NTNU – Trondheim
Norwegian University of
Science and Technology

Human Reliability Assessment and Software Development

Olav Alexander Lende

Master of Science in Computer Science
Submission date: June 2014
Supervisor: Tor Stålhane, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Master Thesis in Computer Science

Human Reliability Assessment and Software Development

Olav Alexander Lende
Supervisor: Tor Stålhane

Norwegian University of Science and Technology
Department of Computer and Information Science

Trondheim, June 2014

Acknowledgements

I would like to express my thanks and gratitude to my supervisor, Professor Tor Stålhane for all guidance, advise, feedback and support during the last year.

Contents

Chapter 1

Introduction

1.1 Motivation

Bugs, faults, glitches, unintended or inaccurate behaviour; Errors in software development comes in many forms and varieties. Common to most of them is that they are caused by human errors. These errors can potentially be the source of substantial cost to both developers and users, both financial and reputational. Errors in emergency systems can in a worst-case scenario contribute to the loss of human lives.

Traditional industry have long been aware of the cost and dangers of human error when working with automated systems, including e.g. in nuclear power plants and the oil industry. To mitigate the possibility of human error in these fields, there have been developed a number of Human Reliability Analysis (HRA) methods which is used within these and a number of other fields.

Seeing the place HRA methods has in other industries, the question is raised whether or not these could also be applied to software development.

1.2 Goals and Research Questions

:

G1: Study how to apply Human Reliability Assessment(HRA) to the field of software development, by studying how the HRA method SPAR-H can be applied to software development.

This is goal is to be achieved by answering the following research questions:

1.2.1 Research Questions

The primary sub-goals are for the document to:

RQ1: Are there any of SPAR-H's Performance Shaping Factors (PSF) that is more useful to the field of software development than others, based on an evaluation of SPAR-H's PSFs?

RQ2: Does SPAR-H's PSFs actually affect performance in a software development scenario, based on an experiment using SPAR-H's PSFs as independent variables?

Chapter 2

Human Reliability Assessment

2.1 Overview

2.1.1 History

Human Reliability Assessment (HRA), also called Human Reliability Analysis, is the study of finding possible human errors in cases of Human-Machine interaction and quantifying them with the intent to minimize human error proactively, or determining which role human error had or could have retroactively. It is a field which has it roots in studies of the "Human Factors Engineering" and the field of Ergonomics which first arose in the U.K. and the U.S. in the first two decades after the second world war.

The primary goal of these fields of study was to *"reduce the frequency of unwanted consequences of human errors in the operation of complex systems"* [6]. The industry at the time, and in large part industry since the start of the industrial revolution, relied on machines which in itself was designed to fulfill a certain purpose or produce a certain product and the way an operator would interacted with the machine was more often than not reliant on learning and training. I.e. the human would have to learn to fit the machine so to speak. Ergonomics and related fields, including HRA, are concerned with making the operator and the machine match in order to improve efficiency and safety.

The industry at the mid- to late 40s had been pushed to its limits during the war, and the "growing red threat" made the need for a more efficient industry apparent. First and foremost the military industry and systems, but HRA spread out to non-military appliance in the 60s when the use of nuclear reac-

tors and power plants demanded absolute minimizing of potential human error and development of routines which avoided or took into account potential human error. From there it spread to the chemical industry, other industries such as petroleum, and non-industry applications ranging from air-traffic control to NASA space programs.

HRA methods is usually divided into three generations of methods. It is important to notice the difference in generation lays in the focus of the methods as well as the time aspect, so first generation methods are not necessarily rendered redundant by the emersion of the second and third generation. To the contrary, first generation methods are considered "tried and true" and is thus often preferred over newer, less tested methods.

First generation:

> First generation methods refer to all developed before the second generation (early 90s), where new methods where developed with a different focus, including newer versions and variations to the original methods developed after 1990. The methods developed in this period, especially in the early 50s and 60s, focused on application in environments with extreme failure states such as death, more specifically, operation of nuclear reactors for military, scientific, or civilian purposes; expanding to chemical and other industries later. The common approach in these, especially early versions, is methodical breakdown of tasks into sub-tasks where the potential for human error were explored within each sub-task, or sub-task of the sub-task etc. Newer versions of these methods such as SPAR-H, which is discussed later, generalize this division into sub-tasks more based on experience and use of the older methods. Especially noteworthy methods from this generation is, THERP, HEART and SPAR-H.

Second generation:

> Second generation methods refer to methods developed with a higher focus on context and error of commission. This trend arose in the early 90s and considered to be still ongoing. However, most of these methods are rarely used used in practice. This can partially be explained by the simple fact that applying new methods always involves some degree of risk, and most of the industry that are using HRA methods are more content with what is considered known and safe. Second generation methods include CREAM and ATHEANA[1].

Third generation:

> Third generation methods refers to new methods developed with the same focus as first generation methods, but developed after the emergence of the

second generation methods. These can often be based on first generation methods, but is not considered a variation or update to an existing method. This generation is as of 2009 still considered fairly young and is not always referenced and there is *"(...) little consensus about which they are, and what they are." (Hollnagel [3])* One of the few examples of a third generation method is NARA which is based on the first generation method HEART.

2.1.2 The HRA Process

Kirwan describes the Human Error Assessment process as a flow-chart(Fig.2.1) [4]. Its components are covered within the different HRA methods/techniques. The major parts of the HRA process can be summarized in three steps: Human Error Identification (HEI), Human Error Quantification (HEQ), and Human Error Reduction (HER).

2.1.2.1 Task Analysis - Human Task Analysis (HTA)

Task analysis is an essential part of HRA as it is the process of identifying the tasks and sub-tasks where human error can occur. With few exceptions this process is performed using the Hierarchical Task Analysis (HTA) method (see [4] and [5]).

Hierarchical Task Analysis (HTA) is a top-down hierarchical method as the name implies. HTA can be summarized as dividing task down to their essential components. This is done by dividing tasks into sub-tasks which is again divided themselves if the resulting sub-tasks won't be considered too trivial, e.g "Pull lever" have the trivial sub-task "Raise hand". The sub-tasks, which are divided into hierarchical level is assigned a "plan" which indicate the order the sub-tasks is to be performed. This can be "Do in order", "Any order", or some form of algorithmic "if-then-else-then", or some other form of specified order.

2.1.2.2 Human Error Identification

Human Error Identification (HEI) is continuing the division of tasks Task Analysis did, but instead of dividing tasks into sub-task HEI is concerned with dividing tasks into possible Human Errors. This is done either by methods which goes through a checklist of regular Human Errors and the underlying related task, or by diving the task into its most fundamental components and analysis possibilities for Human Error at that level.

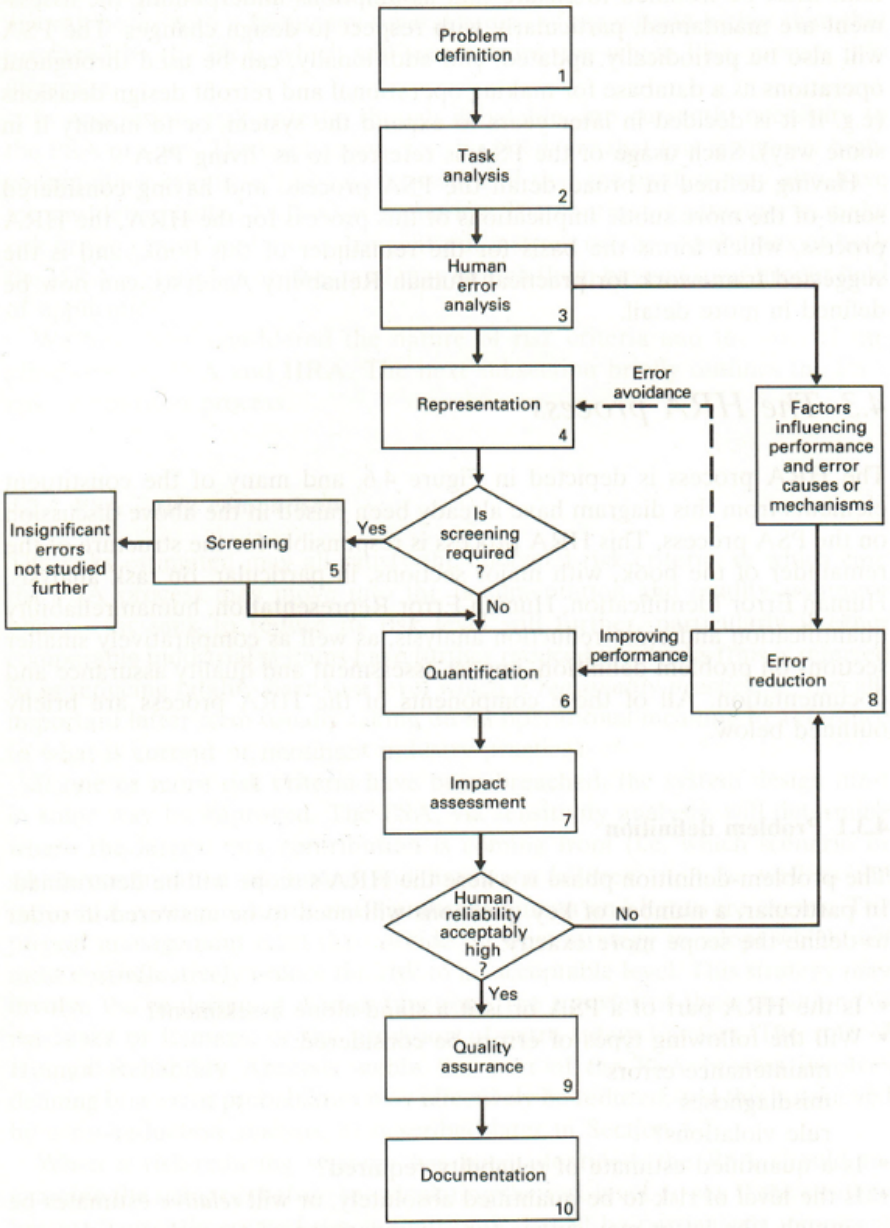


Figure 2.1: The HRA process, from [4]

2.1.2.3 Human Error Quantification

Human Error Quantification is quantifying identified Human Errors by calculating Human Error Probability (HEP). HEP is the quotient of the number of errors which occurred and number of opportunities of which error could potentially occur. Real HEP calculations are ,however, rarely used in practice or theory. Normally HEP is calculated based on standardized HEP, approximations or assumptions.

2.1.2.4 Human Error Reduction

HER is the by far the least defined step in the process, while also arguably being the most important. Based on the HEP calculated in HEQ, action or planning of action must take place to prevent Human Error from occurring, or if done retroactively, see how Human Error could have been prevented. There are different approaches to HER, but generally it boils down to "expert opinion" from an expert on the systems or processes which are being assessed.

2.2 Previous Work - Finding a compatible HRA Method

The author has previously conducted a study of how to apply HRA to the field of software development in order to create a basis for further studies (such as this thesis) as a part of a specialization project.

A part of this study included an evaluation of different HRA methods which could be applicable to the field of software development. The results of the evaluation is cited presented as:

From "TDT4501 MTDT Specialization Project: Human Reliability Assessment and Software Development" by O. A. Lende:

Recommended HRA Methods

	Generality	Usability	Validity	Recommended
HEI				
GEMS	H	M	M	
SHERPA	H	M(H)	H	X
HEQ				
THERP	M(H)	L(M)	H	
HEART	H	M	H	
SPAR-H	M(H)	H	H	X

(...)

