                 [7.9248, 9.7536, 10.668, 15.24, 21.9456]],
        "Draught": [[272.156, 544.311, 1315.418, 1859.729, 4535.925],
                    [1.8288, 1.829, 2.7432, 2.7432, 3.81]],
        "Depth": [[272.156, 544.311, 1315.418, 1859.729, 4535.925],
                  [2.471, 2.471, 3.707, 3.707, 5.149]],
        "Installed_Power": [[272.16 , 544.31, 1315.42, 1859.73, 4535.925],
                            [0, 0, 0, 0, 0]],
        "Prop_Power": [[272.156 , 544.311, 1315.418, 1859.729, 4535.925],
                       [0, 0, 0, 0, 0]]
    }
};
```

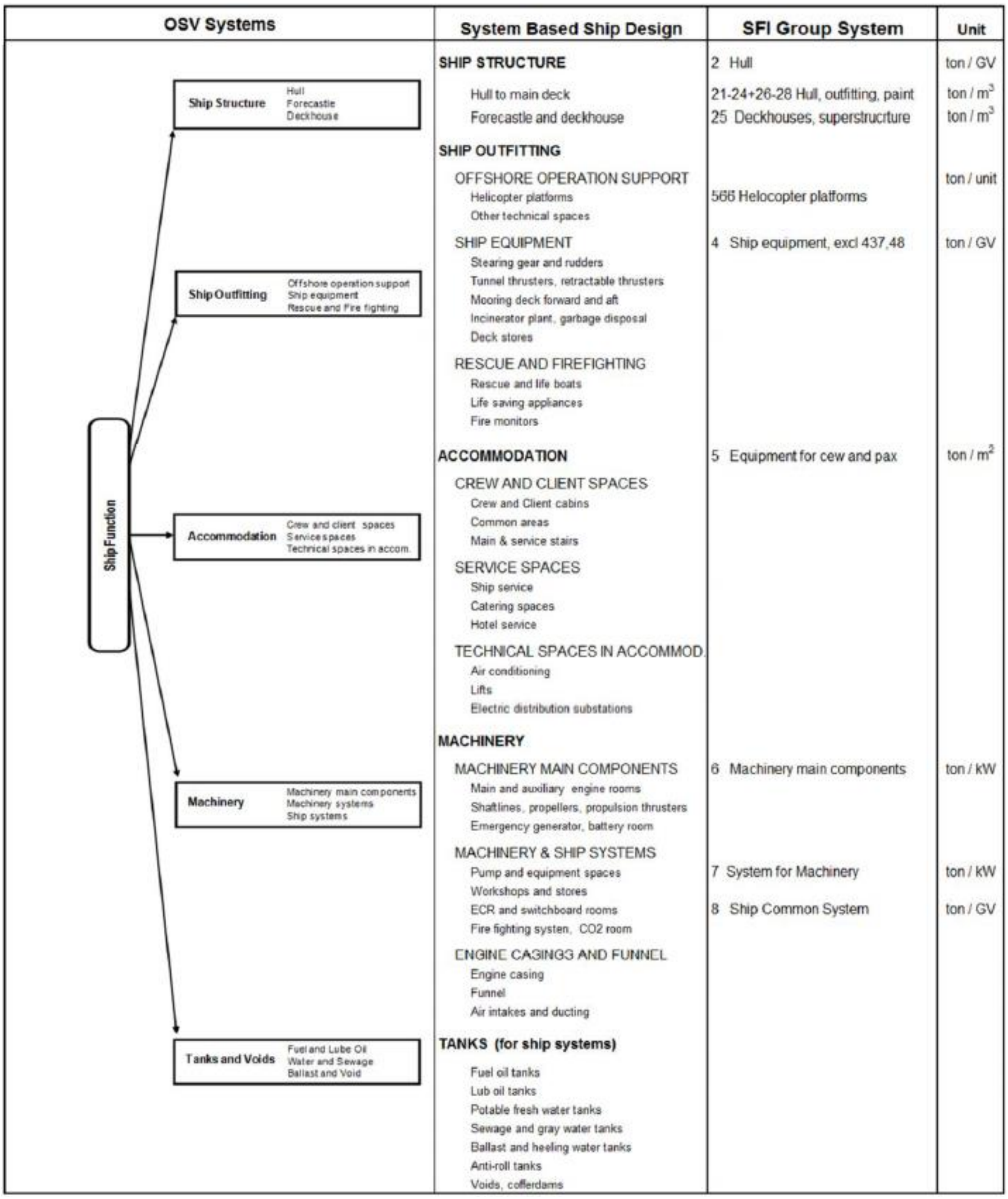
APPENDIX D: SFI Group System



Ship and Task Related Systems
1. Ship general
Specification, drawings, model testing
Insurance, finance, fees
General work, launching, docking
Quality control, tests, trials
Guarantee work
2. Hull
Main hull
Poop, forecastle
Deckhouse, superstructure
Hull outfitting
Material protection
3. Equipment for cargo
Hatches and ports
Movable decks, cell guides
Cargo lifts, cranes, derricks
Liquid cargo handling
Refrigerating and heating system
Ventilation systems for cargo spaces
4. Ship equipment
Rudder and steering gear
Side thrusters
Stabilizers, anti-heeling system
Navigation , communication
Anchoring, mooring and towing
Repair and maintenance equipment
Drilling, diving, ROV, cable laying
5. Equipment for crew and passengers
Lifesaving equipment
Insulation, paneling, doors, windows
Deck covering, ladders, railings
Furniture, inventory, entertainment
Galley, pantry, provision store, laundry
Lifts, gangways for crew, passengers
Provision cranes, helicopter pads
Ventilation, air-conditioning, heating
Sanitary system
6. Machinery main components
Propulsion power production
Power transmission and propellers
Electric power production
Steam boilers, exhaust boilers
7. Machinery systems
Fuel and lube oil systems
Cooling system
Compressed air systems
Exhaust gas systems
Steam and feed water systems
Distilled water system
Machinery control and automation
8. Ship systems
Ballast and bilge system
Fire detection and fire fighting
Air and sounding system
Electric switchboards, cables
Electric consumer systems

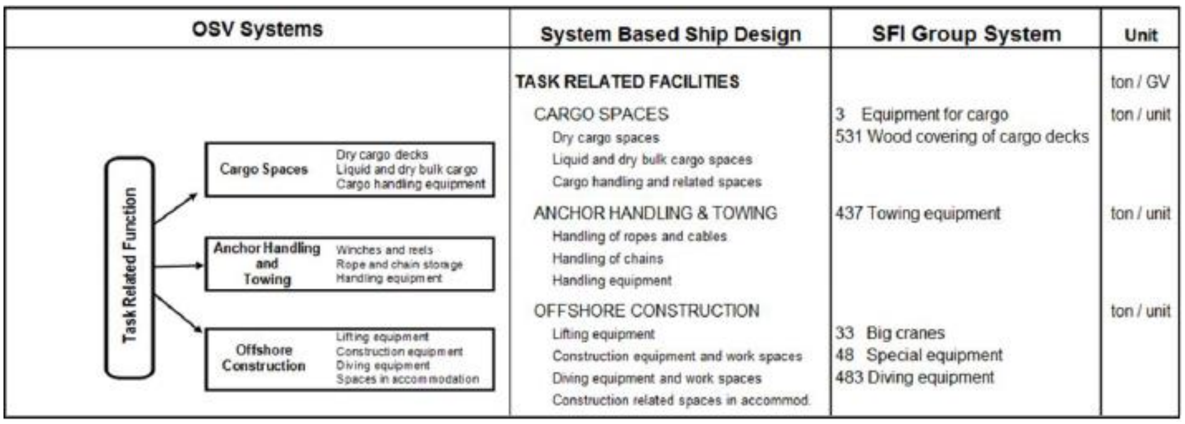
Figure D1 - SFI Group System Structure. (Levander, 2012)

APPENDIX E: System-Based Ship Design and SFI Group System Relation



System Based Ship Design	SFI Group System	Unit
SHIP STRUCTURE	2 Hull	ton / GV
Hull to main deck	21-24+26-28 Hull, outfitting, paint	ton / m^3
Forecastle and deckhouse	25 Deckhouses, superstrucrture	ton / m^3
SHIP OUTFITTING		
OFFSHORE OPERATION SUPPORT		ton / unit
Helicopter platforms	566 Helocopter platforms	
Other technical spaces		
SHIP EQUIPMENT	4 Ship equipment, excl 437,48	ton / GV
Stearing gear and rudders		
Tunnel thrusters, retractable thrusters		
Mooring deck forward and aft		
Incinerator plant, garbage disposal		
Deck stores		
RESCUE AND FIREFIGHTING		
Rescue and life boats		
Life saving appliances		
Fire monitors		
ACCOMMODATION	5 Equipment for cew and pax	ton / m^2
CREW AND CLIENT SPACES		
Crew and Client cabins		
Common areas		
Main & service stairs		
SERVICE SPACES		
Ship service		
Catering spaces		
Hotel service		
TECHNICAL SPACES IN ACCOMMOD.		
Air conditioning		
Lifts		
Electric distribution substations		
MACHINERY		
MACHINERY MAIN COMPONENTS	6 Machinery main components	ton / kW
Main and auxiliary engine rooms		
Shaftlines, propellers, propulsion thrusters		
Emergency generator, battery room		
MACHINERY & SHIP SYSTEMS		
Pump and equipment spaces	7 System for Machinery	ton / kW
Workshops and stores		
ECR and switchboard rooms	8 Ship Common System	ton / GV
Fire fighting systen, CO2 room		
ENGINE CASINGS AND FUNNEL		
Engine casing		
Funnel		
Air intakes and ducting		
TANKS (for ship systems)		
Fuel oil tanks		
Lub oil tanks		
Potable fresh water tanks		
Sewage and gray water tanks		
Ballast and heeling water tanks		
Anti-roll tanks		
Voids, cofferdams		

Figure E1 - SyBSD and SFI relation- Ship Functions. (Levander, 2012)


Task Related Function
Cargo Spaces
Dry cargo decks
Liquid and dry bulk cargo
Cargo handling equipment
Anchor Handling and Towing
Winches and reels
Rope and chain storage
Handling equipment
Offshore Construction
Lifting equipment
Construction equipment
Diving equipment
Spaces in accommodation

OSV Systems	System Based Ship Design	SFI Group System	Unit
	TASK RELATED FACILITIES		ton / GV
	CARGO SPACES	3 Equipment for cargo	ton / unit
	Dry cargo spaces	531 Wood covering of cargo decks	
	Liquid and dry bulk cargo spaces		
	Cargo handling and related spaces		
	ANCHOR HANDLING & TOWING	437 Towing equipment	ton / unit
	Handling of ropes and cables		
	Handling of chains		
	Handling equipment		
	OFFSHORE CONSTRUCTION		ton / unit
	Lifting equipment	33 Big cranes	
	Construction equipment and work spaces	48 Special equipment	
	Diving equipment and work spaces	483 Diving equipment	
	Construction related spaces in accommod.		

