Cover

# Table of Contents

[Table of Contents 2](#_Toc510803351)

[Abstract 4](#_Toc510803352)

[Acknowledgements 4](#_Toc510803353)

[Important Disclaimer 4](#_Toc510803354)

[Chapter 1: Introduction to the Project 5](#_Toc510803355)

[1.1 Motivation & Context 5](#_Toc510803356)

[1.2 Considerations 5](#_Toc510803357)

[1.3 Objectives 5](#_Toc510803358)

[1.4 Planning (Gantt Chart) 6](#_Toc510803359)

[Chapter 2: Background and Research 7](#_Toc510803360)

[2.1 Chess as a game 7](#_Toc510803361)

[2.1.1 Rules and difficulties 7](#_Toc510803362)

[2.1.2 Past attempts to make a chess AI 7](#_Toc510803363)

[2.2 Technical research 7](#_Toc510803364)

[2.2.1 Algorithms & Computational Techniques 8](#_Toc510803365)

[2.2.1.1 MinMax 8](#_Toc510803366)

[2.2.1.2 NegMax 8](#_Toc510803367)

[2.2.1.3 Alpha Beta Pruning 8](#_Toc510803368)

[2.2.1.4 Transposition Tables 8](#_Toc510803369)

[2.2.1.5 Monte Carlo Tree Search 8](#_Toc510803370)

[2.2.1.6 Neural Networks 9](#_Toc510803371)

[2.2.2 Design Patterns 10](#_Toc510803372)

[2.2.3 Software Engineering Practices 10](#_Toc510803373)

[2.2.3.1 Documentation 10](#_Toc510803374)

[2.2.3.2 Coding 10](#_Toc510803375)

[2.2.3.3 Debugging 10](#_Toc510803376)

[2.2.3.4 Testing 10](#_Toc510803377)

[2.2.3.4.1 Unit Tests 10](#_Toc510803378)

[2.2.3.4.2 Alpha Releases and Testers 10](#_Toc510803379)

[2.2.3.4.3 Nim 10](#_Toc510803380)

[2.2.3.4.4 Chess End Games 10](#_Toc510803381)

[2.2.4 Game API 10](#_Toc510803382)

[2.2.4.1 Description 10](#_Toc510803383)

[2.2.4.2 Original API 10](#_Toc510803384)

[2.2.4.3 Features 10](#_Toc510803385)

[2.2.4.4 Changes 10](#_Toc510803386)

[2.2.5 Other considerations 10](#_Toc510803387)

[Chapter 3: Software Requirements 11](#_Toc510803388)

[3.1 Functional Requirements 11](#_Toc510803389)

[3.2 Non-functional Requirements 11](#_Toc510803390)

[Chapter 4: Feasibility Study and Choices Taken 12](#_Toc510803391)

[4.1 Software choices 12](#_Toc510803392)

[4.1.1 Eclipse JEE 12](#_Toc510803393)

[4.1.2 GitHub 12](#_Toc510803394)

[4.1.2.1 Motivation 12](#_Toc510803395)

[4.1.2.2 Folder structure 12](#_Toc510803396)

[4.1.3 Google Drive 12](#_Toc510803397)

[4.2 Model choices 12](#_Toc510803398)

[4.3 Other choices 12](#_Toc510803399)

[4.4 Time and resources estimation 12](#_Toc510803400)

[4.5 Feasibility of the project 12](#_Toc510803401)

[4.6 Testing choices 12](#_Toc510803402)

[4.7 User interface 12](#_Toc510803403)

[Chapter 5: Design, Architecture & Diagrams 13](#_Toc510803404)

[5.1 Overview 13](#_Toc510803405)

[5.2 Class Diagrams 13](#_Toc510803406)

[5.3 Sequence Diagrams 13](#_Toc510803407)

[5.4 Other design information 13](#_Toc510803408)

[Chapter 6: Components & Implementation 14](#_Toc510803409)

[6.1 Most relevant components 14](#_Toc510803410)

[6.2 User Interface 14](#_Toc510803411)

[Chapter 7: Testing 15](#_Toc510803412)

[7.1 Unit tests 15](#_Toc510803413)

[7.2 toString() debugging 15](#_Toc510803414)

[7.3 Nim 15](#_Toc510803415)

[7.4 Chess end-games 15](#_Toc510803416)

[7.5 Playing against human subjects 15](#_Toc510803417)