Recommended Human Error Quantification Method

General Recommendation: SPAR-H SPAR-H is recommended modified by removing alternative worksheets which are considered unrelated to software development. It is also recommended specifiably for software development processes where the number of tasks are great and time/cost is a great factor.

(...)

This forms the basis of why we will be studying SPAR-H in this thesis.

Chapter 3

Standardized Plant Analysis Risk - Human Reliability Analysis (SPAR-H)

3.1 Overview

The Standardized Plant Analysis Risk Human reliability analysis (SPAR-H) method is originally developed for use within the context of nuclear power plants by Idaho National Laboratories while contracted by the U.S. Nuclear Regulatory Commission [2].

SPAR-H is designed to be easy to use. It covers both HEI and HEQ by assuming a set of general Human Errors related to a preset of Performance Shaping Factors (PSF). In its simplest form it consist of going over a checklist of PSF and grading their relevance based on expert opinion and pre-set PSF Levels. Finally, HEP is calculated based on which levels are chosen for each PSF:

- Available time
- Stress and stressors
- Complexity
- Experience and training
- Ergonomics (incl. human-machine interface)
- Procedures

- Fitness for duty
- Work Processes

The first three of these are included in all models, but the rest is situational. The detailed approach is show in the flowchart in Fig.3.1.

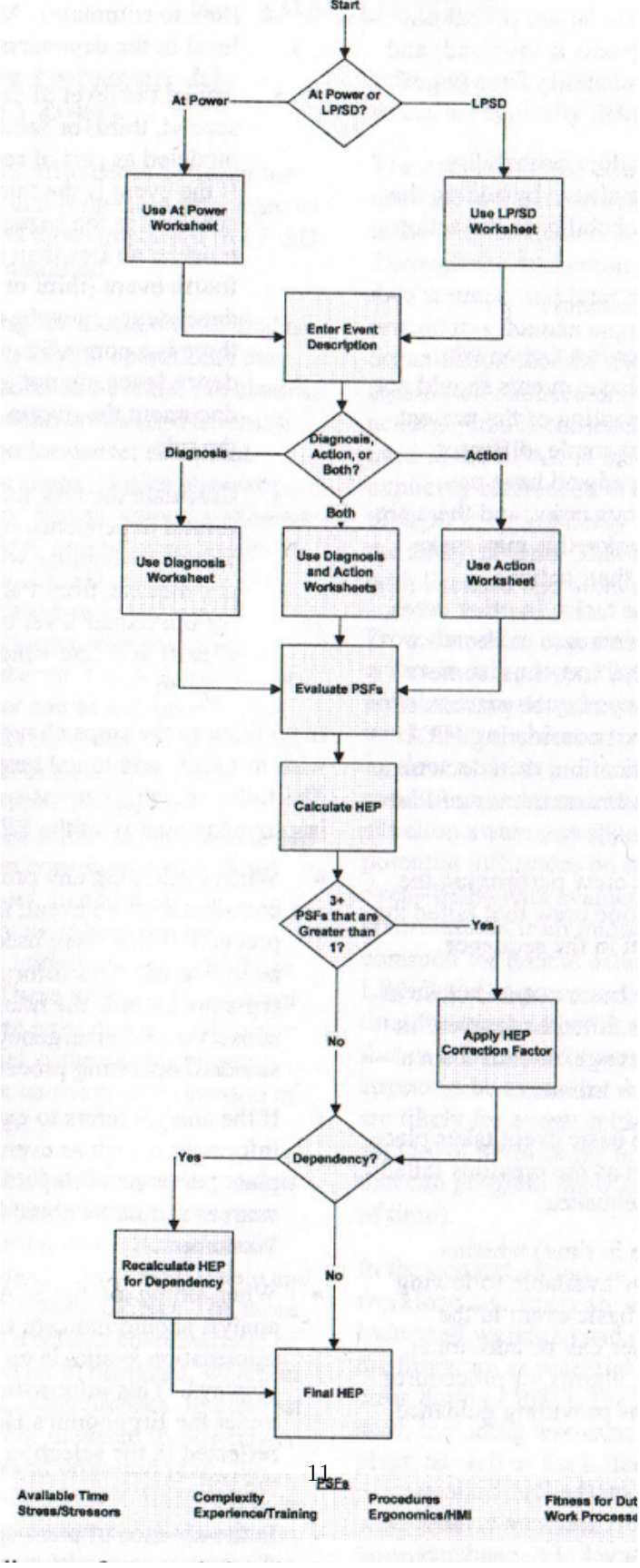


Figure 3.1: SPAR-H Flowchart[2]

One of the more interesting features of SPAR-H is that the method can be applied retroactively, i.e. trying to resolve possible reasons of "why" a specific error occurred, instead of "if".

The worksheets, such as the one shown in Fig.3.2, are general enough that they can apply to situations and PSFs within the field of software development. There is the possibility that they are too general: SPAR-H uses factors such as stress, but by rating it on a scale, thus leaving it to the assessor (the one in charge of following the method) and/or the subject in question (the "worker") to rate on notion, which might be different for another assessor/subject.

The general PSFs may, however, not be sufficient to cover all possible PSFs within the specific field of software development.

HRA Worksheets for At-Power

SPAR HUMAN ERROR WORKSHEET

Plant:_____ **Initiating Event:**_____ **Basic Event :**_____ **Event Coder:**_____

Basic Event Context:_____

Basic Event Description:_____

Does this task contain a significant amount of diagnosis activity? YES ☐ (start with Part I–Diagnosis) NO ☐ (skip Part I – Diagnosis; start with Part II – Action) Why?_____

PART I. EVALUATE EACH PSF FOR DIAGNOSIS

A. Evaluate PSFs for the Diagnosis Portion of the Task, If Any.

PSFs	PSF Levels	Multiplier for Diagnosis	Please note specific reasons for PSF level selection in this column.
Available Time	Inadequate time	P(failure) = 1.0 ☐	
	Barely adequate time (≈2/3 x nominal)	10 ☐	
	Nominal time	1 ☐	
	Extra time (between 1 and 2 x nominal and > than 30 min)	0.1 ☐	
	Expansive time (> 2 x nominal and > 30 min)	0.01 ☐	
	Insufficient information	1 ☐	
Stress/ Stressors	Extreme	5 ☐	
	High	2 ☐	
	Nominal	1 ☐	
	Insufficient Information	1 ☐	
Complexity	Highly complex	5 ☐	
	Moderately complex	2 ☐	
	Nominal	1 ☐	
	Obvious diagnosis	0.1 ☐	
	Insufficient Information	1 ☐	
Experience/ Training	Low	10 ☐	
	Nominal	1 ☐	
	High	0.5 ☐	
	Insufficient Information	1 ☐	
Procedures	Not available	50 ☐	
	Incomplete	20 ☐	
	Available, but poor	5 ☐	
	Nominal	1 ☐	
	Diagnostic/symptom oriented	0.5 ☐	
	Insufficient Information	1 ☐	
Ergonomics/ HMI	Missing/Misleading	50 ☐	
	Poor	10 ☐	
	Nominal	1 ☐	
	Good	0.5 ☐	
	Insufficient Information	1 ☐	
Fitness for Duty	Unfit	P(failure) = 1.0 ☐	
	Degraded Fitness	5 ☐	
	Nominal	1 ☐	
	Insufficient Information	1 ☐	
Work Processes	Poor	2 ☐	
	Nominal	1 ☐	
	Good	0.8 ☐	
	Insufficient Information	1 ☐	

Rev 1 (1/20/04)

Reviewer: _____

Figure 3.2: Example of a SPAR-H (AT POWER) Worksheet[2]

3.2 Performance Shaping Factors PSF

SPAR-H operates with eight different PSF. In comparison THERP have over fifty where many are highly situational, e.g. "Quality of Environment: Noise and Vibration". SPAR-H focuses on more general PSFs such as "Available Time" and "Stress" which makes SPAR-H more accessible across different industries. Kirwan describe it as late as in 2007 to be among the methods in regular use, and referred to to a recent NASA review of HRA methods where it was listed together with SPAR-H and four others as interesting, in context of possible space-missions to Mars[**?**].

3.2.1 Available Time

SPAR-H define Available Time as the time available to an individual or a team to react on an abnormal event. This includes both the diagnosis and the act itself.

Unit of Measure Available Time is measured in time units (days, hours, minutes, seconds etc.)

PSF Levels (Diagnosis) SPAR-H operates with five levels of Available Time to Diagnose which are all based on expert opinion, and/or empirical data, of what is considered to be sufficient time required to diagnose the problem at hand.

Inadequate Time	It is considered impossible to diagnose within the time available. This sets P(failure) = 1.0 which is guaranteed failure.
Barley Adequate Time	2/3 of required time.
Nominal Time	About what is considered required time.
Extra Time	From more than required time to two times required time. This level also has an additional rule: Available Time must also exceed 30 minutes independent of what is considered required time.
Expansive Time	Available Time is considered to be more than two times the required time, while also being more than 30 minutes.

PSF Levels (Action) SPAR-H operates with five levels of Available Time to Act which are all based on expert opinion, and/or empirical data, of what is considered to be minimum time required to diagnose the problem at hand.

Inadequate Time	It is considered impossible to act within the time available. This sets P(failure) = 1.0 which is guaranteed failure.
Time available is equal to time required	Available Time is considered to be about the minimum time required to perform the action.
Nominal Time	Available Time is considered to be above minimum time required with a little additional time to spare, while still being less than five times the minimum time.
Time Available $\geq$ 5x Time Required	Available Time is considered to be equal or greater than five times the minimum time required.
Time Available $\geq$ 50x Time Required	Available Time is considered to be equal or greater than fifty times the minimum time required.

Available Time is the only PSF in SPAR-H which overrides all the others, i.e. if Available time is considered to be inadequate SPAR-H considers the task impossible regardless of the other factors. Available Time however is related to other factors. Different complexity would require different time to complete. If the description of the task tells the expert analyst; it will probably take a week; and the time available is a single day, then the analysis can stop there.

In terms of software development, Available Time is a well known performance shaping factor.

3.2.2 Stress

Stress in SPAR-H is an interpretation of what the the analyst considers the level of stress put on an individual or a team based on expert opinion, human factors, and context of the problem at hand. Please note that this particular PSF is closely tied to SPAR-H's origins, i.e. Nuclear Power Plants. In other words, Stress in SPAR-H is designed to range from dismissive to extreme, in context of threats to the life of oneself and others.

Unit of Measure Stress as a PSF in SPAR-H is measured based on subjective levels based on expert opinion.

PSF Levels SPAR-H differentiate Stress into three levels:

Extreme	A level of stress so extreme there is a risk of performance deteriorating drastically. E.g. Threats to physical well-being, professional status, or career.
High	A level of stress high enough to expect decreased performance as a result. E.g. Known consequences to failure or distractions which shifts focus from the task at hand.
Nominal	A level of stress is not high enough to expect to decreased performance.

In terms of software development, stress rarely concerns itself with the physical well-being off an individual, however, whether the completion or quality of performing a task affects a group or an individual's career is arguably quite common. One could also argue that the development of high-critical systems such as air traffic control systems indeed does affect the physical well-being of others.

3.2.3 Complexity

Complexity in SPAR-H is based on expert opinion of the problem and its context in accordance to a set of factors [FIGURE p.22 2-3 spar-h].

Unit of Measure Complexity as a PSF in SPAR-H is measured based on subjective levels based on expert opinion.