Figure E2 - SyBSD and SFI relation- Task Related Functions. (Levander, 2012)

APPENDIX F: Scientific Paper Draft

An Open Source Approach For A Conceptual Ship Design Tools Library

Thiago G. Monteiro; Henrique M. Gaspar
Faculty of Maritime Technology and Operations
Norwegian University of Science and Technology in Aalesund
Postboks 1517 N-6025 – Aalesund – Norway
E-mail: thiagogabrielm@gmail.com; E-mail: henrique.gaspar@ntnu.no

KEYWORDS
Conceptual Ship Design, JavaScrip, Open Source Software, System-Based Ship Design.

ABSTRACT

A vessel is a complex and integrated system, which in composed by several subsystems and parts which can have common interfaces and interact in a non-linear way. With the continuous development of ship design techniques, a continuous increase in the amount of information generated and handled by the design process can be noted. Having an efficient way of handling all this information during the vessel design process in essential to produce relevant designs in the competitive ship market.

This work proposes an investigation about how the conceptual design phase of a vessel can be approach and improved using concepts from Knowledge-Based Design, System-Based Ship Design and Open Source Software. These theories are combined to put together an Open Source Conceptual Ship Design Tools Library, which provides a set of design tools to be applied in the beginning of a vessel design process.

The development of the tools library is approached in details in this work. It is structured using knowledge-based design prototype concept. The vessel is subdivided using system-based ship design theory to make the design task less cumbersome and easy to be handled by the tools library. JavaScript is used as an open standard to implement the tools library in a web-based platform. The way JavaScript should be used in order to better deal with the vessel subdivision is also discussed, putting some light in the object oriented programming methodology.

Once the tools library is implemented, a case study is conducted to evaluate how well and appropriate its performance is in a real world problem. This study is done in cooperation with Ulstein, which provide precious information and discussions about their design process.

1. Introduction

The ship design task is complex by nature. A vessel is a complex and integrated system, which in composed by several subsystems and parts which can have common interfaces and interact in a non-linear way. (Erikstad, 1996) proposed seven characteristics to describe a vessel system: is a self-contained structure operating in the boundary between two fluids; it consists of a multi-dimensional, partially non-monetary performance evaluation; high-cost of error if inefficient design; shallow knowledge structure between form and function; traditional industry with preconceived standards; strict time and resource constraints; predominantly one of a kind and engineered to order solutions.

With the continuous development of ship design techniques, a continuous increase in the amount of information generated and handled by the design process can be noted. (Gaspar et al., 2012)

During the vessels' life-cycle, all these information need to be handle and exchanged by several people, being them part of the same company or members of different organizations involved with the vessels' life-cycle. In order to make the design process as efficient as possible, it is important a common standard to identify vessel's systems and components to be used for all the parts involve in the life-cycle.

One example of this kind of standard is the SFI Group System, which is a coding system to unequivocally identify any vessel component at any point of the vessel's life-cycle. This kind of coding system is useful for solving communication, cost and control issues. SFI is the most used and well-known coding system used around the world for vessel design.

Besides the standard for components and systems identification, there is also the need for a common platform for handling different types of analyses results involved in the vessel design process. Ship designer already have at their disposal advance techniques to evaluate the vessels' performance according to different merit figures, such structural resistance, hydrodynamic forces, seakeeping capabilities, stability and so on. What they don't have is a common platform to perform all these kinds of analyses, handle the results and combine the data, which could definitely improve several aspects of the design process by means of accelerating the design tasks, reducing rework, saving time and money and etc.

The concept of open libraries was developed with the advent of internet, since it became easy for people to learn, develop and share knowledge about programming. Web based open source software can be an interesting alternative for fulfilling the need for this common platform, since several applications can be developed and integrated in a ship design library which can address the ship design problem from different angles. The applicability of web oriented open source technology need to be investigated to check the extent of its collaboration to the ship design field.

1.1. Problem Statement

The main objective of this study is to develop an open source conceptual ship design tools library which will address vessels' conceptual design problem using knowledge based design and system based design approaches. This object statement can be a little cumbersome at first glance, but it can be easily explained. At first, the concept of a tools library need to be approached. The term library here refers to a collection of objects, which work together to accomplish the required design task. These objects are the tools which the library contains and organizes. Think in these objects as structures to store and handle knowledge or perform functions, calculation or operation.

Secondly, the concept of open source software is relevant here, since it provides a powerful and yet free platform where the tools library can be implemented. Being developed in an open platform allows the library to not be bounded to any proprietary software, which add freedom and flexibility while developing and integrating different tools.

Last but not least, the conceptual design phase is the beginning of the design process, but still holds a big potential affects the final design. Decisions taken here affect largely the cost structure of the product while being cheap to be done, making this phase an obvious target for improvement search, which the tools library will try to address.

In this light, the main goal of this study is to develop, using a not proprietary platform, a collection of useful and integrated tools to be applied in the conceptual design phase of vessels. This problem statement leads to important questions which need to be investigated. They are:

Question 1: How can the vessel be efficiently subdivided in order to allow better design control and knowledge handle in the initial stages of the design process?

Question 2: Which aspects of open source technology can improve the conceptual ship design process?

Question 3: How to develop a tools library which has a structure flexible enough to contemplate the design of any kind of vessel but at the same time be capable of providing results which are good enough to compose a solid ground for the continuation of the design process?

2. Knowledge-Based Design

The knowledge-based design (KBD) process can be seen as a problem solving process of searching through a state space defined by the syntactic knowledge (design variables) and the interpretative knowledge (design performances), where the states represent the design solutions. The searching process should be done using reasoning based on goal and decision variables, which can be constrained by the world or context where the design is applied or produced. (Coyne et al., 1989)

The KBD theory can be used to address any design problem by doing the mapping between the syntactic and the interpretative spaces (Figure 1). One tool to handle knowledge while trying to do this mapping is the Design Prototype.

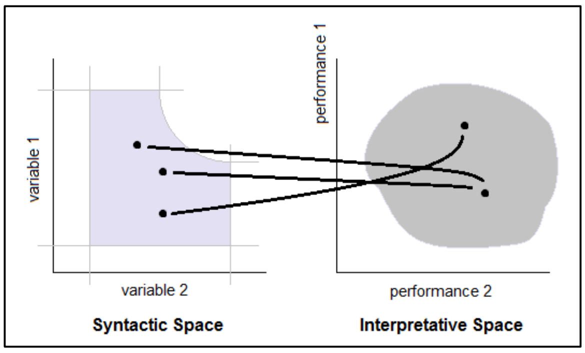


Figure 1 - Example of mapping between syntactic and interpretative spaces. (Coyne et al., 1989)

2.1. Knowledge and Design Prototype

Given that designers design from experience, it is needed a system of storing this experience in a coherent structure. (Gero and Rosenman, 1990) The prototype should be understood as a conceptual schema for knowledge or a clear way of representing the design and its properties.

The prototype represents a class of elements from which instances of elements can be derived. A prototype brings together the three types of variables groups (function, structure, behavior) which define the designed artifact and the relation between them, which includes process for selecting and obtaining values for variables.

Instance can be derived by inheriting properties, functions and variables from a generic prototype. It is possible to derivate new instance from prototype which have already been derivate from other prototype, which make possible to develop a complex hierarchy in the design process. (Gero and Rosenman, 1990)

A diagram of a prototype schema can be seen in Figure 2. The function properties include the intended function in the form of goals and requirements, and the expected behaviors as attributes and variables. The structure properties include the vocabulary, the prototype description, its configuration and the actual behaviors as attributes and variables of the prototype.

Knowledge plays a big hole into the prototype schema. Relational knowledge is required for every mapping from a property to another property. Besides the relational knowledge, the prototype also stores qualitative knowledge, computational knowledge and context knowledge. The qualitative knowledge complements the relational knowledge and provides information on the effects of changing structure variables values on behavior and function properties. The computational knowledge is the quantitative counterpart of qualitative knowledge and specifies symbolic or mathematical relationships among the properties variables. The context knowledge identifies the external variables for a design situation, which should come from the context where the design is inserted. The qualitative and computational knowledges are subjected to constraints, which on function properties appear as expected behaviors and on structure properties reduce the set of possibilities.

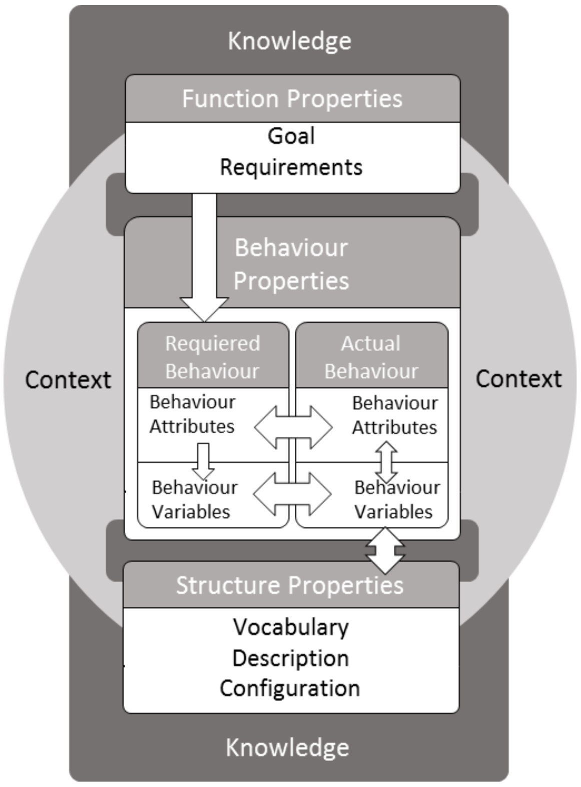


Figure 2 - Prototype schema diagram. (Gero and Rosenman, 1990)

3. System-Based Ship Design

According to (Hubka and Eder, 1988), who described the bases for technical systems, the system thinking has as its main features (Levander, 2012):

- Delivers the relationships that are valid for all products;
- Presents an opportunity to treat problems as a whole;
- Is a necessary pre-condition for a successful design and engineering effort;
- Provides a framework for the design task and formalize many logical operations;
- Supports those human operations, that are not strictly logical, like intuition and creativity.