[Chapter 8: Project Evaluation 16](#_Toc510803418)

[8.1 Overview 16](#_Toc510803419)

[8.2 Strengths 16](#_Toc510803420)

[8.3 Problems encountered 16](#_Toc510803421)

[8.3.1 Critical problems 16](#_Toc510803422)

[8.3.2 Other problems and solutions 16](#_Toc510803423)

[8.4 Potential improvements 16](#_Toc510803424)

[Chapter 9: Conclusions 17](#_Toc510803425)

[9.1 Overview 17](#_Toc510803426)

[9.2 Lessons learnt 17](#_Toc510803427)

[9.3 Should (not) have done 17](#_Toc510803428)

[9.4 Personal opinion 17](#_Toc510803429)

[Chapter 10: Future Work 18](#_Toc510803430)

[10.1 State of the project 18](#_Toc510803431)

[10.2 Future new features 18](#_Toc510803432)

[10.3 Improvements to existing features 18](#_Toc510803433)

[10.4 Expectations 18](#_Toc510803434)

[Chapter 11: Bibliography 19](#_Toc510803435)

# Abstract

The following report aims to describe the steps, decisions and various aspects involved in the development of a *Chess Delta* as part my Undergraduate Thesis in the past few months.

*Chess Delta* is not only a game that you can play with other people in a single computer, but also features several different AI modes.

Every aspect of the project will be discussed in the different sections of this report, including future ideas, design decisions that were made as well as flaws in the project and insights from the author on how these could have been solved or approached in different ways.

The whole project will follow best practices in Software Engineering, such as clean and properly commented code, unit and system tests, version control, documents following the pertinent IEEE standards, etc. CHECK

# Acknowledgements

Firstly, I would like to thank my family and friends for their support and patience with me during these months.

Special thanks to my father, for getting me interested in chess and teaching me how to play at a very young age, and my mother, for supporting me throughout this whole degree and project.

I am extremely grateful for all the people that believed in me since the beginning and gave me the motivation for a project of this size and level of difficulty.

I would also like to thank Dr Daniel Polani, my project supervisor, for his support and valuable advice throughout the whole process, and Dr Guy Saward, for making my stay at the University of Hertfordshire a truly pleasant experience that taught me a lot both academically and as an individual. CHECK

# Important Disclaimer

It is important to note that although the project here described IS a final version of the software, including all required features to allow subjects to play, as well as a fully functional artificial intelligence engine, some more features might be added for the presentation at my home university (Complutense University of Madrid), since the presentation date at the tribunal (if applicable) will be in July. The potential future features will be discussed in this document as well (See 10.2 Future new features). CHECK

Chapter 1: Introduction to the Project

# 1.1 Motivation & Context

For a very long time, artificial intelligence has been a prime subject of interest for me. Although this final year project is mainly a Software Engineering project, I also wanted to include Artificial Intelligence and Networking concepts as part of the it. The idea of creating a Chess game was one of the project proposals by Dr Peter Lane at the *Project Planning* module, during semester A. I found it attractive because for a long period of my life, I was an avid chess player.

An additional reason was that during my second year at the *Complutense University of Madrid*, one of the projects required for the *Computer Programming Technology* module involved using a game API (See 2.2.4 Game API) to create different board games by using inheritance in Java. Therefore, my main aim was to reuse the same API, modifying the starting code only where it was needed, since the original API was only designed to support simple games where every piece had the same value to every other. This allowed games like Tic-Tac-Toe, Connect4 or Ataxx to be implemented with ease, given their relatively simple game mechanics and rules. CHECK

# 1.2 Considerations

Chess was the game selected for the game engine, however, it was not the only game contemplated for study and development. Dr Daniel Polani suggested two alternatives to Chess in the early stages of the development.

The first game considered was *Go*, a game original from Asia that is believed to be the oldest board game still in play nowadays[[1]](#footnote-1). The reason why this game was originally considered as a better alternative to Chess, is that *Go* is a much simpler game in terms of representation since every piece is worth the same at the end of the match, and includes way less special moves (See 2.1.1 Rules and difficulties), although it features some of its own that may make the game slightly complicated to compute, such as the Ko or Suicide rules[[2]](#footnote-2). The reason why *Go* was finally dismissed was that the board design differed too much from the original game API that was already working for Chess, and because of the lack of familiarity with this game when compared to Chess. One additional reason is that in *Go*, it is much more difficult to determine the winner and it is often agreed by the players or by resignation.