PSF Levels SPAR-H operates with four levels of complexity:

Highly Complex	The task is considered very difficult to perform. The problem involves high levels of ambiguity to how it is diagnosed and/or solved.
Moderately Complex	The task is considered moderately difficult to perform. The problem involves moderate levels of ambiguity to how it is diagnosed and/or solved.
Nominal	The task is considered to not be difficult to perform. Diagnosing and solving the problem is considered to be relatively unambiguous.

Obvious Diagnosis	Diagnosis is considered to be so simple and obvious it should be trivial to the individual or team performing the task. However, this level does not cover "Obvious Action", which is considered encompassed by the "Nominal" level.

3.2.4 Experience/Training

Experience and training in SPAR-H is based on expert opinion of whether the individual or team is considered sufficiently competent to perform the task. It can take into account acquired certificates, diploma, and similar, but it also covers whether experience is expected to be sufficient.

Unit of Measure Experience/Training as a PSF in SPAR-H is measured based on subjective levels based on expert opinion.

PSF Levels SPAR-H operates with three levels of Experience/Training:

Low	Insufficient experience/training. Less than six months of experience and/or training.
Nominal	Sufficient experience/training. More than six months of experience and/or training.
High	Highly sufficient experience/training. Experience and/or training is considered to be at a level of a master. High familiarity with the task and/or similar scenarios.

3.2.5 Procedures

Procedures refer to existence and use of formal operating procedures in accordance with diagnosis and execution of a task.

Unit of Measure Procedures as a PSF in SPAR-H is measured based on subjective levels based on expert opinion.

PSF Levels (Diagnosis) SPAR-H uses five levels for the PSF of Procedures of Diagnosis:

Not Available	No formal operating procedure exists for diagnosing the task.

Incomplete	Necessary information is not covered by the procedure.
Available, but poor	Procedure is available, but does not enhance performance due to inconsistencies, incompatibility, ambiguity, or similar.
Nominal	Procedures exist and they enhance performance of diagnosis.
Diagnostic/Symptom oriented	Procedures exists that are symptom oriented (and enhance performance). Symptom oriented procedures refers to procedures which does not require a full diagnosis, just a direct response to certain symptoms. Thus allowing a faster response in critical situations, e.g. a pilot's checklists.

PSF Levels (Action) SPAR-H uses four levels for the PSF of Procedures of Action:

Not Available	No formal operating procedure exists for executing the task.
Incomplete	Necessary information is not covered by the procedure.
Available, but poor	Procedure is available, but does not enhance performance due to inconsistencies, incompatibility, ambiguity, or similar.
Nominal	Procedures exist and they enhance performance of execution.

3.2.6 Ergonomics/HMI

Ergonomics and Human Machine Interaction (HMI) is the PSF covering condition, quality, design etc. concerning the instrument used in the task. It is first and foremost concerned with the quality of a physical machine's input and output, e.g. does the instrument read correctly, or are the controls intuitive in comparison to established standards etc.

Unit of Measure Ergonomics/HMI as a PSF in SPAR-H is measured based on subjective levels based on expert opinion.

PSF Levels SPAR-H operates with four levels of Ergonomics/HMI:

Missing/Misleading	The instrument is missing, misleading, unreliable, or unintuitive to a point where operators ignore it.
Poor	The instrument functions so poorly that it effects performance negatively.
Nominal	The instrument functions as expected. It does not improve performance, but it does not decrease it either.
Good	The instrument functions so good that it positively effect performance.

In terms of software development, Ergonomics/HMI does not necessarily fit intuitively as a PSF. Software development are usually only concerned with a keyboard and mouse environment. However, one could argue familiarity and user-friendliness of development software, programming language, and the underlying operating system are HMI in a similar sense to HMI in industrial machines.

3.2.7 Fitness for Duty

Fitness for Duty is a PSF in SPAR-H which takes into account the condition of the individual or team in context of the task, i.e. expert opinion on the physical and mental state of the individual/team. This is not on a deeper psychological level, but on a level which is concerned with fatigue, sickness, known personal problems, distractions, overconfidence, drug use (alcohol) etc. The context of the task is naturally a factor here.

Unit of Measure Fitness for Duty as a PSF in SPAR-H is measured based on subjective levels based on expert opinion.

PSF Levels SPAR-H operates with three levels of Fitness for Duty:

Unfit	The individual is considered unable to perform the task.
Degraded Fitness	The individual is considered able to perform the task in some respect, but performance is to some degree degraded. E.g. an individual with a broken leg pumped up on pain killers may be technically able to perform some task, but might at

	the same time be drowsy and unable to diagnose a task properly.
Nominal	The individual is considered able to perform the task, as no factor has been found to indicate negatively affected performance.

3.2.8 Work Processes

Work Processes refer to the culture in the workplace, organization, management, polices and so forth. It is evaluated using expert opinion on whether it affect performance in a negative or positive fashion.

Unit of Measure Work Processes as a PSF in SPAR-H is measured based on subjective levels based on expert opinion.

PSF Levels SPAR-H arranges Work Processes into three levels:

Poor	Work Processes is considered to affect the performance of the individual or the team in a negative fashion. E.g. the culture in a workplace environment can be infested with misplaced loyalty in an older, proven inefficient, system for the sole sake of familiarity.
Nominal	Work processes is considered to not affect the performance significantly.
Good	Work Processes is considered to affect the performance of the individual or the team in a positive fashion. E.g. organization helps communications between different team working on different parts of the same project.

Chapter 4

Evaluation of PSFs

Each PSF is evaluated based on the criteria listed below. Each PSF is rated on a three level scale:
low (L), medium (M), or high (H):

- Relevance to Software Development
- Relation to performance
- Measurability and controllability

"Relevance" is the most important criteria, while "Relation" comes in as a close second. The reason for this is that if a PSF is barely relevant in software development, the potential performance increase or decrease from this factor is close to irrelevant. Finally, "Measurability and Controllability" is in relation to low budget experiments. For instance, "Stress" is very difficult to measure directly without proper medical equipment and expertise. Additionally, simulating degrees of "Stress" is difficult at lower degrees as it often is very subjective.

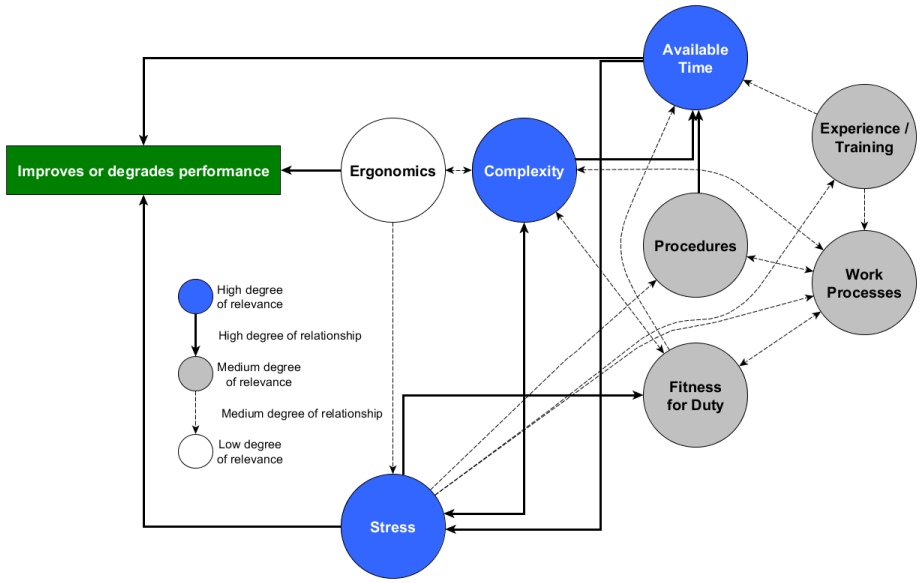


Figure 4.1: PSF Relations

Available Time	Relevance	H
	Relation	H
	Measurability	H

Available Time is highly relevant to any type of development process, including software development. In terms of relation to other PSFs, Available Time is the among the most important as shown in FIG.4.1. It is also easy to control.

Complexity	Relevance	H
	Relation	H
	Measurability	M

Complexity is as important as Available Time and they are directly related. I.e. the more complex a task is, the more time is required to perform it properly. However, what qualifies as complex is difficult to assess on its own without involving other PSFs, such Experience, which can be difficult to assess in it's own right. Complexity is not directly related to performance, but is directly to both Available Time and Stress which are directly related to performance.

Stress	Relevance	H

	Relation	H
	Measurability	L

Stress is also an important factor in software development, and has strong ties to Available Time and Complexity. However, it is difficult to control since what is considered stressful or not can vary dramatically between individuals. True unbiased measurements of stress, as in chemicals released into the blood stream, require advanced medical knowledge and equipment.

Experience	Relevance	H
	Relation	M
	Measurability	M

Experience with the problem at hand is relevant to software development, but only with a medium degree relationship to Available Time.

Procedure	Relevance	M
	Relation	H
	Measurability	M

Procedures within software development exists for the overall development process in the form of development models such as waterfall or Scrum. On lower levels of development, i.e. writing code etc., there are rarely any form of formal procedure. A possible exception would be checking an existing library for functions and pre-written code, but this would be situational depending on the environment. Procedure does have a strong relation to Available Time, but as with Experience it does not directly affect performance.

Ergonomics	Relevance	L
	Relation	H
	Measurability	H

Ergonomics has little relevance to software development as man-machine interactions tend to be limited to standardized keyboard-mouse-monitor interfaces.

Fitness for Duty	Relevance	M
	Relation	M
	Measurability	L

Fitness for Duty has some relevance to Software Development, but has only some relation to Complexity and Available Time.

Work Processes	Relevance	M
	Relation	L
	Measurability	L

Work Processes do have some relevance, but has the weakest relation to other PSFs.

4.1 Selected Performance Shaping Factors

	Relevance	Relation	Measurability	Recommended
PSF				
Available Time	H	H	H	X
Complexity	H	H	M	X
Stress	H	H	L	X
Experience	H	M	M	
Procedures	M	H	M	
Fitness for Duty	M	M	L	
Work Processes	M	L	L	
Ergonomics	L	M	M	

Based on the evaluation "Available Time", "Complexity", and "Stress" is chosen as three PSF which will be the main focus of the experiment. However, as "Stress" is rated low in "Measureability and Controllability" it won't be used directly as an independent variable. It is, however, an important PSF which theoretically is influenced heavily by both "Complexity" and "Available Time". It will therefore be used as a dependent variable measured by the students own experience of stress. This is not as reliable or objective as measuring stress using medical equipment, but it will give some indication of whether the students experienced stress and how the degree of the reported stress compares to the achieved result.