Kai Levander, who believed that this system approach could help the development of innovative vessel designs, applied the idea of system thinking to ship design, developing a design methodology called System-Based Ship Design (SyBSD). This methodology works as a framework for the vessel design task. This framework is structured over the idea of dividing the vessel in different systems, based on their functions, which work together to accomplished the desired ship mission. (Levander, 2012)

Different from a top-down design approach where the design starts from the vessel and continue to detail the vessel's systems, the SyBSD uses a bottom-up approach, going from the vessel's required functions to the composition of the vessel itself. The designs start from the mission specification, which defines task, capacities and expected performance by the vessel's stakeholders. This approach straightens the beginning of the design spiral, delaying the beginning of the decision process and reduces the number of iterations needed to find a feasible solution. The SyBSD spiral can be seen in Figure 3.

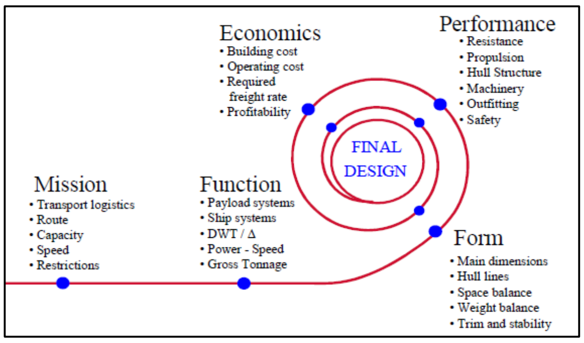


Figure 3 - System-Based Design Spiral (Erikstad and Levander, 2012)

According to (Erikstad and Levander, 2012), the SyBSD process can be summarized as follow:

Customer requirements - Mission statement

- Task, capacity, performance demands, range and endurance;
- Rules, regulations and preferences;
- Operating conditions (wind, waves, currents, ice).

Functional requirements - Initial sizing of the ship

- Based on capacity, where the areas and volumes needed for cargo spaces and task related equipment defines the size of the vessel;
- Based on weight, where the cargo weight and the weight of task related equipment and of the ship itself defines the size of the vessel.

Form - Parametric exploration

- Variation of main dimensions, hull form and layout of spaces on board to satisfy the demands for both capacity and weight.

Engineering synthesis

- Calculating and optimizing ship performance, speed, endurance and safety.

Evaluation of the design

- Calculating building cost and operation economic.

A functional breakdown is used in order to divide the vessel in systems. The vessel is split into the categories Ship Systems and Payload Systems. The Ship Systems are all systems related to the safe and correct operation of the vessel, without taking the cargo into consideration. The Payload Systems are functions and requirements which generates cash flow for the vessel, which can include cargo and cargo related systems but also specific systems for specialized vessel such offshore support vessels, which can have winch and heavy lift cranes. Due to this special cases, the Payload Systems can also be called Task Related Systems. An example of this division can be seen in Figure 4.

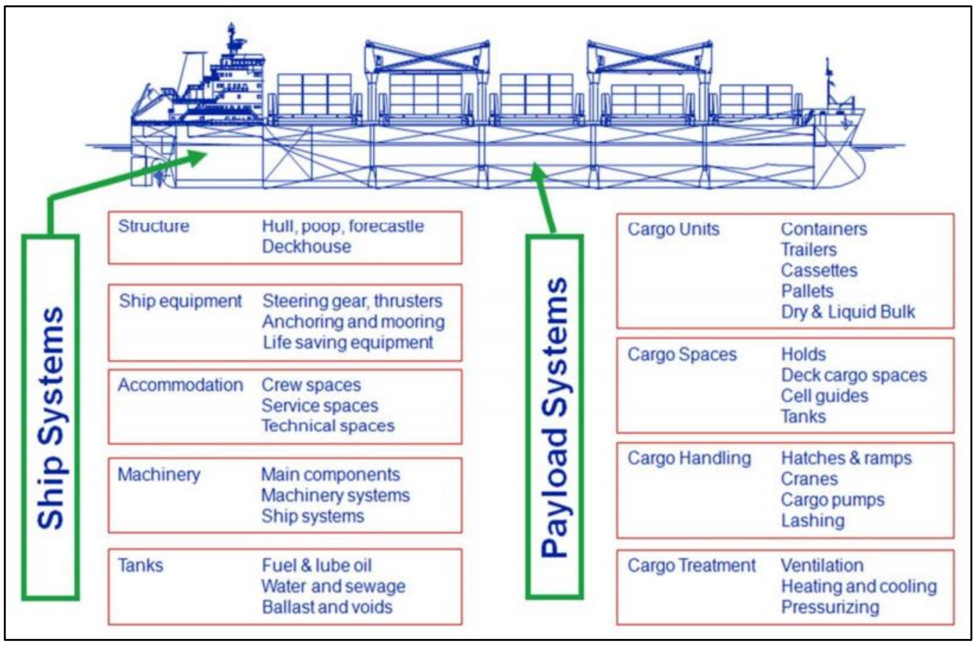


Figure 4 - Payload and ship functions in a cargo vessel. (Levander, 2012)

In order to facilitate data collection, the SyBSD divides the systems based on the structure of the SFI Group System. (Urke, 1976) The SFI group system was developed at the Norwegian Ship Research Institute and is the most widely used classification system for the maritime and offshore industry worldwide. It is an international standar,d which stablishes a functional breakdown of the ships technical and economic information and is used by shipping and offshore companies, shipyards, consultancies, software suppliers, authorities and classification societies. It helps the control of operations by tying together all their procedures such as purchasing, accounting, maintenance and technical records.

The correlation between SyBSD and SFI is not completely accurate, since the former does not distinguish between payload and ship systems (Figure 5), which results is some minor differences between the subdivision of these two structures. Although some differences exist, they can be overcome making some special relations among discordant items. (Erikstad and Levander, 2012)

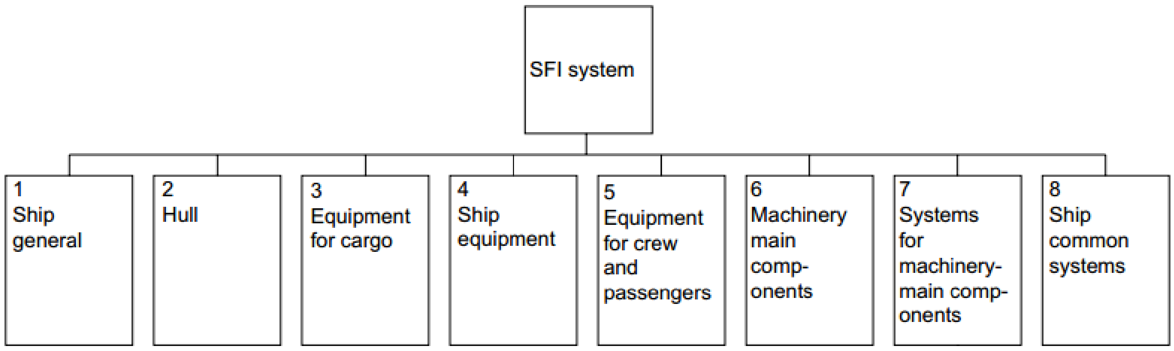

SFI system
1 Ship general
2 Hull
3 Equipment for cargo
4 Ship equipment
5 Equipment for crew and passengers
6 Machinery main comp-onents
7 Systems for machinery-main comp-onents
8 Ship common systems

Figure 5 - SFI main groups structure. (Utne, 2009)

Since the vessel are usually generic, they follow a design pattern based on previous experience with resembles a scaling process. This traditional approach easily locks the designer to his first assumption, making him patch and repair the same design, what makes this traditional method not prone to innovation.

The SyBSD, using a bottom-up approach determines the needed area, volume and weights for each vessel's function, and from this figures estimates the displacement, main dimensions and building costs. By doing this evaluation without defining the vessels dimension, the SyBSD method does not lock assumptions in the conceptual phase and supports a more creative process in the start of the project.

The SyBSD method is suitable for the early design decisions, and can be considered as a checklist that reminds the designer of all the factors that affect the design and record his choices. Its use ensure that the design is based on the most fitted basis ship, and reduces the number of iterations in the design spiral later on. (Vestbøstad, 2011) The final product of the SyBSD method application is a complete description of the new ship, which can be used as an advanced start point for the next design phases.

3.1. The Perks of Delaying Decision Making

SyBSD methodology delays the start of the decision making process while the traditional ship design spiral methodology value the early decision making. This traditional approach can prejudice the outcome of the design process, mainly due to three factor: evolution of product's cost, management's ability to affect these costs, and evolution of designers' knowledge about a design problem. The negative effect of these factor in the outcome of the design process can be reduced by mean of delaying the decision making as far as possible. (Bernstein, 1998)

The first factor to be approached is the product's cost. The design team is responsible for defining everything related with the product's cost. The chosen design, the way it will be produced, how it should be transported, required sale price are all important decisions defining the structure of a product's cost. This decision process is a tricky task, since the choices made in the very beginning of the design process (with the least data) have the most impact in the product's costs structure. (Ward et al., 1995)

At the end of the conceptual design phase between 60% and 80% of the product's total life-cycle cost is determined. (Dierolf and Richter, 1989) Although the impact of the conceptual design phase decisions in the final product are enormous, the amount of resources spent in this stage are minimal. For a vessel design, the time and money allocated to the conceptual design phase are about 2% of that of the design, detailing and construction phases all together. (Levander, 1991)

(Gaspar, 2013) also considered in this evaluation the effect of the changes in the design over the vessel life-cycle (Figure 6). The cost of removing or correcting a design flaw or making any change in the vessel design in the construction phase at the ship yard can be as big as 1000 times higher than if the same change was made during the conceptual design phase.