The second possible alternative was *Focus*. This game was a solid alternative because of the relative ease to compute several positions, as well as only having one kind of pieces per player. The main reason why this game was discarded was, once more, the board design and rules, such as stacking pieces[[3]](#footnote-3). The initial game API offered tools to easily create and represent dimensions boards. While it could be modified, the main aim of the project was not to create and represent the board, but to create an effective artificial intelligence engine to a board game. Familiarity with the game when compared to Chess was also a determining factor. DONE

# 1.3 Objectives

…

# 1.4 Planning (Gantt Chart)

…

Chapter 2: Background and Research

# 2.1 Chess as a game

## 2.1.1 Rules and difficulties

Chess as a game has been around for almost two centuries[[4]](#footnote-4) in different versions. However, the most standard version nowadays, dates back to the final decades of the 15th century[[5]](#footnote-5). In the early years of the game, special moves such as castling or pawn opening double moves did not yet exist, but were later introduced into the game, which saw different rules (such as letting either white or black move first) but preserved the essence until the 19th century, when all of the current rules were finally fixated.

Some of these special rules increase the game complexity. Their main purpose is avoiding certain loopholes, such as the possibility of matches that would last for an infinite number of moves. While being greatly useful to enhance the possibilities and strategies in human vs human matches, they do make it a lot harder to develop artificial intelligence engines suitable for this game, as well as making it more complex to evaluate instances of the board, if all special rules are taken into account.

The main rules to take into account are Castling, En-Passant, Pawn Promotion, draw by Stalemate (formerly considered to be a win condition and still argued by some[[6]](#footnote-6)), Fifty-Move rule and Threefold repetition of position. DONE

## 2.1.2 Past attempts to make a chess AI

Chess is notorious for being a very computationally intensive game, having exponential complexity in n (on an board) to find an optimal strategy[[7]](#footnote-7). This means that for a given starting board, there is an estimate of possible different games of chess that could be legally played[[8]](#footnote-8). This number is known as *Shannon Number* and it was an attempt made by *Claude Shannon* to demonstrate that if someone were to create a computer that played all possible chess games until an end in order to make a decision on the move to make next, this computer would never be able to make a move, therefore suggesting that brute-force approaches to creating a chess engine should be abandoned. UNFINISHED

# 2.2 Technical research

Given the complexity above mentioned, creating a chess engine requires a vast knowledge of computational complexity theory, the programming language of choice as well as its libraries and different computational techniques and approaches to game theory. The following sections will cover the research done in each field needed for the development of the game and the base for the decisions and choices taken will be included in the next chapters (See 4.3 Other choices).

## 2.2.1 Algorithms & Computational Techniques

Given that the project’s aim was to explore, develop and experience various approaches to creating an intelligent chess engine, algorithms comprise a vital part of the same. In the following sub-sections, different algorithms suitable for this purpose will be analysed.

### 2.2.1.1 MinMax

MinMax (also known as MiniMax) is a decision rule that can be applied to various fields of knowledge, including statistics and philosophy, but most importantly in this case, for game theory. Its ruled by the MinMax principle which, as described by Michiel Hazewinkel, is: “An optimality principle for a two-person zero-sum game, expressing the tendency of each player to obtain the largest sure pay-off”[[9]](#footnote-9). In other words, the MinMax algorithm is used to select the best possible move to make next, in order for the opponent to be able to gain the least possible potential advantage in the following moves. Trivially, in a two-player game, minimising the opponent’s chances of winning directly increases your chances of winning (or not losing). The main drawbacks for the MinMax algorithm in many cases are that it requires a predefined evaluation function for each board state, as well as the inability to interrupt the search at any given moment, being necessary to establish the depth to be explored beforehand. One more significant disadvantage is the fact that MinMax explores every possible node, rather than being restricted to the most promising ones, which is improved partially by Alpha Beta pruning and much more significantly by Monte Carlo Tree Search. Another restriction for MinMax is that it cannot be applied to games with more than two players, although a generalisation of the MinMax theorem can be used for such cases[[10]](#footnote-10).

In the case of basic chess engines, MinMax is a good option to develop a relatively smart artificial intelligence engine with a limited amount of resources and development time. In general, a more complex but more elegant and powerful solution for this problem can be achieved by using Monte Carlo Tree Search (currently used by some of the strongest game engines, see 2.2.1.4 Monte Carlo Tree Search) or Deep Neural Networks (used by AlphaZero, the strongest game engine up to date, see 2.2.1.5 Neural Networks).