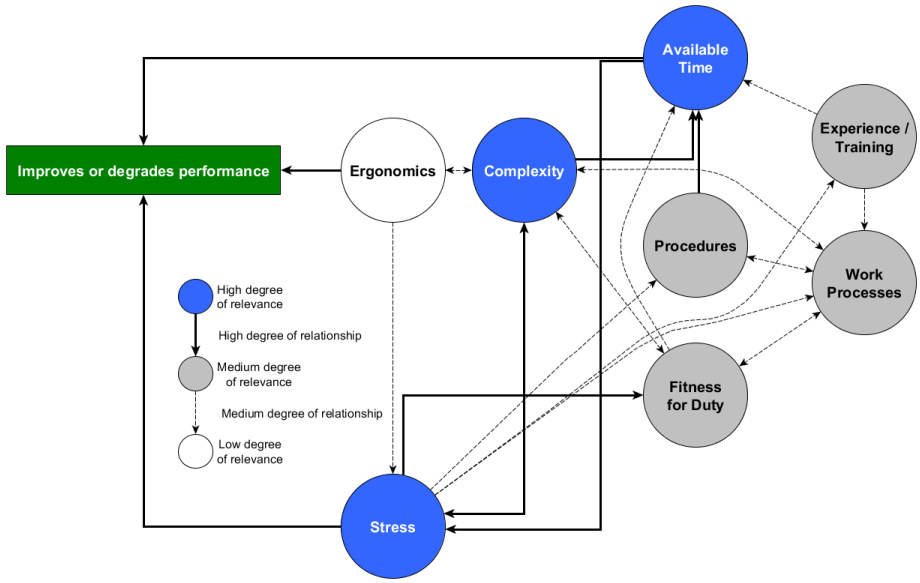


Figure 4.2: SPAR-H's PSFs

Chapter 5

Experiment

5.1 Introduction

If we assume SPAR-H can to be used within the field of Software Development, then the most relevant and influential PSFs used in SPAR-H needs to be influential to performance in Software Development. In order to investigate whether this is the case we will perform an experiment using student voluntaries. The student voluntaries are expected to be 2nd graders studying Computer Science at NTNU.

The experiment itself will be low budget as the students will get paid for their time, but will provide their own equipment (Laptops). The focus of the experiment will be to investigate differentiating results using selected PSFs used in SPAR-H as independent variables. The selection of PSFs is based on the evaluation in Section 4.

5.2 Design

This is a summary of the design choices used when planning the experiment.

Context

The context is an experiment with thirty-eight 2nd grade students of Computer Science at NTNU as subjects. The experiment is low budget, where the students are paid for their time while providing their own equipment (Laptops). The experiment itself consist of letting the students solve software development tasks based on modified exam problems from the NTNU course TDT4100 Object-oriented Programming.

Subjects

The subjects, i.e. the students, will be divided into two-person groups using Single random sampling. There are 38 participants, which leaves about 19 groups.

Variables

Based on the evaluation we use the following variables:

Independent variables

- Time
- Complexity

Time will be the available time the groups have to work on the provided problems and Complexity the total amount of problems to be solved.

Dependent variables

- Completion
- Perceived Stress
- Perceived Available Time
- Perceived Difficulty/Complexity

Completion is the result a group gets, based on their completion of the task they where given, measured in percent. The completion is measured based on a score of points divided by total amounts of points available.

Perceived Stress is the subjects own rating of experienced stress on a scale form 1 to 10 where higher, is higher more stressful. Additionally, the subject will also report whether they felt that the stress affected their performance positively, negatively, or both positively and negatively. This is because SPAR-H operates with stress as a negative factor, and whether this holds true in software development scenarios (where stress rarely reaches SPAR-H's life-threatening degree of stress) is essential in regard to the PSF "Stress" can be considered relevant in a software development environment.

Perceived Available Time and Complexity is the subject's own rating of how they individually perceived the constraints set by the independent variables. SPAR-H operates with terms such as "Extra Time" and "Inadequate Time". Thus while the constraints set by the independent variables may be objectively more or less sufficient, the dependent variable Perceived Time is used to measure the students' subjective experience of whether there was sufficient or insufficient time available.

Hypothesis

We operate with four hypotheses for this experiment:

$\mathbf{H}_0$

The null hypothesis.

$\mathbf{H}_1$

Available Time will significantly effect the results, but not Complexity.

$\mathbf{H}_2$

Complexity will significantly effect the results, but not Available Time.

$\mathbf{H}_3$

Both Available Time and Complexity significantly effect the results.

Design Type

Two factors with two treatments gives us a 2*2 factorial design.

Instrumentation

- A survey (See Appendix A)
- Two sets of tasks (one more complex than the other) (See Appendix B and C)
- Laptops (provided by the students)

Survey

The survey sheet (Appendix A) is the main method used to collect the majority of the dependent variables. After the student groups are either done with the provided task or their set time have run out, they deliver their work and are handed out a survey to answer individually.

The survey asks the student to rate, on a scale from 1 to 10 where higher is more, how they found:

- The Difficulty of the task provided
- Available Time was sufficient for performing the task provided

- Stress effected performance

The students are also asked if they feel stress effected their performance:

- Positively
- Negatively
- Both/Don't know

Additionally, the survey provide students with space to comment their rating, explaining if anything unexpected or unforeseen effected their performance, and space for any additional comment.

Task sets

There are two set of tasks, Task A (Appendix B) and Task B (Appendix C). The difference between the two tasks are that B have additional problems to solve. This is done to give B a higher degree of complexity. Complexity can be difficult to assess or compare as different individuals may have different opinions as to what they consider lesser or more complex. However, we can assume that one task will be objectively more difficult than another task if the first task is the same as the second task with additional things to do within the same limit of time.

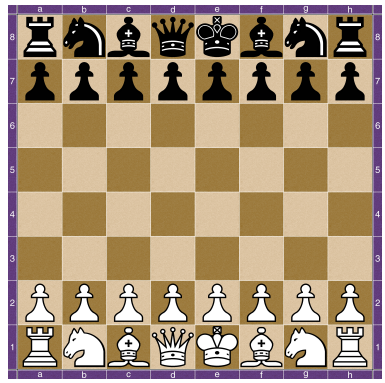


Figure 5.1: The tasks are centered around moving pieces in chess

The set of tasks are modified versions of sub-tasks from "Task 1" and "Task 2" from *"Final Exam in TDT4100 Object-oriented Programming June 6th 2008"* by Hallvard Trætteberg. The original exam was written in Norwegian and as the students are expected to know Norwegian, the modified versions are also in Norwegian. The software development part of the exam is programming tasks such as (translated):

```
b) Implement the Board class with the internal table
   and the two following functions:

    Piece getPiece(String position) { ... }
```

```
void setPiece(String position, Piece piece) { ... }

* getPiece returns the piece from the square labeled with position,
  or null of the square is empty.
* setPiece places the piece "piece" in the square labeled with position
```

This example is among the easier tasks, but it is time consuming. The other part of the original exam tasks are theoretical questions related to the course which requires a written answer. These are removed in the modified version as it is not the intention to test the students theoretical knowledge.

The original exam was written for pen and paper and the two tasks Task A and B are based upon constituted 40% (Task 1) to 65% (Task 1 + Task 2) of a four hour exam. I.e. the original exam expected these unmodified tasks to take about 96 minutes (1h 36min) to 156 minutes (2h 36min) when solved solely on paper by an individual. Taking into account that the tasks are to be solved with pair-programming; in a proper programming environment (contrary to pen and paper); by students who has an additional year of programming experience since they had their own exam in TDT4100; it is assumed the students require less time to solve the programming tasks, even with the added modifications and extensions. It is not the intention that most groups should be able to complete the all the tasks within the given time frame. I.e. the experiment is more interested in how far groups are able to get with a certain amount of time to work on a task with a certain complexity. If a large portion of the groups are done before the given time, especially if their done within the shortest time frame, the result would not reflect the difference between the smaller and the larger time frame. Therefore the two time frames, which are the two treatments to the independent variable "Time", are set to 60 minutes and 90 minutes which are about 60% of the time given to solve the unmodified versions of Task A (Task 1 from the original exam) and Task B (Task 1 + Task 2 from the original exam) respectively.

Each student groups are thus given one of four tasks which represent a given combination of the independent variables:

Less time + Lower complexity Task A with 60 minutes (A60)

Less time + Higher complexity Task B with 60 minutes (B60)

More time + Lower complexity Task A with 90 minutes (A90)

More time + Higher complexity Task B with 90 minutes (B90)

5.3 Execution

Date: 2014-03-26

Time: 14:15-16:00

Location: Auditorium S2, Campus Gløshaugen, NTNU

Participants: 38 second year students of Computer Science at the Department of Computer and Information Science

The students were first given a short presentation of the agenda:

- They would be divided into groups of two
- Each group would be given one of two tasks, marked with either A or B.
- There would be two different time limits; Marked on the front of the task as either 60 minutes or 90 minutes.
- No communication with other groups was allowed.
- The programming language to be used was Java. However, they were free to use whatever development environment they felt most comfortable with, e.g. Eclipse.
- When the provided time was over: The written code should be set to a provided e-mail address marked with task name (A or B), time frame (60 or 90), and group name; Each students would then be required to answer the survey before confirming presence by signature (for payment).
- Practical information regarding payment for participation.

The students were then divided into groups of two and the groups were spread out in the auditorium to avoid communication. Additionally, the observer remained in the auditorium at all times to further prevent communication between groups.

When the experiment started a representative from each group received a task from the observer, the timer was started, and the groups started working. Due to there only being 38 participants, 40 were expected, one of the task sets, Task B with 60 minutes (B60), was only given to four groups, whereas the other three was given to five.

The groups were given two announcements by the observer of remaining time left during the experiment, at 15 minutes and 5 minutes left for each of the two time limits. After the experiment a short reminder of how to deliver and to fill out the survey was also given.

Chapter 6

Analysis and Discussion

6.1 Processing Data

The raw output of the experiment, the gathered data, consisted of two separate sets of data:

- Compressed .zip-folders with collected code, one from each group.
- Answered surveys, one from each student.

The delivered code will be used to calculate a score for each group which represent the amount and quality of the work done. This score is what will represent the experiments dependent variable "Completion". The dependent variables "Perceived Stress", "Perceived Available Time", and "Perceived Difficulty" on the other hand are gathered from the survey.

6.1.1 Evaluating Code

The evaluation of the delivered code was done in two parts. First the tasks were divided into sub-tasks of approximately equal size. This size generally entail the creation of separate functions. For example, this task:

```
b) Implement the Board class with
the internal table and the two following methods:

Piece getPiece(String position) { ... }
void setPiece(String position, Piece piece) { ... }

(...)
```

is divided into two sub-tasks: “b) getPiece” and “b) setPiece”. The complete list of sub-tasks can be seen in Table D.2.

Using this list each sub-task for each group is assigned points based on the following criteria:

0 Points Nothing, or practically nothing, is done.

1 Point Incomplete/ Not working, this is given to tasks that are either incomplete, not working due to bugs or not functioning in the manner specified by the task description.

3 Points Complete, the code is implemented and working as the task specifies.

The difference in points between “Incomplete” and “Complete” is there to reward functioning code over non-functioning code.

Originally the intention was to assign the “Nothing Done” by hand and differentiate between “Complete” and “Incomplete” by asserting the majority of the remaining code with a pre-written Java program (Appendix E): Asserter.java). In theory this would work as all written functions from the tasks had to follow a specified format with the same name, return type, parameters, such as:

```
boolean isStraight(String from, String to){...}
```

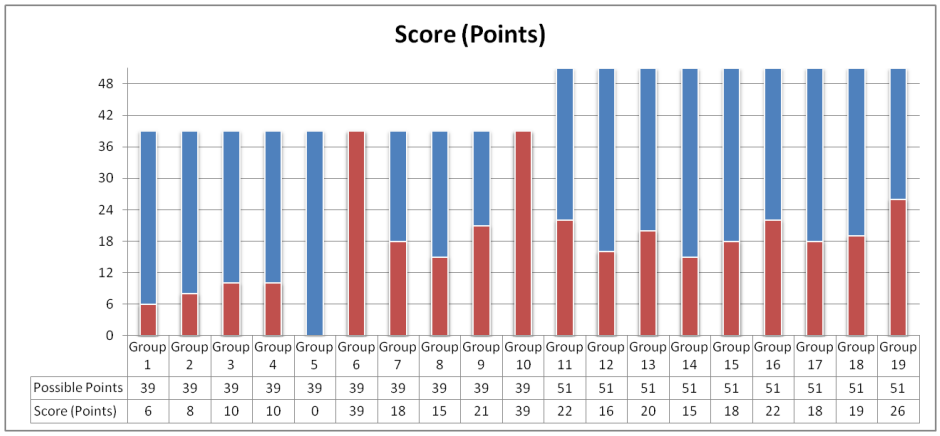
which could easily be asserted with:

```
private static void assertC(){

Board board = clearBoard(new Board());;

assert(board.isStraight("a1", "a2"));
assert(board.isStraight("a2", "a4"));
assert(board.isStraight("b3", "b6"));
assert(board.isStraight("c5", "c8"));
assert(board.isStraight("a1", "b1"));
assert(board.isStraight("c3", "f3"));

(...)
```

However, after working with the code from a couple of different groups it became apparent this too had to be done by hand as many groups failed to

follow the specified format, for instance by adding their own parameters or using different return types.