With these considerations, it is possible to notice the importance of the decisions made in the early product development and how they can affect the product's cost structure. Also, the impact in lowering the cost structure of decisions made in the later stages of development are small. (Anderson, 1997)

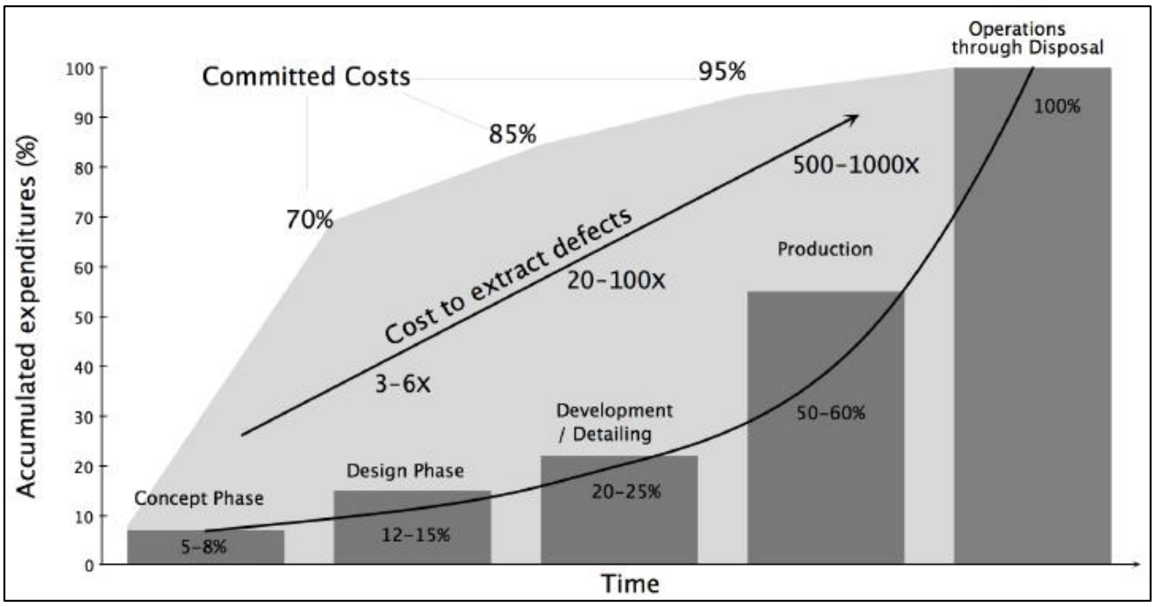


Figure 6 - Costs evolution through the main design phases. (Gaspar, 2013)

Other factor influencing the product's cost structure is the management's ability to influence a product's design. During the conceptual design phase, the design team has a great capacity to influence the product's design. As the design progress, this capacity is greatly diminished. This characteristic is due to the fact that the design team constraint the available options for design solution with each decision made. (Krishnan et al.,

1991) So, as the design progress, more decisions are made and more constraints are inserted in the design formulation, making the management's power to influence the product's costs structure decline, while these costs increase.

The conceptual design phase is also characterized by the lack of knowledge from the design team. At the beginning of the product's development the design team is not really aware about the problem's variables, constraints and user's needs. As the problem is worked, the knowledge starts to be consolidate and the design team becomes more capable of making better decisions.

As already discussed, these initial decisions have the most impact in the product's cost structure, however they are made when the design team has the least knowledge about the product and the process.

(Erikstad, 1996) corroborates with the relation between the design knowledge and the design freedom and how they evolve over the life-cycle of a vessel. At the beginning of the conceptual design, no decisions have yet been made, and the only constraint are that related to the top-level mission requirements and all subsequent decisions will constraint the design freedom.

The limited design knowledge can be mainly attributed to the uncertainty in the relation between the form and the function of the vessel. The uncertainty in these mapping functions can be related to the vessel itself or to the vessel's environment. Specially in the early stages of the design process the mapping between form and function need to be modelled, in a large extent, using heuristic and empirical rules.

Figure 7 show how useful delaying the decision make process can be.

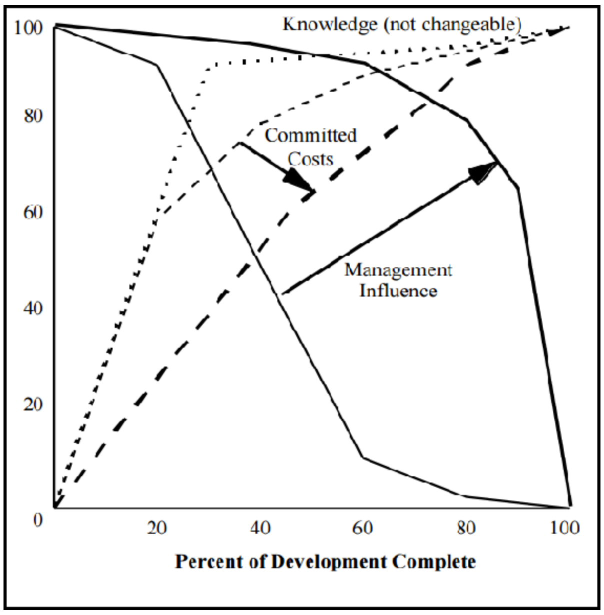


Figure 7 - Perks of delaying the decision making process. (Bernstein, 1998)

Firstly, it can delay the commitment of costs for a moment when more knowledge about the product and the design process is available. Secondly, it can increase the manager's influence over the late design, since several constraints can be considered only when the decisions regarding them are made, which would give more options for the product's design. The knowledge is considered unchangeable, since it is already considered that it is obtained as soon as possible. (Bernstein, 1998)

(Kalyanaram and Krishnan, 1997) pointed additional benefits of delaying the decision making process:

- better balance between customer needs and technical feasibility;
- keep the design open to receive the latest technology available;
- more competitive products, both in terms of price and performance;
- better track changes in customer desires.

4. Open Source Technology and JavaScript

In order to protect intellectual property and ensure the possibility of profit, several commercial software companies keep their software's source code in secret. The source code is a sequence of logical instructions written in a certain programming language and used by a computer to execute a given task or achieve a given purpose. In order to protect the source code, companies release the program in a binary version (the code converted in a sequence of zeros and ones), which can be understood by computer but is difficult for users to interpret. (Simon, 1996)

This kind approach goes against the origins of computer programming, where codes were written and shared by several individuals, with no commercial interested involved. (von Hippel and von Krogh, 2003) To fight this tendency, the now called Open Source Software (OSS) movement was created, aiming to ensure free access to programs' source code, making possible for users to understand, change, adapt and improve the original programs' source code.

The term "free software" is not related to price or value. It is about freedom. According to (Stallman, 1999) a free software can be called like this if an individual user can:

- Run the program for any purpose;
- Have access to the program's source code so it is possible to modify it;
- Redistribute copies of the original program, either gratis or for a fee;
- Distribute modified versions of the program.

The possibility of selling copies of the program is a crucial feature to finance the free and open source software development and communities. In order to ensure all this freedom, the copyleft licensing concept was developed, adding distribution terms to the conventional copyright practice, which give anyone the right to use, modify and redistribute the code, since the distribution terms are kept intact.

The licensing is a complex subject in software development and distribution. There are several licensing standards for both free and proprietary software. The following diagram (Figure 8) by (Kuei, 1999) shows the different categories of software licensing. Some of them limit the access to the closed program (proprietary software), some only limit the access to the source code (public domain software without source code) while others only impose distribution terms (copylefted software).

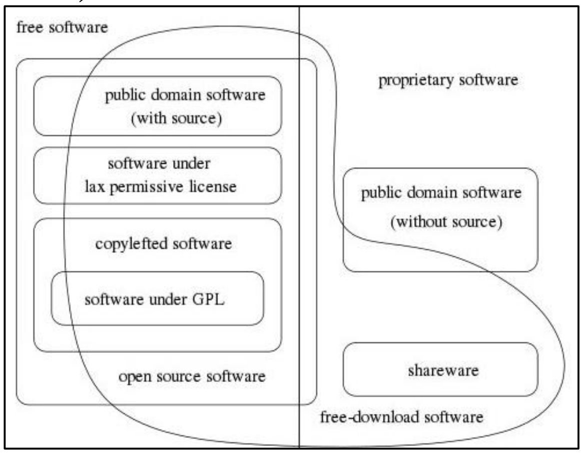


Figure 8 - Overview of types of software licenses. (Kuei, 1999)

4.1. JavaScript as an Open Standard

An OSS needs to be written in a computational language which is not propriety, in other words being owned and controlled by one company. An open computational language can be called an Open Standard (OS). This is the case of JavaScript, which is a particular implementation of the ECMAScript language standard.

An OS like JavaScript is open but not open source, since is it not a program. It is only a document describing expected behaviors for lines of code written in its computational language syntax. Although an OS cannot be open source, an implementation of it can be (and in this case it is an OSS).

In this section, JavaScript is presented as an OS, presenting its background, main features and justifying why it will be the choice for developing the conceptual ship design open source tools library.

4.2. Object Oriented Programming in JavaScript

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data and code. In OOP, computer programs are designed by making them out of objects that interact with one another. The most popular and developed model of the OOP is the class-based programming (CBP). In this model, objects are entities that combine data, behavior and identity. The structure and behavior of an object are defined by its class, which includes all objects of a specific type. The objects are created based on classes and are considered instances of them, inheriting some of their properties.

JavaScript does not follow the CBP model. It is structured as a class free language, where object inherit properties from other objects. This model is called prototype-based since behavior reuse is performed via a process of cloning existing objects that serve as prototypes. (Stefanov, 2010) This approach is powerful, making the inheritance process easier to implement, but it is also different from what a conventional CBP language is. (Crockford, 2008)

An object, which is the JavaScript's core data type and its only complex one, is an unordered list of primitive (Number, String, Boolean, Undefined, and Null) and complex data types that is stored as a series of key-value pairs. The key property serves as an identifier while the value represents the value of the expression, which can be a primitive or an object value. Each item in the list is called a property or, if an item is a function, it is called method. (Stefanov, 2010) This easy notation inspired JSON, a popular data interchange format. (Crockford, 2008) An example of an object (car) containing both properties (type, model and color) and methods (showColor()) can be seen below.