The implementation aspects of MinMax will be discussed on 6.1 Most relevant components.

### 2.2.1.2 NegMax

Although the principle of NegMax (or NegaMax) is identical to the one applied by MinMax, the main difference resides in the way that it identifies players. Whilst MinMax identifies one player as the *min* player, and the other one as the *max* player, NegMax simply changes the symbol (+, –) of the rating from the previously explored level, making it have almost exactly the same performance and results as MinMax. According to T. Anthony Marsland, “The NegaMax approach is preferred since the programming is simpler [when compared to MinMax]”[[11]](#footnote-11).

### 2.2.1.3 Alpha Beta Pruning

Alpha Beta Pruning is simply an improvement that can be applied to both MinMax and NegMax. Its main objective is to minimise the number of branches explored by not exploring further on those proven to be worse than or equal to at least one other branch, thus saving time by avoiding iterations on branches that are not going to be selected[[12]](#footnote-12). It is a very simple improvement to the algorithm that makes use of two values at each node: α represents the minimum guaranteed rating for the *max* player, whilst β represents the maximum guaranteed rating for the *min* player. If at any given node alpha becomes greater than or equal to beta, it means that the child nodes that would be created from the current one, will never be selected as optimal, thus saving time by omitting them.

### 2.2.1.4 Transposition Tables

POSSIBLY ADD IN THE FUTURE. <https://en.wikipedia.org/wiki/Transposition_table>

### 2.2.1.5 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS from here onwards) is a probabilistic method of evaluation whose aim is analysing the most promising moves by creating a random selection of possible moves and playing the match to and end given a board state. It represents a significant improvement when compared to MinMax and Alpha Beta for several reasons, although if the branching factor is increased sufficiently, the resulting tree has actually been proven to match exactly with the one created by MinMax[[13]](#footnote-13). This accurately represents the main improvement of MCTS respective to MinMax; having a lower branching factor (discarding non-promising branches early), it enables a much deeper search to be made. Some other meaningful advantages are the possibility to interrupt the search and return the best movement found so far at any point of the execution or the fact that MCTS does not require any heuristics except for the rules of the game itself, but no evaluation function whatsoever. These improvements have made this technique quintessential and widely used in all the most advanced game engines created to date. The most remarkable results have been achieved by Google DeepMind’s AlphaGo, AlphaGoZero and AlphaZero[[14]](#footnote-14) by using neural networks (See 2.2.1.5 Neural Networks) to select the most promising branches as early on as possible, thus reducing the branching factor and subsequently number of moves explored significantly.

### 2.2.1.6 Neural Networks

As mentioned while analysing the advantages and drawbacks of the MinMax algorithm, neural networks have been the strongest proven approach to creating a chess engine. They became widely known after testing AlphaZero against the strongest chess engines available at the time[[15]](#footnote-15). The results for said games were heavily in favour of AlphaZero, which after only four hours of playing against itself was able to become the strongest chess player of all times. The main difference between AlphaZero and any other previous engine, is that AlphaZero achieved the results by playing against itself, only knowing the rules of the game and not having access to heuristics or opening tables, whilst traditional chess engines used a more systematic approach to the problem, selecting results based on heuristics tailored by different experts and making use of different opening tables for the initial movements of the game13. The main difference in the concept was that, although AlphaZero also included a Monte Carlo Tree Search as part of the engine, when compared to some of the other stronger engines, it computed a vastly reduced amount of moves per second (in the order of 1/900 approximately13), selecting only the most promising branches by making use of neural networks instead of exploring as many branches as some of the other engines, such as Stockfish13. As of 2018, this has been proved to be the most effective approach to creating a chess engine.

## 2.2.2 Java Design Patterns

The use of design patterns is one of the core features of good practices in Software Engineering, therefore, it needs to be studied to determine the potential advantages and drawbacks of design patterns to later determine whether they should be applied or how to apply them to the final project. (See 5.4 Other design information)

### 2.2.2.1 Definition

Design patterns are not specific pieces of code but rather concepts. These concepts are used to portray practices that should be used to solve common problems in Software Engineering by applying similar solutions across different products with the purpose of increasing reusability[[16]](#footnote-16).