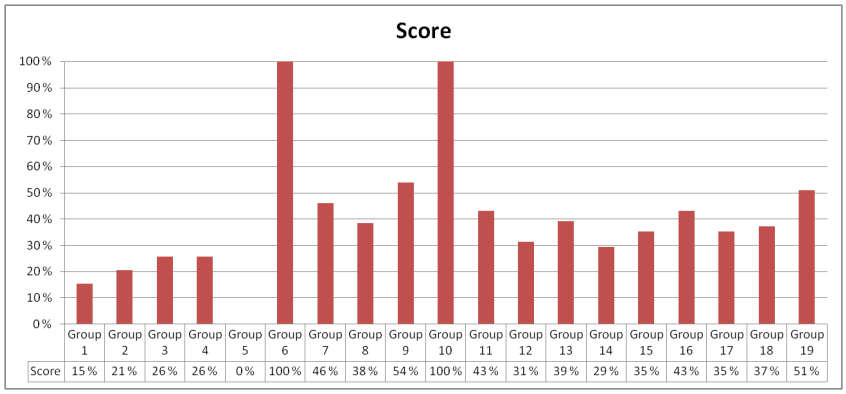
The points from each sub-task was then combined giving each group a score in points as shown in red in Figure. 6.1.



	Group 1	Group 2	Group 3	Group 4	Group 5	Group 6	Group 7	Group 8	Group 9	Group 10	Group 11	Group 12	Group 13	Group 14	Group 15	Group 16	Group 17	Group 18	Group 19
Possible Points	39	39	39	39	39	39	39	39	39	39	51	51	51	51	51	51	51	51	51
Score (Points)	6	8	10	10	0	39	18	15	21	39	22	16	20	15	18	22	18	19	26

Figure 6.1: Score (Points)

The score is then calculated as percent to reflect completion of the given task (Task A or B) based on the total amount of points in available in each task, 39 points and 51 points respectively which is shown in blue in Figure.6.1. This results in the scores shown in Figure.6.2.



	Group 1	Group 2	Group 3	Group 4	Group 5	Group 6	Group 7	Group 8	Group 9	Group 10	Group 11	Group 12	Group 13	Group 14	Group 15	Group 16	Group 17	Group 18	Group 19
Score	15%	21%	26%	26%	0%	100%	46%	38%	54%	100%	43%	31%	39%	29%	35%	43%	35%	37%	51%

Figure 6.2: Score (Percents)

6.1.2 Survey

The survey is the source of the data on the dependent variables for perceived stress, time, and difficulty. The three values are graded by the students answering the survey on a scale from 1 to 10 where higher is more, i.e. a score of 10 would indicate that; stress effected performance to a high degree; available time was very sufficient; and the task was very difficult.

The rating each student gave is summarized in Figure. 6.3, Figure. 6.5, and Figure. 6.4. Additionally, the students were asked to rate how they felt stress effected their performance. This is shown in Figure.6.6.

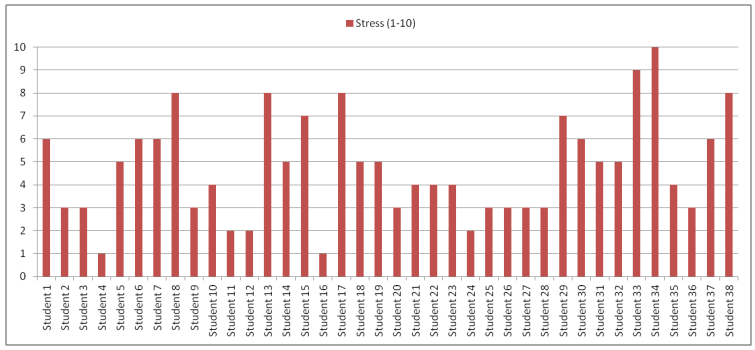


Figure 6.3: Stress

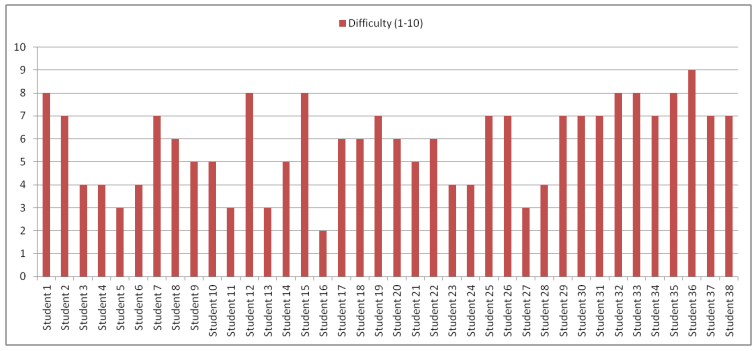


Figure 6.4: Difficulty

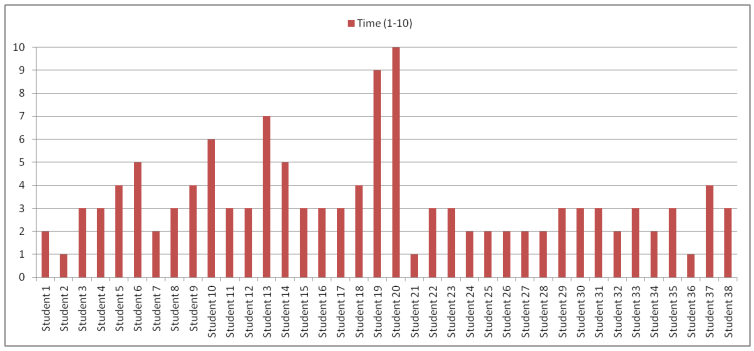


Figure 6.5: Time

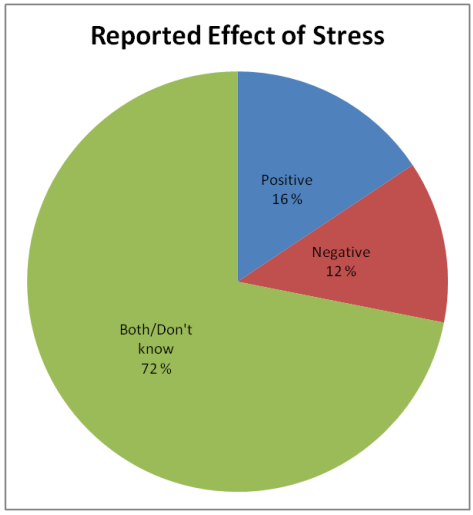


Figure 6.6: Reported Effect of Stress

6.1.3 Data Summary

Table 6.1: Raw Results

Task	Group	Score	Student	Difficulty (1-10)	Time (1-10)	Stress (1-10)	Stress Effect Comment
A60	1	15 %	1	8	2	6	Negative
			2	7	1	3	Negative
	2	21 %	3	4	3	3	Both/Don't know
			4	4	3	1	Both/Don't know
	3	26 %	5	3	4	5	Both/Don't know
			6	4	5	6	Both/Don't know
	4	26 %	7	7	2	6	Both/Don't know
			8	6	3	8	Both/Don't know
	5	0 %	9	5	4	3	Positive
			10	5	6	4	Both/Don't know
A90	6	100 %	11	3	3	2	Negative
			12	8	3	2	Both/Don't know
	7	46 %	13	3	7	8	Positive
			14	5	5	5	Both/Don't know
	8	38 %	15	8	3	7	Both/Don't know
			16	2	3	1	Positive
	9	54 %	17	6	3	8	Positive
			18	6	4	5	Positive
	10	100 %	19	7	9	5	Positive
			20	6	10	3	Positive
B60	11	43 %	21	5	1	4	Both/Don't know
			22	6	3	4	Both/Don't know
	12	31 %	23	4	3	4	Both/Don't know
			24	4	2	2	Both/Don't know
	13	39 %	25	7	2	3	Both/Don't know
			26	7	2	3	Both/Don't know
	14	29 %	27	3	2	3	Both/Don't know
			28	4	2	3	Both/Don't know
B90	15	35 %	29	7	3	7	Both/Don't know
			30	7	3	6	Both/Don't know
	16	43 %	31	7	3	5	Both/Don't know
			32	8	2	5	Both/Don't know
	17	35 %	33	8	3	9	Negative
			34	7	2	10	Negative
	18	37 %	35	8	3	4	Both/Don't know
			36	9	1	3	Both/Don't know
	19	51 %	37	7	4	6	Positive
			38	7	3	8	Both/Don't know

6.2 Analyzing Data

6.2.1 Exclusions

Among the groups scores (see Figure.6.2) there are 3 scores which strikingly sticks out from the others. These are the scores from Group 5, Group 6, and Group 10; which scores are 0%, 100%, and 100% respectively.

Group 5 did deliver a folder with some of the Java classes, but all the files where empty. Their comments in the survey insinuates they where not done sub-task d) (Implying they were done with a), b), and c)). It is thus assumed that this is a case of delivering the wrong files and all data from Group 5, both from code and survey, are excluded from further analysis.

Group 6 and Group 10 both managed get a perfect 100% score. This is extraordinary considering considering the total average among all groups are 36% and the highest scoring group after them have a 54% score. It is assumed this points to one of two explanations, significantly higher skill than other groups or plagiarism.

In regards to the skill explanation, this is a definite possibility. Many of the tasks which to an experienced programmer seems trivial was not performed correctly by many of the groups. For instance, sub-task a) asks the groups to create an enum with two values and a function which returns the non-selected value. Five of eighteen (excluding Group 5 as mentioned above) did not manage to solve this task completely. However, while this experiment uses second year students as subjects, it is not a goal of the experiment to observe difference in skill between between them. On the contrary, significant difference in skill among the subjects, either positively or negatively, is undesired in such a small pool of just 38 subjects.

In regards to plagiarism the original exam is not mentioned or referenced in Task A or Task B which were handed out to the students. However while the tasks are modified and expanded upon to some degree, the solution set of the original exam is available publicly (if one is looking for it) and would outright provide the solution for some of the tasks and a similar way to solving others. So when one of the students in these two groups mentions remembering this particular exam from when he practiced his own exam in the same course there is at least reason for concern.

As both possible explanation are undesirable both groups and data related to them are excluded from further analysis.

6.2.2 Relations between Time and Complexity

Due to the already small sample size which is further divided into four groups based on Time-Complexity combinations (A60, A90, B60, and B90), all further analysis will be primarily concerned with the averages of the different dependent variables over different groups, as shown in Table 6.2.