```
var car = {                      // car object.
    type:"Fiat",                 // property.
    model:"500",                 // property.
    color:"white",               // property.
    showColor: function (){      // method.
        alert(this.color)
    };
}
```

One of the fundamental concepts from OOP is the inheritance concept. Usually in a CBP language, objects are instances of classes, from which they can inherit properties and functions. In JavaScript, this process is a little different since object inherit from other objects.

Other important concept of OOP is the encapsulation concept. Encapsulation refers to enclosing all the functionalities of an object within that object so that the object's methods and properties are hidden from the rest of the application, allowing to abstract or localize specific set of functionalities on objects.

This two concept will be important since they allow the build of applications with reusable code, scalable architecture, and abstracted functionalities.

4.3. Why JavaScript?

The idea in this work is not only create an open source tools library, but also to create a ship design tool which is simple to use, not computational intensive and requires as minimum effort as possible to share designs and results.

Working with a software which requires to be installed in the computer could make it difficult to share result with clients, team members or other stakeholders. Using an online platform for the vessel design can reduce these information sharing difficulty, as the only thing one need to access the information about the design is a web browser, and the only thing needed to edit is a text editor, reducing the need for client-side software to a minimum. Also, since all the processing is done in browser, the computational requirements are very low.

Web development is not restrict to one operational system or one platform, since internet is universal. When thinking about client-side web development, JavaScript is an obvious choice. JavaScript is so important and popular because it is the language of the web browsers. (Crockford, 2008) It is supported by all modern web browsers without the need of plugins, since each browser has its own built-in JavaScript engine. (Flanagan, 2011) JavaScript composes a triad of web technologies essential for web development, together with HTML to specify the content of the web page and CSS to specify the presentation of the web page. JavaScript is responsible for describing the behavior of the web page. (Flanagan, 2011)

The position of JavaScript as the main languages in web browser makes it development very fast, with new tools and libraries been developed all the time, by the gigantic JavaScript's community.

Since JavaScript is prototype-based with first-class functions, supporting object-oriented, imperative, and functional programming styles (Flanagan, 2011), its prototype and inheritance capabilities make it an good choice for dealing with objects to handle the vessel subsystem division and the knowledge base data. It has a simple API for working with text, arrays, dates and regular expressions, which can be completed using third parties' APIs. It has no input-output functionalities, relaying on the environment where it is embedded to handle these operations. (Flanagan, 2011)

The biggest drawn back of JavaScript is the fact that it is an OOP language which is PBP model. This makes it an unusual language for most developers, which are used to conventional CBP model. Using, directly, programming techniques from CBP will not work in JavaScript, which can be frustrating for an unadvised programmer. (Crockford, 2008)

5. Research Approach

The methodology for this work will be developed following the system architecture presented in Figure 9, where the tools library components are organized and their relations stablished. The User provides inputs to the library and receives outputs from it. The Tools Library is developed using JavaScript, for both Prototype and Knowledge-Base. The Prototype contains the most important KBD elements, namely Function, Behavior and Structure. The User's inputs feed the Function block, while the Structure and Behavior blocks are developed using SyBSD theory. The Structure and Behavior blocks receive information from the Knowledge Base through an Inference Mechanism, respectively from the SyBSD Structure Database and Regression Database blocks.

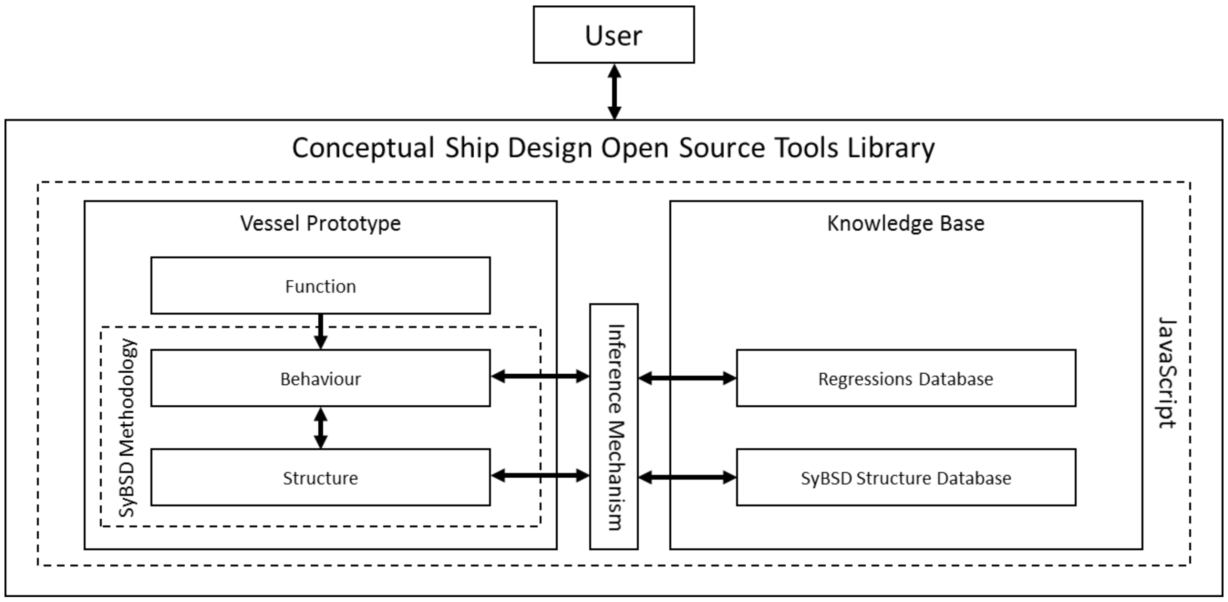


Figure 9 - Conceptual Ship Design Open Source Tools Library architecture.

5.1. Knowledge-Base

One important element in KBD is the knowledge-base. It is responsible for storing knowledge about the design process, which can be accessed by some sort of inference mechanism to retrieve facts, knowledge and control whenever the reasoning process requires. The tools library knowledge base was constructed to be the most generic as possible in order to provide knowledge to the design of several kinds of vessels and is composed by two databases: the Regressions Database and the SyBSD Structure Database.

The regression database contains important vessel design coefficients regression and knowledge about previously built vessels of several types, including container carriers, bulk carriers, tankers, ferries, roro and offshore support vessels.

The SyBSD database contains classes definitions based on the SyBSD structure. These classes are used to instantiate vessel elements, which compose the vessel's subsystem, which compose the vessel's systems. Whenever the user instantiated a new element, this database will be accessed and the required class structure will be retrieved.

5.2. Vessel Structure

The proposed vessel prototype subdivision structure is presented in Figure 10. SyBSD uses a simple division for the physical structure of the vessel. There are two main groups of systems: Task Related Systems [1.1] and Ship System [1.2]. The Task Related Systems group includes any cargo and cargo handling systems and specialized system for offshore support vessels which are not related directly to cargo but is related to the money making capacity of the vessel. The Ship Systems group includes any system required for the vessel to safely operate and considers the Outfitting [1.2.1], Crew [1.2.2], Service [1.2.3] and Machinery Systems [1.2.4].

Besides the vessel physical structure, there is also other important elements to consider in the prototype structure. We are defining two JavaScript objects to store data. The first one is related to the required functions of the vessel, namely the Mission Requirements object [1.4]. The second one is responsible for storing the vessel's main dimensions and behaviors, namely the Main Dimensions object [1.5].

Last but not least the prototype will hold several JavaScript methods (or function), which will be responsible for data handling, reasoning, knowledge retrieve and knowledge application. The methods present in the vessel prototype are: Prototype [1.3.1], Area [1.3.2], Volume [1.3.3], Light Weight [1.3.4], Dead Weight [1.3.5], Displacement [1.3.6], Main Dimensions [1.3.7], Holtrop [1.3.8], Seakeeping [1.3.9] and Hull Lines [1.3.10].

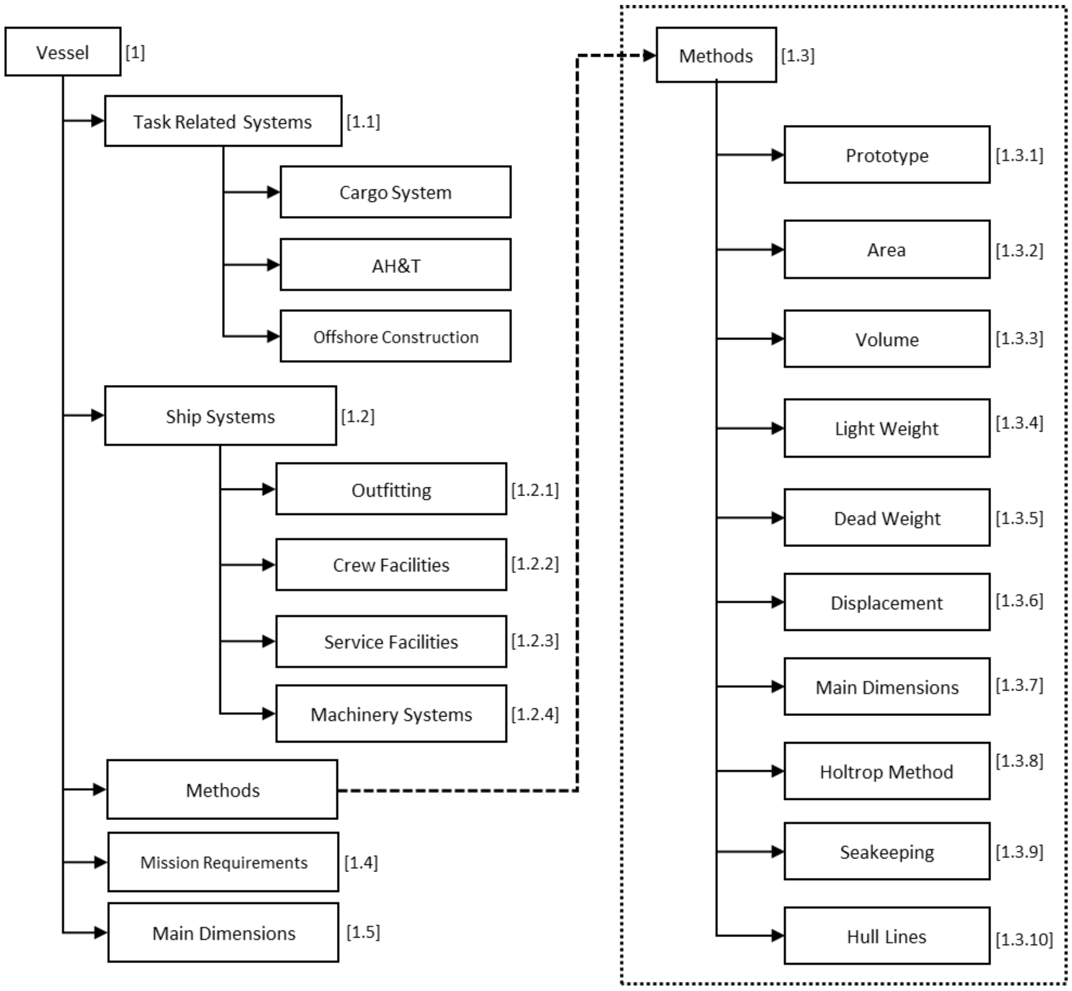


Figure 10 - Vessel prototype subdivision structure.