### 2.2.2.2 Advantages and drawbacks

Whilst the use of design patterns is widely accepted and encouraged in almost every case, there are also disadvantages[[17]](#footnote-17) to using them, especially the level of complexity that a project can develop if a number design patterns are used. Another good argument against design patterns was made by Peter Norvig who stated that design patterns are not necessary and are currently mainly being used to fix issues that programming languages should not have if properly designed. He also proves that most of these patterns would be simplified or non-existent in other languages such as Lisp[[18]](#footnote-18) because of the way the language naturally addresses this issues without the need to use design patterns. It should also be noted that design patterns were mainly created for Object Oriented languages such as C++ when originally conceived, and some adaptations were later published for Java[[19]](#footnote-19) and other languages.

In general, design patterns try to address certain problems that appear naturally and repeatedly during the software development process and whose solutions — without following certain patterns — would cause reusability, maintainability and modularity issues. The main reason why they are implemented is to unify proposed solutions and reduce costs by creating the ability to reuse past solutions, avoiding the repetition of the engineering process multiple times for fairly similar projects that would be costly and present little or no benefit[[20]](#footnote-20).

## 2.2.3 Software Engineering Practices

Software Engineering is defined as “the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software”[[21]](#footnote-21) by the IEEE. Therefore, it should be no surprise that every software project should follow and include certain standards and features.

### 2.2.3.1 Documentation

Every software project should be well documented to further increase reusability, reduce risks, avoid errors, plan and measure time and financial costs, among other purposes. To adapt to the software life-cycle, the IEEE created a set of standards that should be taken into consideration in every software product development[[22]](#footnote-22). The list of documents described by the IEEE is the following:

* Software quality assurance[[23]](#footnote-23) IEEE Std. 730.
* Software configuration management23 IEEE Std. 828.
* Software test documentation23 IEEE Std. 829.
* Software requirements specification23 IEEE Std. 830.
* Software verification and validation23 IEEE Std. 1012.
* Software design description23 IEEE Std. 1016.
* Software project management23 IEEE Std. 1058.
* Software user documentation23 IEEE Std. 1063.

### 2.2.3.2 Coding

POSSIBLY REMOVE

### 2.2.3.3 Debugging

POSSIBLY REMOVE

### 2.2.3.4 Testing

POSSIBLY REMOVE

#### 2.2.3.4.1 Unit Tests

POSSIBLY REMOVE

#### 2.2.3.4.2 Alpha Releases and Testers

POSSIBLY REMOVE

#### 2.2.3.4.3 Nim

POSSIBLY REMOVE

Only objective research on this game.

#### 2.2.3.4.4 Chess End Games

POSSIBLY REMOVE

## 2.2.4 Game API

### 2.2.4.1 Description

### 2.2.4.2 Original API

### 2.2.4.3 Features

### 2.2.4.4 Changes

## 2.2.5 Other considerations

Mention Go or Focus? Maybe move 1.2 here.

Chapter 3: Software Requirements

# 3.1 Functional Requirements

# 3.2 Non-functional Requirements

Chapter 4: Feasibility Study and Choices Taken

# 4.1 Software choices

## 4.1.1 Eclipse JEE

## 4.1.2 GitHub

### 4.1.2.1 Motivation

### 4.1.2.2 Folder structure

## 4.1.3 Google Drive

# 4.2 Model choices

# 4.3 Other choices

Mention rules and variants of chess.