Averages	A60	B60	A90	B90	All
Score	22 %	36 %	46 %	40 %	36 %
Perceived Stress	4,75	3,25	5,67	6,30	5,03
Perceived Time	2,88	2,13	4,17	2,70	2,88
Perceived Difficulty	5,38	5,00	5,00	7,50	5,88

Table 6.2: Averages

First perform two sample t-tests on all the combinations of averages of the dependent variables for the time-complexity combinations. The resulting p-values are shown in Table 6.3, while more details are found in Appendix D.

P-values(t-Tests)		Score	Perceived Stress	Perceived Time	Perceived Difficulty
A60	A90	0,017	0,511	0,136	0,742
A60	B60	0,014	0,110	0,161	0,664
A60	B90	0,002	0,164	0,738	0,004
A90	B60	0,132	0,073	0,026	0,049
A90	B90	0,343	0,636	0,076	0,017
B60	B90	0,332	0,002	0,115	0,115

Table 6.3: p-values (t-tests)

Further, we calculate the main effect of Complexity and Time and their interaction effect for all dependent variables by solving the following design of experiments (DoE) table:

	Complexity	Time	CxT	Dependent Variable
A60	-	-	+	AA60 (Average for A60)
A90	-	+	-	AA90 (Average for A90)
B60	+	-	-	AB60 (Average for B60)
B90	+	+	+	AB90 (Average for B90)
Effects	Main Effect(Complexity)	Main Effect(Time)	Interaction Effect(Complexity x Time)	

Table 6.4: DoE Table

Using this table, we calculate the main effect for Complexity:

$$Effect(Complexity) = \frac{-AA60 - AA90 + AB60 + AB90}{2}$$

where AA60 represent the average of a dependent variable's value over A60, and AB90 over B90 etc. Similarly, we calculate the main effect of Time and the Interaction effect:

$$Effect(Time) = \frac{-AA60 + AA90 - AB60 + AB90}{2}$$

$$Effect(ComplexityxTime) = \frac{+AA60 - AA90 - AB60 + AB90}{2}$$

This calculates the Marginal Means for time and complexity and subtracts the mean for the lower time/complexity values from the mean for the higher higher time/complexity values to find Main effect. The interaction effect is found by calculating the cross product of Complexity and Time.

The resulting effects are shown in Table 6.2.2.

	Complexity	Time	CxT	Score	Perceived Stress	Perceived Difficulty	Perceived Time
A60	-	-	+	22 %	4,75	5,38	2,88
A90	-	+	-	46 %	5,67	5,00	4,17
B60	+	-	-	36 %	3,25	5,00	2,13
B90	+	+	+	40 %	6,30	7,50	2,70
Effect(Score)	0,04	0,14	-0,10	36 %			
Effect(Perceived Stress)	-0,43	1,98	1,07		4,99		
Effect(Perceived Difficulty)	1,06	1,06	1,44			5,72	
Effect(Perceived Time)	-1,11	0,93	-0,36				2,97

Table 6.5: Effects

Score

From just observing the box-plot (Fig.6.7) of the score one will notice the scores from Task A with 60 minutes (A60) appear to vary significantly from the other three combinations. After using two sample t-tests on the different combinations (more details in Fig.D) and observing the resulting p-values (see Fig.6.3) we find significant difference between A60 and all other combinations as resulting p-values are <0.05 and we can thus assume this with about 95% certainty.

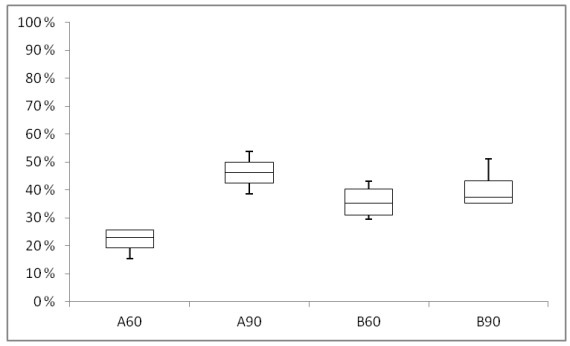


Figure 6.7: Boxplot: Score

However, there does not seem to be a significant difference between any of the other combinations unless we accept a lower degree of certainty. Even then we would have to go down to about 85% certainty to find significant difference between B60 and A90, while still not between B60 and B90.

This would indicate that the specific combination of low complexity (Task A) and low time (60 minutes) results in significantly poorer results, even compared to B60 where the groups had to do more work for the same score within the same time frame. This result was unexpected and is difficult to explain, but two possible explanations comes to mind.

Firstly, it could be a fluke. The sample size is extremely small as only four groups, i.e. 8 students, worked with A60. It is entirely possible the four groups which worked with B60 happened, by chance, to be significantly more experienced than the ones which worked with A60 even though the groups and tasks were assigned randomly.

Alternatively, there could be unknown psychological factors at play. This would fall outside the author's field of study and would have to be investigated by an expert in psychology, but it is the idea that, for instance, an individual with a known, set time-frame aim to be done at the end of the time frame, thus working faster and achieving more if he aims to be done with more within the same time-frame.

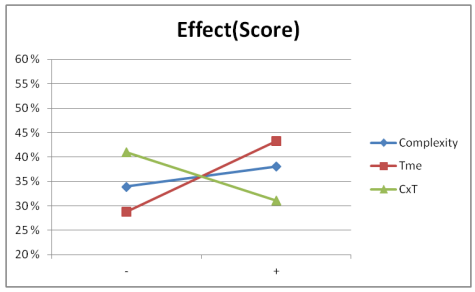


Figure 6.8: Effect(Score)

When calculating the effects of complexity and time (see Table 6.2.2) on score we find the main effect of complexity and time to be 4% and 14% respectively, while the interaction effect is -10% as shown in Fig.6.8. The main effect of complexity appears to be dismissible while the main effect of time appears to be significant, however the interaction effect appears to be both significant and negative. This can be explained by the same reasons as discussed above, as this stems from the gap in difference between A60 and A90 compared to B60 and B90.

In the end, however, these effects indicate that both time and complexity significantly effects the score. "Time" directly and "Complexity" through interaction with "Time". Thus, the results from the scores supports the H_3 hypothesis (both independent variables effect the results) while also dismissing the null hypothesis H_0 (none of the independent variables effect the result).

Perceived Stress, Time, and Difficulty

The three dependent variables collected through the survey all share one inheritable weakness: They, in essence, are a collection individuals' experience of concepts rated on an arbitrary scale from one to ten where each individual have their own understanding of both the concepts, the scale and how the concepts affect them. Remember, while stress can be measured using medical equipment and expertise, what is measured here is individuals' "experience of stress".

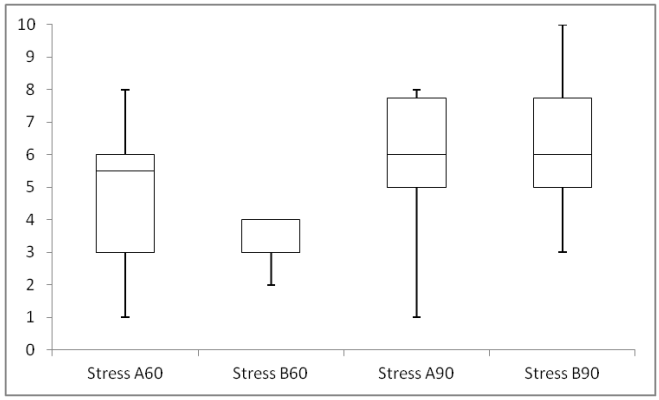


Figure 6.9: Boxplot: Perceived Stress

While SPAR-H generally operates with stress as a negative factor, the survey indicates very few of the students see stress as solely negative factor (see Fig.6.10). Further, if we observe the box-plot of perceived stress we find the range of each task set to span over half the scale with B60 as an exception. The sample size is large enough to detect some degree of clustering, but this is around 5 which for many is the go to value on arbitrary scales.

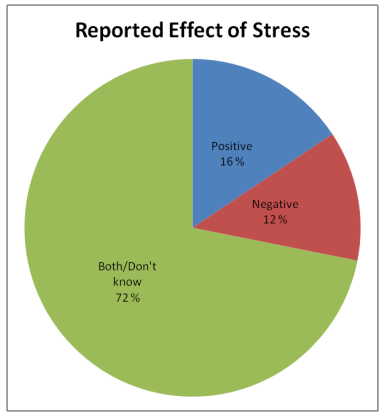


Figure 6.10: Reported effect of stress

These weaknesses makes it harder to explain oddities and trust expected results with a sample size of only six to ten per task set, but the results can arguably be used to show small degrees of effect from time or complexity, but will then

require larger effects to not be dismissed. The effects of time, complexity and their interaction is listed in Table 6.2.2. Most of the effects are <1 or barely more than 1 which isn't much on a one to ten arbitrary scale. Because of the weaknesses described above all these effects are dismissed as too small, however "Time's" main effect (1.98) on stress is considered large enough to be considered significant. Thus, the dependent variable "Perceived Stress" appears to indicate some support to the experiments H_1 hypothesis, while also indicating dismissal of H_o. While the dependent variables "Perceived Available Time" and "Perceived Difficulty" fail to prove significant effect of either "Time" or "Complexity", thus supporting the null hypothesis H_o.

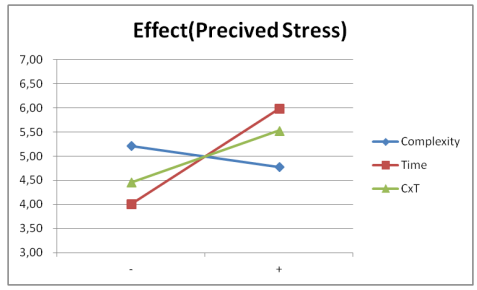


Figure 6.11: Effect(Perceived Stress)

6.3 Test Validity

6.3.1 Conclusion Validity

38 participants divided into groups of two, and then again divided by four by the task sets. These groups are then further reduced by data reduction. This experiment has a problem with low statistical power due to the small sample size. As mentioned both in the analysis of score and the values collected from the survey.

Reliability of measure is also also a problem when it comes to the measurements collected from the survey. As discussed earlier these measurements are in essence a collection of individuals' experience of concepts rated on an arbitrary scale from one to ten where each individual have their own understanding of both the concepts, the scale and how the concepts affect them. This is also a problem in any poll, but it gets worse due to the low statistical power.

6.3.2 Construct Validity

Experimenter expectancies can have affected the experiment to some degree. The experiment wishes to investigate if familiar concepts such as time and complexity effects the result of solving a task. However, finding someone with no expectations to how time and complexity effects task-solving may be proven difficult.

6.3.3 External Validity

The experiment used second year computer science students, because they were available and were expected to be familiar with software development. While they can be expected to have a shared lower boundary of experience and knowledge due to their shared three semester, there is no upper boundary. Individuals could have years of experience, training and programming skills even before they entered university. This lack of upper boundary leaves the uncertainty of results are based on skill or on cheating, which was the reason two groups were excluded in Section 6.2.1.

6.4 Experiment Conclusion

After the analysis we find that analysis the dependent variable "Score" indicates support to the H_3 hypothesis and "Perceived Stress" indicates support to H_1, while they both dismiss the null hypothesis H_o which "Perceived Available Time" and "Perceived Difficulty" supports. Due to the aforementioned weaknesses of the dependent variables from the survey, and the more empirical nature of "Score"; "Score" will be considered to have more weight. Additionally, H_3 requires there to be a significant effect on any of the results for either "Time" or "Complexity". So to this respect "Perceived Stress" only strengthens the notion that "Time" significantly effect the result, while not undermining "Complexity's" effect in other results. Thus, this analysis conclude that the experiment supports hypothesis H_3: "Both Available Time and Complexity significantly effect the results".