5.3. System-Based Ship Design Process

In order to apply the SyBSD methodology, the workflow presented in Figure 11 was developed. It includes the main design process the user should perform while applying the ship design tool library. Some of the phases relay on users' knowledge, while others relay on the developed knowledge-base.

The tools library user will have direct interaction with phases 1 2 and 7 of this workflow. The phase 1 and 2 are used for data input while the phase 7 is used by the user to check the feasibility of the obtained concept. If the answer for phase 7 is negative, the user need to redefine some systems input and reapply the tolls library.

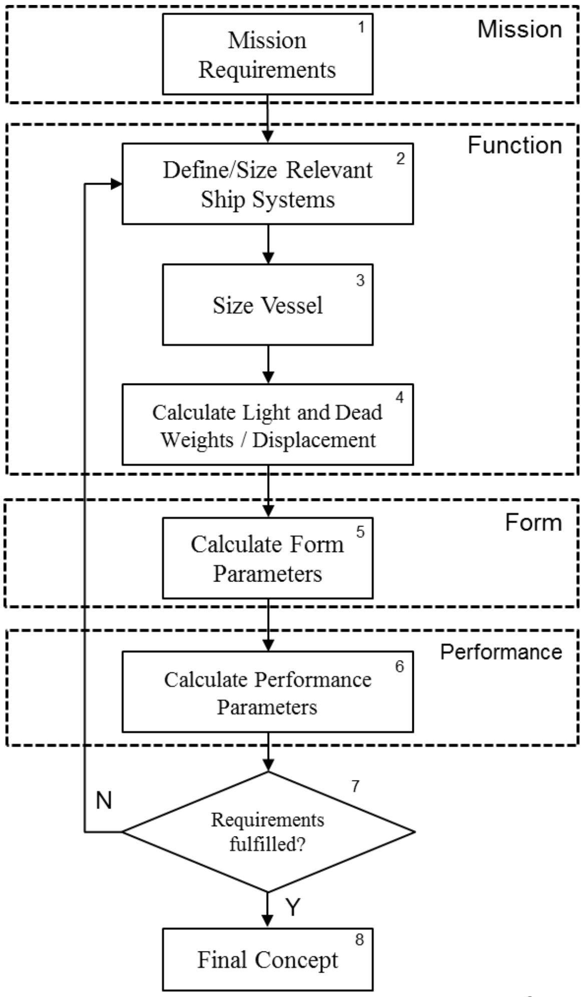

Mission
Mission Requirements
1
Function
Define/Size Relevant Ship Systems
2
Size Vessel
3
Calculate Light and Dead Weights / Displacement
4
Form
Calculate Form Parameters
5
Performance
Calculate Performance Parameters
6
Requirements fulfilled?
7
N
Y
Final Concept
8

Figure 11 - System-based ship design process workflow.

5.4. Vessel Prototype

We are going to represent the vessel using an open prototype and in order to construct it, we are going to implement the tools library using object oriented programming concepts and JavaScript language as an OS. JavaScript was not design as a conventional CBP language, but its object concept can be used to work around this issue.

5.4.1 Object Oriented Programming Implementation in JavaScript: Encapsulation and Inheritance

In order to implement the OOP in JavaScript, we are going to use to different techniques. The first one will be the encapsulation, for creating objects with specialized functionalities. The second one will be inheritance, for code reuse.

The encapsulation concept basically means to put all the inner workings of an object inside that object. To do so, we need to identify and define the properties and methods of that object, so we can apply an encapsulation pattern to construct the object. Implementing inheritance in this application will allow to inherit functionality from parent functions so that we can easily reuse code in the application and extend the functionality of objects, which can make use of their inherited functionalities and still have their own specialized methods and properties.

The best encapsulation mechanism in JavaScript is the Combination Constructor/Prototype Pattern. (Zakas, 2009) This method is not only capable of dealing with the encapsulation matter, but it is also possible to use it in order to implement inheritance through Prototypical Inheritance.

The use of encapsulation makes no sense if you just want to store some data inside an object. For this kind of task, writing the object using object literal is sufficient. But when you need to create several object with similar properties and methods, it makes sense to encapsulate all this properties and methods inside a function and use it to construct these objects.

In order to exemplify the use of Combination Constructor/Prototype Pattern technique in JavaScript, we are going to approach the implementation of the systemPrototype method, which is held by the vessel object and is used to instantiate new system objects. Each system object will contain subsystem objects instantiated by the user, following the defined SyBSD structure database. So the systemPrototype method will need to write down the instantiated subsystems as pairs key-property inside the system object. Also, each system object will have the following method: add, area, delete, input and volume. Since all system objects will have the same methods, the Prototypal Inheritance will be used to make the child system objects inherit the methods from the parent system object. Each system object will be afterward specified with the relevant properties addressed by the user.

Since we want all vessel systems to have the same methods, we can use a constructor function (class in OOP) to encapsulate these methods. In order to create the constructor function, we will use the Combination Constructor/Prototype Pattern technique.

The first step of the creation of the constructor function is to initialize the instance properties. These properties will be defined on each System instance that is created. The object doesn’t have default properties, but has a code routine responsible for getting the constructor input, searching in the SyBSD structure database for the subsystem classes and instantiate the required classes as properties. So the properties values will be different for each System, depending on the user’s input.

After the constructor is defined, the next step is to overwrite the prototype property with an object literal, where we define all the methods that will be inherited by all the System instances. By overwriting the prototype with a new object literal we have all the methods organized in one place, effectively implementing the encapsulation.

When overwriting the prototype property, we are preparing the function to provide Prototypal

Inheritance. The properties and methods added on the prototype property will be inherited by each instance of the System object, so they can use them and also receive new properties and methods.

5.4.2 Class Diagram Representation of the Vessel Prototype

For modeling the static design view of the vessel prototype the UML Class Diagram (Booch et al., 1998) is going to employed. It is the most common diagram for modeling object-oriented systems and presents the systems' classes, interfaces, collaborations and their relationships.

Using the UML class diagram the object-orient vessel prototype is modeled. The representation is crucial for presenting such a complex relational structure of all the classes composing the vessel prototype.

The representation done here is a simplified one, where some less relevant aspects of the class diagram were neglected, such as methods' arguments, relations' labeling and responsibilities' definition. The neglected aspect can be useful in several situations, but for the reason the diagram is used here (mostly to make the relations between classes clearer) they are not needed. The main objective with this simplification was to make the diagram more readable in the limited space provided by this report's pages.

The vessel prototype class diagram can be seen in Figure 13. The main class of the vessel prototype is the *Vessel* class. Its attributes and operations are specified, although the input parameters for the operations were omitted.

The *Vessel* class is composed by *System* classes. This *System* classes are related to the *Vessel* class by mean of composition relations of multiplicity zero or one, since each *Vessel* class can have zero or only one of each *System* class.

The *System* classes don't have any default attribute or operation, but they inherit operations from the class *VesselSystem*, which they are connected to by inheritance relations (which have no multiplicity).

The attributes of the *System* classes are represented by the *Subsystem* classes, which are connected to their respective *System* class by mean of composition relations of multiplicity zero or more. This happens because System classes can contain any number of their respective Subsystem classes.

All the attributes and operations of the vessel prototype were considered public, since they all need to be manipulated by the *Vessel* class.

6. Case Study

The case study is based on the PSV NAO FIGHTER. The vessel belongs to the PX121 product family (which is a medium-size class) and was designed by Ulstein Design & Solutions AS, constructed by Ulstein Verft AS and is owned by Nordic American Offshore (NAO).

In this case study, the tools library is applied aiming to attend the NAO FIGHTER's mission requirements (Table 1). The results obtained from the tools library application are compared with the real vessel parameters in order to verify how realistic (or unrealistic) the final concept is. Since the result of the tools library is only a preliminary concept, it is not expected to obtain a perfect matchup between the results, but instead, a deviance of about 10% to more or less is expected and considered fine. The library application will be done considering a subsystem division which is common for PSV vessel but can be a little bit different from the NAO FIGHTER's subdivision since its exact subdivision is unknown, but it is expected close results anyways.

Table 1 - NAO Fighter's mission requirements (Ulstein, 2016)

NAO FIGHTER Requirements	
Tunnel thruster	1
Retractable thruster	1
Azimuth thruster	2
Speed (max)	15.9 kn
Accommodation	24 POB
Deck area	850 m^2
Fuel Oil (MDO)	1474 m^3
Fresh Water	1033 m^3
Ballast water/Drill water	1676 m^3
Liquid mud (sp. gr.2,8 t/m3)	1307 m^3
Brine (sp. gr.2,5 t/m3)	1307 m^3
Cement (4 tanks)	254 m^3
LFL* (4 tanks)	153 m^3
Base oil	259 m^3

6.1. Tools Library Application

In this case study, the vessel has all the ship systems. In the task related systems, the only present is the cargo system, since this vessel is a PSV which the only attribution is to transport cargo. The system breakdown structure can be seen in (Figure 12)

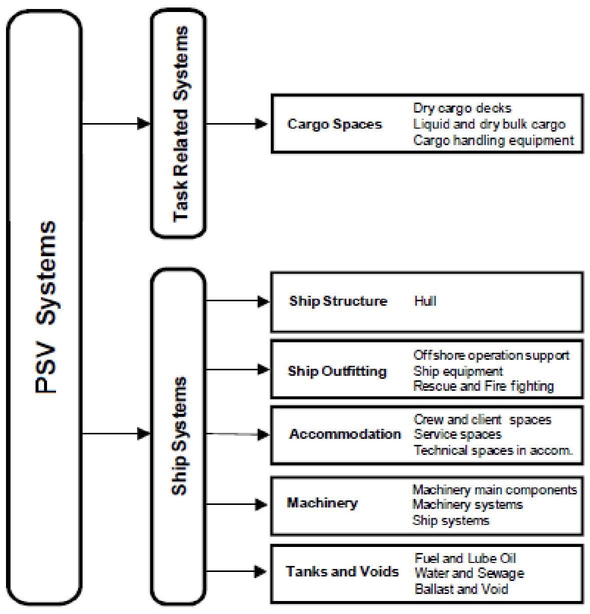


Figure 12 - NAO Fighter case study system breakdown

.