# 4.4 Time and resources estimation

# 4.5 Feasibility of the project

# 4.6 Testing choices

# 4.7 User interface

Chapter 5: Design, Architecture & Diagrams

# 5.1 Overview

# 5.2 Class Diagrams

# 5.3 Sequence Diagrams

# 5.4 Other design information

Design patterns used

Chapter 6: Components & Implementation

# 6.1 Most relevant components

# 6.2 User Interface

Chapter 7: Testing

# 7.1 Unit tests

# 7.2 toString() debugging

# 7.3 Nim

# 7.4 Chess end-games

# 7.5 Playing against human subjects

Chapter 8: Project Evaluation

# 8.1 Overview

# 8.2 Strengths

# 8.3 Problems encountered

## 8.3.1 Critical problems

## 8.3.2 Other problems and solutions

# 8.4 Potential improvements

Things that could have been done better

Chapter 9: Conclusions

# 9.1 Overview

# 9.2 Lessons learnt

# 9.3 Should (not) have done

# 9.4 Personal opinion

Chapter 10: Future Work

# 10.1 State of the project

Why it will continue

# 10.2 Future new features

# 10.3 Improvements to existing features

# 10.4 Expectations

Chapter 11: Bibliography

1. Peter Shotwell (2008), “The Game of Go: Speculations on its Origins and Symbolism in Ancient China”, American Go Association. [↑](#footnote-ref-1)
2. British Go Association (December 2017), “Comparison of Some Go Rules”, <http://www.britgo.org/rules/compare> [↑](#footnote-ref-2)
3. Sid Sackson (1982), “Focus”, *A Gamut of Games*. Pantheon Books. Pp. 125-35. ISBN 0-394-71115-7 [↑](#footnote-ref-3)
4. Niklesh Kumar Jain (2014), “Hindi and the Origins of Chess”, <https://en.chessbase.com/post/hindi-and-the-origins-of-chess> [↑](#footnote-ref-4)
5. Hooper & Whyld (1992), “The Oxford Companion to Chess”, pp. 173-175. [↑](#footnote-ref-5)
6. Larry Kaufman (Sept 2009): “Calling stalemate a draw is completely illogical, since it represents the ultimate zugzwang, where any move would get your king taken.” [↑](#footnote-ref-6)
7. Fraenkel & Lichtenstein (1981), “Computing a perfect strategy for nxn chess requires time exponential in n”, Journal of Combinatorial Theory, Series A (31): pp. 199-214. [↑](#footnote-ref-7)
8. Claude Shannon (1950), “Programming a Computer for Playing Chess”, Philosophical Magazine. 41 (314) [↑](#footnote-ref-8)
9. Michiel Hazewinkel, ed. (2001) [1994], "Minimax principle", Encyclopedia of Mathematics, Springer Science+Business Media B.V. / Kluwer Academic Publishers, ISBN 978-1-55608-010-4 [↑](#footnote-ref-9)
10. Cai, Y. and Daskalakis, C., 2011, January. On minmax theorems for multiplayer games. In Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms (pp. 217-234). Society for Industrial and Applied Mathematics. [↑](#footnote-ref-10)
11. Marsland, T. A. (1991). “Computer chess and search”. Department of Computing Science, University of Alberta. [↑](#footnote-ref-11)
12. Edwards, D.J.; Hart, T.P. (4 December 1961). "The Alpha–beta Heuristic (AIM-030)". Massachusetts Institute of Technology. [↑](#footnote-ref-12)
13. “An Adaptive Sampling Algorithm for Solving Markov Decision Processes” Hyeong Soo Chang, Michael C. Fu, Jiaqiao Hu, and Steven I. Marcus Operations Research 200553:1, 126-139 [↑](#footnote-ref-13)
14. Silver D., Huang A., Maddison C.J., Guez A., Sifre L., Van Den Driessche G., Schrittwieser J., (...), Hassabis D. (2016) “Mastering the game of Go with deep neural networks and tree search” Nature, 529 (7587), pp. 484-489. [↑](#footnote-ref-14)
15. Silver, David; Hubert, Thomas; Schrittwieser, Julian (December 2017). “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm” pp. 4-5. [↑](#footnote-ref-15)
16. Sarcar, V. (2016). Java Design Patterns. Apress. [↑](#footnote-ref-16)
17. Patwardhan, A., 2016. Analysis of Software Delivery Process Shortcomings and Architectural Pitfalls. arXiv preprint arXiv:1607.03748. [↑](#footnote-ref-17)
18. Norvig, Peter (1998). Design Patterns in Dynamic Languages. [↑](#footnote-ref-18)
19. Alur, Deepak; Crupi, John; Malks, Dan (May 2003). Core J2EE Patterns: Best Practices and Design Strategies (2nd Edition). Prentice Hall. ISBN 0-13-142246-4. [↑](#footnote-ref-19)
20. E. Guerra Sánchez (2008/09). Técnicas de Programación (UC3M). Patrones de diseño. (Lecture slides) [↑](#footnote-ref-20)
21. Systems and software engineering - Vocabulary, ISO/IEC/IEEE Std 24765:2010(E), 2010. [↑](#footnote-ref-21)
22. “IEEE Standard for Developing Software Life Cycle Processes” in IEEE Std 1074-1991, vol., no., pp.1-, 1992 doi: 10.1109/IEEESTD.1992.101080 [↑](#footnote-ref-22)
23. Standards.ieee.org. (2018). IEEE Standards for Software Lifecycle Processes. [online] All standards available at: https://standards.ieee.org/ [↑](#footnote-ref-23)