There are however considerations to this conclusion as discussed in above in Section 6.3. The conclusion validity is lowered due to low statistical power, which in turn lowers the validity of the experiment and it's conclusion. However, even when accounting for this we still found significant differences and effects for some of the results. Thus, the conclusion stands, but with a remark to a lower validity than preferred.

Chapter 7

Conclusion and Further Work

7.1 Conclusion

The goal of this thesis was to study how to apply Human Reliability Assessment(HRA) to the field of software development, by studying how the HRA method SPAR-H can be applied to software development.

In order to reach that goal this document has answered two research questions:

R1

Are there any of SPAR-H's Performance Shaping Factors (PSF) that is more useful to the field of software development than others, based on an evaluation of SPAR-H's PSFs?

This thesis have answered this by performing an evaluation on SPAR-H's PSFs, where the PSFs "Available Time", "Complexity", and "Stress" was recommended as most suited for within the field of software development.

R2

Does SPAR-H's PSFs actually affect performance in a software development scenario, based on an experiment using SPAR-H's PSFs as independent variables?

This thesis have answered this by performing an experiment using "Complexity" and "Available Time" as independent variables. The experiment concludes

that both “Complexity” and “Available Time” affects the result, and thus the performance, in software development. However, with remarks to lower degrees of conclusion validity due to a small sample size.

7.2 Further Work

The lower validity of the experiment generates a degree of uncertainty to the conclusion. For further work a retake on the experiment with a larger sample size and more experienced participants would help remove some of this uncertainty.

Beyond that point, applying SPAR-H or a modification of SPAR-H directly on a software development projects to see if HRA can be applied to the field software development.

Appendix A

Survey

Survey

ID _____

Group _____

Task _____

Rate how difficult you found the task (Higher is more difficult):

1 2 3 4 5 6 7 8 9 10

Rate whether time available was sufficient for performing the task (Higher is more sufficient):

1 2 3 4 5 6 7 8 9 10

Additional comment? Write below.

Rate to which degree you feel stress effected your performance in this task (Higher is to a higher degree):

1 2 3 4 5 6 7 8 9 10

Additional comment? Write below.

Turn page

Would you say stress effected your performance more positively or negatively?

◯ Positive ◯ Negative ◯ Both/Don't know

Additional comment? Write below.

Did anything unexpected or unforeseen effect your performance in any way? Write below.

Anything else? Write below.

Appendix B

Experiment Task A

Task A

Tidsfrist: 1,5 time (90 minutter)

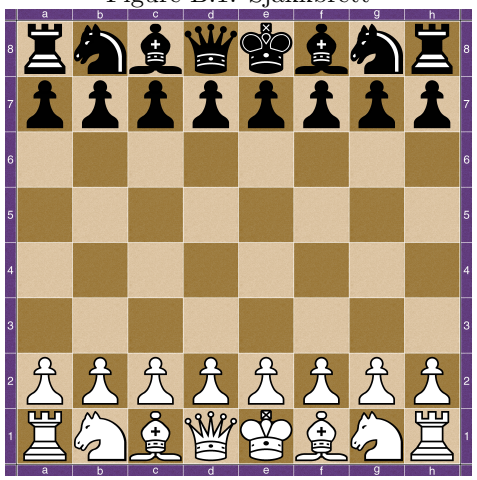
Du skal lage et sjakkspill, med klasser for å representere brikkefarge, brikker og brettet med alle brikkene. Reglene er noe forenklet, bl.a. er omgjøring av bønder og rokkade utelatt.

Sjakk spilles på et 8x8 brett, hvor rutene angis med bokstav a-h for kolonnen og tall 1-8 for raden. Nederste venstre hjørne har koordinatene "a1" og motsatt hjørne har koordinatene "h8". En rute på brettet kan være tom eller inneholde en hvit eller sort brikke.

Det er 6 forskjellige typer brikker, med hver sine regler for hvordan de flytter. Generelt kan en brikke enten flytte til et tomt felt eller til et felt med en brikke med motsatt farge, heretter kalt en motstanderbrikke. Det siste kalles å slå og brikken som flyttes vil da erstatte motstanderbrikken.

- Bonden flytter normalt 1 rute frem, men kan også flytte 2 (uten å hoppe over andre brikker) dersom den (fortsatt) står i utgangsposisjonen. Bonden slår motstandere 1 rute diagonalt forover, dvs. oppover for hvit og nedover for svart.
- Springeren (hest) flytter 2 ruter i én retning og 1 rute i retning vinkelrett på første, f.eks. 2 ruter frem og 1 til venstre eller 1 rute bakover og 2 til høyre. Springeren kan hoppe over andre brikker.
- Løperen (biskop) flytter 1 eller flere ruter diagonalt i en av 4 retninger og kan ikke hoppe over andre brikker.
- Tårnet flytter 1 eller flere ruter rett frem eller til siden i en av 4 retninger og kan ikke hoppe over andre brikker.
- Dronningen flytter 1 eller flere ruter rett frem, til siden eller diagonalt i en av 8 retninger og kan ikke hoppe over andre brikker.
- Kongen flytter 1 rute rett frem, til siden eller diagonalt i en av 8 retninger. Dersom kongen står slik at den kan slås av motstanderen i neste trekk, så er den sjakk.

Figure B.1: Sjakkbrett



Ved start har hver spiller 8 bønder, 2 springere, 2 løpere, 2 tårn, 1 dronning og 1 konge. Den ene spilleren har hvite brikker og den andre svarte. Utgangsposisjonen er som vist i figuren, med 1 rad bønder og 1 rad med hhv. tårn, springer, løper, dronning, konge, løper, springer og tårn, for hver spiller. De hvite bøndene flytter oppover og de sorte bøndene nedover.

a) Definer en enum-klasse kalt PieceColor, med verdiene WHITE og BLACK. Implementer metoden getOtherColor, som ikke tar noen parametre og som returnerer den andre fargeverdien.

Grensesnittet Piece er definert som følger:

```
public interface Piece {
public PieceColor getPieceColor();
public boolean canTake(String from, String to, Board board);
public boolean canMove(String from, String to, Board board);
}
```

Tanken er at dette grensesnittet skal implementeres av 6 brikkeklasser Pawn (bonde), Knight (springer), Bishop (løper), Rook (tårn), Queen (dronning) og King (konge).

Klassen Board skal representere sjakkbrettet vha. en tabell (array) med Piece-objekter, med dimensjoner 8x8. Metodene som refererer til ruter, skal bruke String-posisjoner (koordinater) på formatet "a1" til "h8", som vist i figuren over.

b) Implementer Board-klassen med den interne tabellen og følgende to innkapslingsmetoder.

```
Piece getPiece(String position) { ... }
void setPiece(String position, Piece piece) { ... }
```

- getPiece returnerer brikken på ruta angitt med position, eller null dersom ruta er tom. F.eks. skal getPiece("a1") og getPiece("h8") returnere en evt. brikke i hhv. ruta nederst til venstre og øverst til høyre.
- setPiece plasserer brikken piece i ruta angitt med position.

c) Implementer følgende hjelpemetoder i Board-klassen:

boolean isStraight(String from, String to) { ... }
boolean isDiagonal(String from, String to) { ... }
boolean isOccupiedBetween(String from, String to) { ... }

- isStraight returnerer en boolean som angir om et flytt fra from til to går langs en rad eller kolonne, altså i en av de 4 retningene parallelt med kantene på brettet.
- isDiagonal returnerer en boolean som angir om et flytt fra from til to går diagonalt, altså i en av de 4 retningene på skrå over brettet.
- isOccupiedBetween skal returnere en boolean som angir om én eller flere av rutene mellom (og ikke inkludert) from og to er fylt med en brikke.

d) Implementer de 6 brikkeklasser Pawn (bonde), Knight (springer), Bishop (løper), Rook (tårn), Queen (dronning), og King (konge); som alle implementerer grensesnittet Piece. Implementer også en konstruktør, som brukes for å sette brikkefargen. Logikken til metodene i Piecegrensesnittet skal være som følger:

- getPieceColor returnerer fargen til brikken.
- canMove returnerer en boolean som angir om denne brikken, dersom den står på from-posisjonen, har lov til å flytte til to-posisjonen. Du trenger ikke sjekke om brikken faktisk står på from eller om to er tom.
- canTake returnerer en boolean som angir om denne brikken, dersom den står på from-posisjonen, har lov til å ta en motstanderbrikke på to-posisjonen. Du trenger ikke sjekke om brikken faktisk står på from eller om det faktisk står en motstanderbrikke på to.

Appendix C

Experiment Task B

Task B

Tidsfrist: 1,5 time (90 minutter)

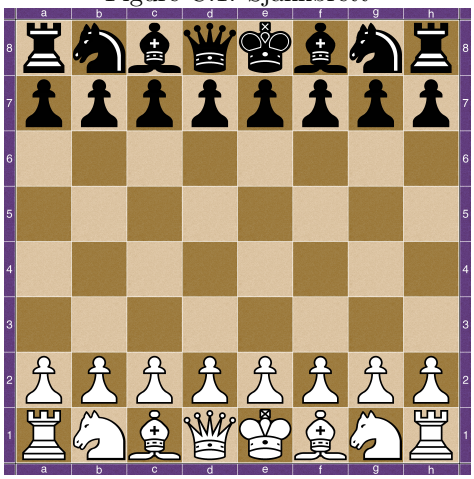
Du skal lage et sjakkspill, med klasser for å representere brikkefarge, brikker og brettet med alle brikkene. Reglene er noe forenklet, bl.a. er omgjøring av bønder og rokkade utelatt.

Sjakk spilles på et 8x8 brett, hvor rutene angis med bokstav a-h for kolonnen og tall 1-8 for raden. Nederste venstre hjørne har koordinatene "a1" og motsatt hjørne har koordinatene "h8". En rute på brettet kan være tom eller inneholde en hvit eller sort brikke.

Det er 6 forskjellige typer brikker, med hver sine regler for hvordan de flytter. Generelt kan en brikke enten flytte til et tomt felt eller til et felt med en brikke med motsatt farge, heretter kalt en motstanderbrikke. Det siste kalles å slå og brikken som flyttes vil da erstatte motstanderbrikken.

- Bonden flytter normalt 1 rute frem, men kan også flytte 2 (uten å hoppe over andre brikker) dersom den (fortsatt) står i utgangsposisjonen. Bonden slår motstandere 1 rute diagonalt forover, dvs. oppover for hvit og nedover for svart.
- Springeren (hest) flytter 2 ruter i én retning og 1 rute i retning vinkelrett på første, f.eks. 2 ruter frem og 1 til venstre eller 1 rute bakover og 2 til høyre. Springeren kan hoppe over andre brikker.
- Løperen (biskop) flytter 1 eller flere ruter diagonalt i en av 4 retninger og kan ikke hoppe over andre brikker.
- Tårnet flytter 1 eller flere ruter rett frem eller til siden i en av 4 retninger og kan ikke hoppe over andre brikker.
- Dronningen flytter 1 eller flere ruter rett frem, til siden eller diagonalt i en av 8 retninger og kan ikke hoppe over andre brikker.
- Kongen flytter 1 rute rett frem, til siden eller diagonalt i en av 8 retninger. Dersom kongen står slik at den kan slås av motstanderen i neste trekk, så er den sjakk.