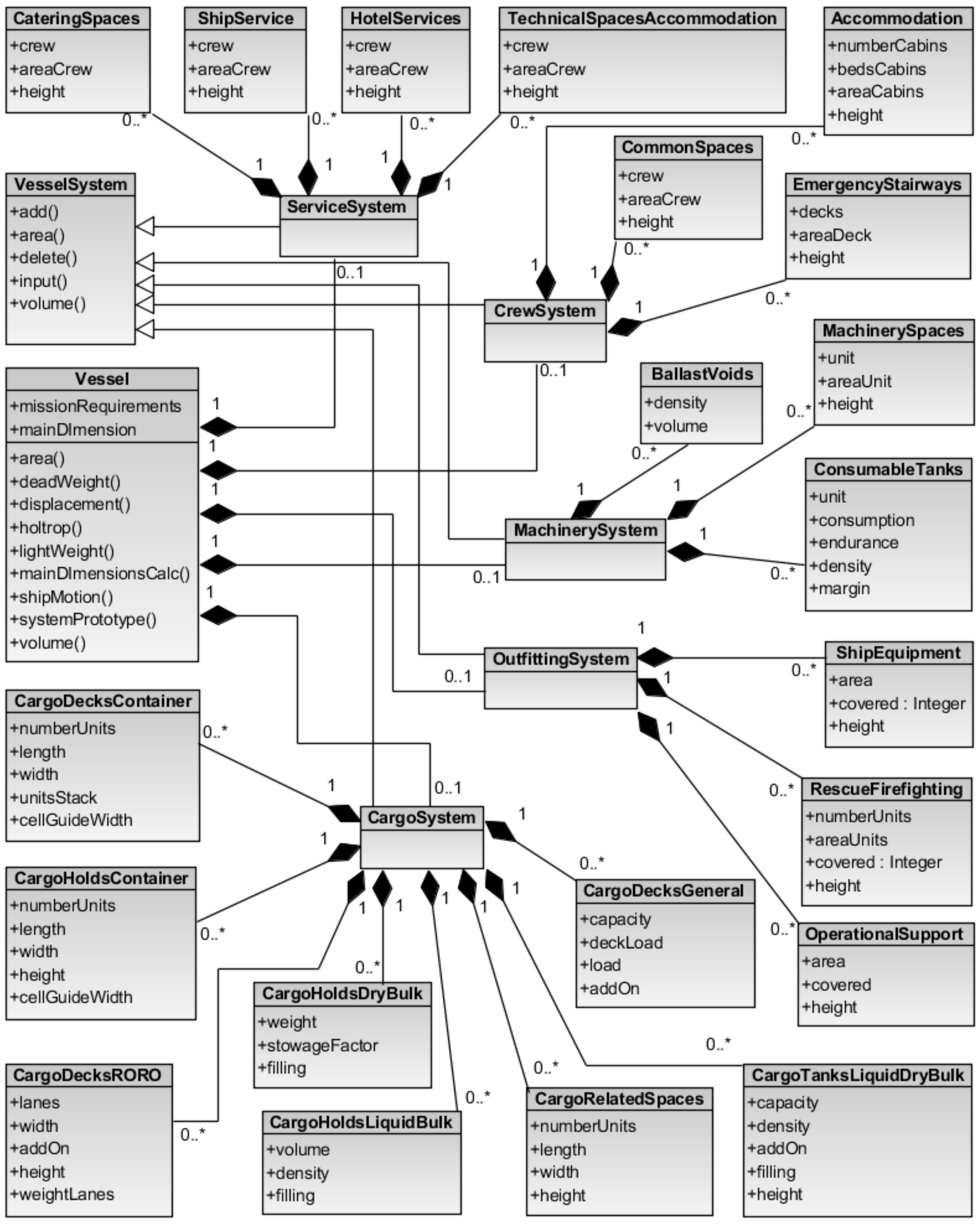


Figure 13 - Vessel prototype class diagram.

In order to construct this PSV structure, we need to begin by instantiating a new vessel object using the tools library *Vessel* constructor. This vessel object will hold all information about the ship and its systems. The input parameters are vessel type ("PSV"), cargo hold capacity (4000 ton), cargo deck capacity (2025 ton), crew size (24), vessel speed (15.85 knots), installed power (6000 kW), autonomy (1000 km) and operational area ("North Sea").

The next step consists of instantiating system objects, to hold subsystem object instances, which hold elements instances. After defined, the elements need to be initialized, insertinf the required properties. In this case study, the instatiated systems, subsystems and elements are presented below:

cargo system

cargo decks general: open cargo deck

cargo tanks liquid and dry bulk: brine and mud, fresh water, lfl, base oil, cement

cargo related spaces: transfer pumps and piping

outfitting

ship equipment: tunnel thrusters, retractable thrusters, steering gear, mooring deck forward, mooring deck aft, incinerator plant, decks stores, rope stores

rescue firefighting: fast rescue boats, life saving appliances, fire monitors

crew facilities

crew accommodation: captain class suite, officer cabin, crew single, crew double, cabin corridors wall lining

crew common spaces: mess room, officers dayroom, crew dayroom, duty mess, gymnesium, laundry linen, change room, toilets, corridors

crew emergency stairways: main stair, service stairs fore, service stairs aft

service facilities

ship service: wheelhouse, ship offices, iscp office, conference room, hospital

catering spaces: galleys, galley provision store, dry provision store, cold provision store, scullery

hotel services: linen store, ship laundry, storage spaces in the accommodation, cleaning lockers

technical spaces accommodation: ac rooms and ducting, electric substations, instrument room under wheelhouse, void spaces in deckhouse

machinery

machinery spaces: main and auxiliary engine rooms, shaftlines propellers propulsion thrusters, emergency generator and battery room, pump rooms and equipment spaces, workshops and stores, ecr and switchboard room, fire fighting system and CO2 room, engine casing, air intakes, funnel

consumables tanks: fuel oil, lub oil, fresh water, sewage and grey water

ballast and voids: ballast water

The hull subsystem doesn't need to be instantiated. It will be defined according to the required area and volume for the vessel. Once all the required systems and subsystems are instantiated, the last step in the conceptual design phase for obtaining the vessel's main dimensions is using the *mainDimensionCalc* method, held by the *Vessel* object. This method does not require any input, since all need information is gathered from the tools library database and from the instantiated systems.

6.2. Results and Analyses

The results obtained from this design example can be seen on the mainDimensions object, held by the Vessel object (Table 2). The obtained results don't show anything which raises any worried about the feasibility of the design. All the parameters are quite normal for a PSV of this size.

It is also possible to apply the *shipMotion* method to obtain an estimative of the vessel response for a specific sea state. For example, the method can evaluates the vessel response (in its center of gravity) for a sea state of wave in beam sea, with 2 m amplitude and natural period of 6.5 s. Figure 14 shows an example of movement plot from the *shipMotion* method.

Other method implemented in the library and which can be applied in this case is the *holtrop* method. It can give a rough guess about the resistance the vessel would experience while cruising in a specific speed. Figure 15 shows the expected curve of resistance for the vessel for different Froude numbers (and consequently, different speeds), although it is possible to also obtain the resistance value for a specific speed.

Table 2 - Conceptual design results – main parameters.

Beam (m)	19.39
Metacentric height above center of buoyancy (m)	6.52
Beam/Depth	2.44
Block coeff.	0.675
Center of gravity coeff.	0.74
Mid ship coeff.	0.985
Prismatic coeff.	0.685
Vertical prismatic coeff.	0.807
Waterline coeff.	0.836
Depth (m)	7.94
Froude number	0.28
Metacentric height above center of gravity (m)	1.19
Vertical center buoyancy(m)	2.73
Vertical center gravity (m)	8.06
Length/Beam	4.48
Length/Depth	10.93
LCB (%)	-0.022
Length perpendiculars (m)	84.29
Length waterline (m)	86.82
Draft (m)	4.93
Draft/Depth	0.62
Deadweight (ton)	2997.06
Displacement (ton)	5568.23
Installed Power (kW)	7789.15
Lightweight (ton)	2571.17
Slenderness ratio	4.94

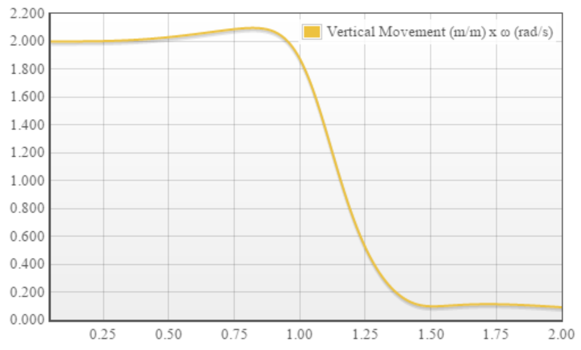


Figure 14 - Vertical motion (m/m) as function of wave frequency. Combined movement from the pitch and heave at CoG.

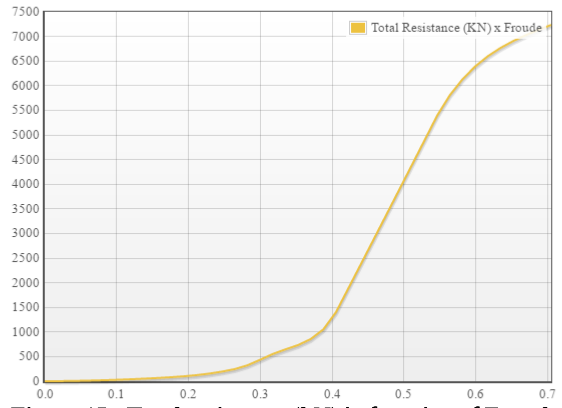


Figure 15 - Total resistance (kN) in function of Froude Number.

The results obtained from the application of the tools library can be compared with the original vessel's parameters. Table 3 presents these figures and also the deviance of the obtained values from the original design. Some parameters have no deviance at all, since they were set as required values, from which the design was built around. For the other main parameters, it is possible to verify cases where the deviation goes almost 10% up and almost 10% down.

Luckily enough, the obtained values are all inside the required range of ± 10%. If the obtained results were not as close of the real vessel as these ones, it would not necessarily indicate a failed design. As long as none of the parameters make the design unfeasible for any reason, it can be considered as a new design solution.

Table 3 - Comparison between case study and NAO Fighter parameters

	Case Study	NAO FIGHTER	Deviation
Length	84.20 m	83.40 m	1%
Beam	19.30 m	18.00 m	7%
Dead weight	2997 tonnes	3300 tonnes	-9%
Draught (max)	6.53 m	6.00 m	9%
Speed (max)	15.85 kn	15.85 kn	0%
Accommodation	24 POB	24 POB	0%
Deck area	850 m^2	850 m^2	0%

7. Concluding Remarks

After applying the open source conceptual ship design tools library in a real design problem, I concluded that the tools library can be successful used to handle the conceptual design phase of a vessel. The quality of results is strongly connected to the designer's knowledge about the vessel structure, subdivision and elements. The library also relays on an extensive database which is constructed using information from previous vessel. The quality of this database also has the potential to greatly impact the final concept, so it should be kept updated and organized in order to ensure the most trustable results.

While doing the problem statement, I have defined 3 main resource questions to investigate through this study. Regarding the first research question, the use of OOP and the concept of classes to represent vessel's systems and subsystems proved to be a very efficient way of handling the conceptual design knowledge. Having the vessel divided in subsystems and elements, which can be attributed to classes in a OOP language, works well to approach a complex design problem by dividing it in smaller and simpler problems, which can be solved individually.

At the moment, the biggest drawn back of the tools library is that it applies the SyBSD theory directly, with does not provide a mechanism to consider the systems, subsystems and elements interdependencies and interfaces. In a product with such a complex structure as a vessel, neglecting this type of relation will, inevitably, lead to design problems, especially in the more advanced design stages. Since these effects are less relevant in the conceptual design phase, the tools library can be applied to the conceptual design but should not be applied, in its current development stage, to further design phases.

When evaluating the second research question, I have noticed that the freedom and power provided by open source technology make the tools library possibilities almost endless. Since the tools library is developed in an OSS, it is free and, consequently, more accessible to end users, which can be constrained by high costs related to licensing fees of commercial software packages. Since the code is open, the tools library can be continuously improved by interested users. There is no limitation to its functionality, since when a limitation is found, it can be corrected or extinguished by the adding of new tools.

One of the strongest point in the conceptual ship design tools library is its modularity. As presented in Chapter 5, the library in composed by several design tools organized under the Vessel object's methods. As the need for different functionalities appears, new tools can be added to the library without the need to modify the functions that are already there. The new functions just need to be developed having in mind the Vessel object's hierarchy and structure. New systems and subsystem structures can also be easily added by just updating the SyBSD Structure Database. The modular characteristic makes the tools library a very flexible and powerful ship design tool.

The choice for JavaScript as an OSS was not aleatory. At first, it is a web-based programming language which is present in all the web browsers. It is easy and intuitive to code, which collaborates with its fast development and spread. Developing the library to be accessed via web browsers reduces the needs for client side software to a minimum, also reducing the computational requirements for the user's hardware. When developed for web, the tools library became accessible for any operational system and device, which also contributes to increase the application reach. The web can also be used as a platform for cooperative development, where design teams can store, share and discuss the design task. Having a tools library which supports natively the internet as a platform makes the development of this collaborative design environment easier

Finally, while investigating the third research question, the way I found to make the tools library the most generic as possible was by SyBSD theory to simplify as much as possible the systems, subsystems and elements definitions. Having design elements being represented by a handful set of parameters, considering only which is highly relevant, make these definitions as simple and generic as possible. That way, the same element structure can be reused for different elements, which also works very well with the OOP philosophy. When using a simplified definition for systems, subsystems and elements, the generic design tools library just need a well-structured knowledge base (more specifically the SyBSD Structure database) to handle the design task without any complication.

7.1. Future Work

The tools library is suitable enough for addressing the conceptual design phase, but lack support for further and more details stages of design. The capacity to store and handle data can be increased in order to make detail design a reality, especially considering the systems communication and interfaces, implementing a mix of SyBSD and Holistic design processes. This improvement in the library can be difficult to implement, since its whole structure is based on isolating systems, subsystems and elements, but definitely worth investigating in a further study.

One of the greatest points of the OSS is its fantastic community, which can work together in direction of a common goal, by helping improving the original code, suggesting and implementing changes and new features, finding and correcting bugs and so on. In this work, this aspect of the OSS was not investigated and can be a field for future study in order to further investigate how the OSS technology can impact the ship design task.

The library offers a great set of ship design functions, which can be further expanded, but miss a native support for a graphic user interface (GUI). Right now, a user need to hardcode a GUI and link it with the vessel object and its properties and methods. Having a GUI library which this integration already done and given the user the opportunity to customize it if he or she wants it, could make it more appealing for people which are not very experienced with JavaScript or with coding at all.

REFERENCES

ANDERSON, D. M. 1997. *Agile product development for mass customization : how to develop and deliver products for mass customization, niche markets, JIT, build-to-order, and flexible manufacturing,* Chicago, Irwin Professional Pub.

BERNSTEIN, J. I. 1998. *Design methods in the aerospace industry : looking for evidence of set-based practices.* M s, Massachusetts Institute of Technology, Technology and Policy Program.

BOOCH, G., RUMBAUGH, J. & JACOBSON, I. 1998. *Unified Modeling Language User Guide, The,* Addison Wesley

COYNE, R. D. D., ROSENMAN, M. A., RADFORD, A. D., BALACHANDRAN, M. & GERO, J. S. 1989. *Knowledge-Based Design Systems*, Addison-Wesley Longman Publishing Co., Inc.

CROCKFORD, D. 2008. *JavaScript: The Good Parts*, " O'Reilly Media, Inc.".

DIEROLF, D. A. & RICHTER, K. J. 1989. Computer-Aided Group Problem Solving for Unified Life Cycle Engineering (ULCE). Alexandria: Institute for Defense Analyses.

ERIKSTAD, S. O. 1996. *A decision support model for preliminary ship design.* PhD Thesis, NTNU.

ERIKSTAD, S. O. & LEVANDER, K. System Based Design of offshore support vessels. Proceedings 11th International Marine Design Conference—IMDC201, 2012.

FLANAGAN, D. 2011. *JavaScript: The definitive guide,* " O'Reilly Media, Inc.".

GASPAR, H. M. 2013. *Handling Aspects of Complexity in Conceptual Ship Design.* PhD Thesis, NTNU.

GASPAR, H. M., RHODES, D., ROSS, A. & ERIKSTAD, S. O. 2012. Handling complexity aspects in conceptual ship design. *International Maritime Design Conference.* Glasgow, UK.

GERO, J. S. & ROSENMAN, M. A. 1990. A conceptual framework for knowledge-based design research at Sydney University's design computing unit. *Artificial Intelligence in Engineering,* 5**,** 65-77.

HUBKA, V. & EDER, W. 1988. *Theory of Technical Systems,* New York, Spring.

KALYANARAM, G. & KRISHNAN, V. 1997. Deliberate Product Definition: Customizing the Product Definition Process. *Journal of Marketing Research,* 34**,** 276.

KRISHNAN, V., EPPINGER, S. D. & WHITNEY, D. E. 1991. *Towards a cooperative design methodology : analysis of sequential decision strategies,* Cambridge, Mass., Sloan School of Management, Massachusetts Institute of Technology.

KUEI, C. 1999. Categories of Free and Non-Free Software in Open Sources. *In:* DIBONA, C., OCKMAN, S. & STONE, M. (eds.) *Open sources : voices from the open source revolution.* 1st ed. Beijing ; Sebastopol: O'Reilly.

LEVANDER, K. 1991. System-based passenger ship design. *4th Int. Marine Systems Design Conference (IMSDC'91).* Kobe.

LEVANDER, K. 2012. *System Based Ship Design Kompendium*.

SIMON, E. 1996. Innovation and intellectual property protection: the software industry perspective. *The Columbia Journal of World Business,* 31**,** 30-37.

STALLMAN, R. 1999. The GNU Operating System and the Free Software Movement. *In:* DIBONA, C., OCKMAN, S. & STONE, M. (eds.) *Open sources : voices from the open source revolution.* 1st ed. Beijing ; Sebastopol: O'Reilly.

STEFANOV, S. 2010. *JavaScript Patterns*, O'Reilly Media.

ULSTEIN. 2016. *NAO Fighter* [Online]. Available: http://ulstein.com/references/blue-fighter [Accessed 1 June 2016].

URKE, T. 1976. SFI GROUP SYSTEM - A CODING SYSTEM FOR SHIP INFORMATION.

UTNE, I. B. 2009. Life cycle cost (LCC) as a tool for improving sustainability in the Norwegian fishing fleet. *Journal of Cleaner Production,* 17**,** 335-344.

VESTBØSTAD, Ø. 2011. System Based Ship Design for Offshore Vessels. Institutt for industriell økonomi og teknologiledelse.

VON HIPPEL, E. & VON KROGH, G. 2003. Open Source Software and the “Private-Collective” Innovation Model: Issues for Organization Science. *Organization Science,* 14**,** 209-223.

WARD, A., LIKER, J. K., CRISTIANO, J. J. & SOBEK, D. K. I. 1995. The second Toyota paradox: How delaying decisions can make better cars faster. *Long Range Planning,* 28**,** 129.

ZAKAS, N. C. 2009. *Professional javascript for web developers,* John Wiley & Sons.