Figure C.1: Sjakkbrett



Ved start har hver spiller 8 bønder, 2 springere, 2 løpere, 2 tårn, 1 dronning og 1 konge. Den ene spilleren har hvite brikker og den andre svarte. Utgangsposisjonen er som vist i figuren, med 1 rad bønder og 1 rad med hhv. tårn, springer, løper, dronning, konge, løper, springer og tårn, for hver spiller. De hvite bøndene flytter oppover og de sorte bøndene nedover.

a) Definer en enum-klasse kalt PieceColor, med verdiene WHITE og BLACK. Implementer metoden getOtherColor, som ikke tar noen parametre og som returnerer den andre fargeverdien.

Grensesnittet Piece er definert som følger:

```
public interface Piece {
public PieceColor getPieceColor();
public boolean canTake(String from, String to, Board board);
public boolean canMove(String from, String to, Board board);
}
```

Tanken er at dette grensesnittet skal implementeres av 6 brikkeklasser Pawn (bonde), Knight (springer), Bishop (løper), Rook (tårn), Queen (dronning) og King (konge).

Klassen Board skal representere sjakkbrettet vha. en tabell (array) med Piece-objekter, med dimensjoner 8x8. Metodene som refererer til ruter, skal bruke String-posisjoner (koordinater) på formatet "a1" til "h8", som vist i figuren over.

b) Implementer Board-klassen med den interne tabellen og følgende to innkapslingsmetoder.

```
Piece getPiece(String position) { ... }
void setPiece(String position, Piece piece) { ... }
```

- getPiece returnerer brikken på ruta angitt med position, eller null dersom ruta er tom. F.eks. skal getPiece("a1") og getPiece("h8") returnere en evt. brikke i hhv. ruta nederst til venstre og øverst til høyre.
- setPiece plasserer brikken piece i ruta angitt med position.

c) Implementer følgende hjelpemetoder i Board-klassen:

```
boolean isStraight(String from, String to) { ... }
boolean isDiagonal(String from, String to) { ... }
boolean isOccupiedBetween(String from, String to) { ... }
```

- isStraight returnerer en boolean som angir om et flytt fra from til to går langs en rad eller kolonne, altså i en av de 4 retningene parallelt med kantene på brettet.
- isDiagonal returnerer en boolean som angir om et flytt fra from til to går diagonalt, altså i en av de 4 retningene på skrå over brettet.
- isOccupiedBetween skal returnere en boolean som angir om én eller flere av rutene mellom (og ikke inkludert) from og to er fylt med en brikke.

d) Implementer de 6 brikkeklasser Pawn (bonde), Knight (springer), Bishop (løper), Rook (tårn), Queen (dronning), og King (konge); som alle implementerer grensesnittet Piece. Implementer også en konstruktør, som brukes for å sette brikkefargen. Logikken til metodene i Piecegrensesnittet skal være som følger:

- getPieceColor returnerer fargen til brikken.
- canMove returnerer en boolean som angir om denne brikken, dersom den står på from-posisjonen, har lov til å flytte til to-posisjonen. Du trenger ikke sjekke om brikken faktisk står på from eller om to er tom.
- canTake returnerer en boolean som angir om denne brikken, dersom den står på from-posisjonen, har lov til å ta en motstanderbrikke på to-posisjonen. Du trenger ikke sjekke om brikken faktisk står på from eller om det faktisk står en motstanderbrikke på to.

e) Implementer en konstruktør for Board som setter opp brikkene riktig og at klassen i tillegg har følgende metoder:

```
boolean isLegalMove(PieceColor color, String from, String to) { ... }
void movePiece(String from, String to) { ... }
boolean isCheck(PieceColor color) { ... }
```

- isLegalMove returnerer en boolean som angir om det er lov for spilleren med fargen color å flytte brikken på from- posisjonen til to-posisjonen.
- movePiece flytter en brikke fra from-posisjonen til to-posisjonen.
- isCheck returnerer en boolean som angir om kongebrikken med fargen color er sjakk.

f) Implementer en main-metode, som oppretter et sjakkbrett og lar brukeren skrive inn flytt (velg selv formatet). Husk på at hvit trekker først og at de deretter veksler mellom å trekke. Programmet skal si fra når en konge blir sjakk og avslutte når en konge blir tatt. Ellers trenger du ikke bry deg om regler ut over de som er beskrevet over.

Under ser du mulig utskrift og input ved kjøring av en slik main-metode.

```
    WHITE's turn:
d2-d4
BLACK's turn:
e7-e5
WHITE's turn:
d2-d4
Illegal move!
WHITE's turn:
d4-e5
BLACK's turn:
```

Appendix D

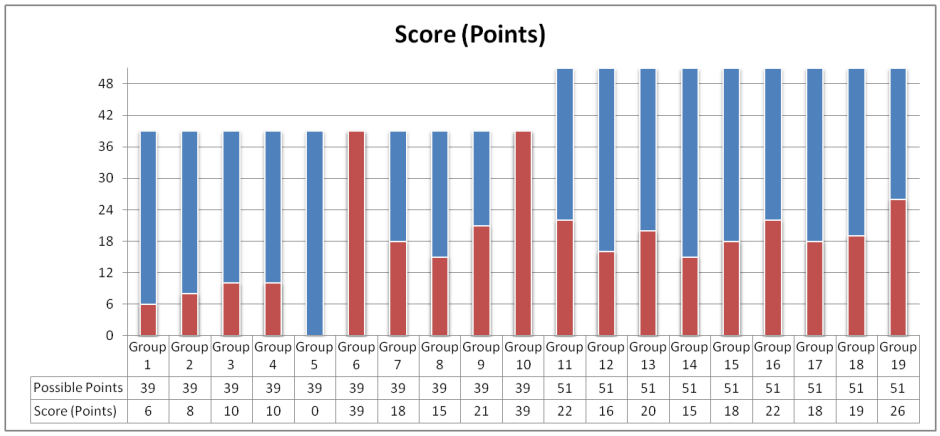
Experiment Results

Table D.1: Raw Results

Task	Group	Score	Student	Difficulty (1-10)	Time (1-10)	Stress (1-10)	Stress Effect Comment
A60	1	15 %	1	8	2	6	Negative
			2	7	1	3	Negative
	2	21 %	3	4	3	3	Both/Don't know
			4	4	3	1	Both/Don't know
	3	26 %	5	3	4	5	Both/Don't know
			6	4	5	6	Both/Don't know
	4	26 %	7	7	2	6	Both/Don't know
			8	6	3	8	Both/Don't know
	5	0 %	9	5	4	3	Positive
			10	5	6	4	Both/Don't know
A90	6	100 %	11	3	3	2	Negative
			12	8	3	2	Both/Don't know
	7	46 %	13	3	7	8	Positive
			14	5	5	5	Both/Don't know
	8	38 %	15	8	3	7	Both/Don't know
			16	2	3	1	Positive
	9	54 %	17	6	3	8	Positive
			18	6	4	5	Positive
	10	100 %	19	7	9	5	Positive
			20	6	10	3	Positive
B60	11	43 %	21	5	1	4	Both/Don't know
			22	6	3	4	Both/Don't know
	12	31 %	23	4	3	4	Both/Don't know
			24	4	2	2	Both/Don't know
	13	39 %	25	7	2	3	Both/Don't know
			26	7	2	3	Both/Don't know
	14	29 %	27	3	2	3	Both/Don't know
			28	4	2	3	Both/Don't know
B90	15	35 %	29	7	3	7	Both/Don't know
			30	7	3	6	Both/Don't know
	16	43 %	31	7	3	5	Both/Don't know
			32	8	2	5	Both/Don't know
	17	35 %	33	8	3	9	Negative
			34	7	2	10	Negative
	18	37 %	35	8	3	4	Both/Don't know
			36	9	1	3	Both/Don't know
	19	51 %	37	7	4	6	Positive
			38	7	3	8	Both/Don't know

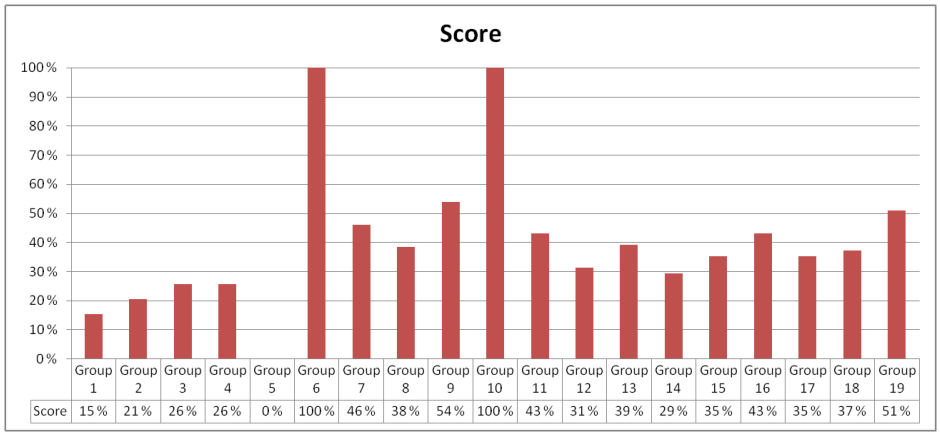
Table D.2: Score (Points) Overview

Group	Total Score	a)	b1) getPiece	b2) setPiece	c1) isStraight	c2) isDiagonal	c3) isOccupiedBetween	-			
Group 1	6	3	3	0	0	0	0	-			
Group 2	8	1	1	1	3	1	1	-			
Group 3	10	1	1	1	3	3	1	-			
Group 4	10	3	1	1	3	1	1	-			
Group 5	0	0	0	0	0	0	0	-			
Group 6	18	3	3	3	3	3	3	-			
Group 7	8	1	1	1	3	1	1	-			
Group 8	10	3	1	1	3	1	1	-			
Group 9	10	3	1	1	3	1	1	-			
Group 10	18	3	3	3	3	3	3	-			
Group 11	10	3	1	1	3	1	1	-			
Group 12	16	3	3	3	3	3	1	-			
Group 13	16	1	3	3	3	3	3	-			
Group 14	12	3	1	1	3	3	1	-			
Group 15	16	3	3	3	3	3	1	-			
Group 16	12	3	1	1	3	3	1	-			
Group 17	11	3	1	1	3	3	0	-			
Group 18	14	1	3	3	3	3	1	-			
Group 19	16	3	3	3	3	3	1	-			
d) Pawn canMove	**d) Pawn canTake**	**d) King canMove**	**d) Queen canMove**	**d) Rook canMove**	**d) Bishop canMove**	**d) Knight canMove**	**e1) isLegalMove**	**e2) movePiece**	**e3) isCheck**	**f)**	
0	0	0	0	0	0	0					
0	0	0	0	0	0	0					
0	0	0	0	0	0	0					
0	0	0	0	0	0	0					
0	0	0	0	0	0	0					
3	3	3	3	3	3	3					
0	1	1	3	3	1	1					
1	1	0	0	0	0	3					
1	1	0	3	3	3	0					
3	3	3	3	3	3	3					
1	1	1	3	3	3	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	
0	0	0	1	3	0	0	0	0	0	0	
0	0	3	0	0	0	0	0	0	0	0	
1	1	0	0	0	0	0	0	0	0	0	
1	3	3	0	3	0	0	0	0	0	0	
0	0	0	3	1	3	0	0	0	0	0	
1	0	0	0	3	1	0	0	0	0	0	
3	1	3	0	3	0	0	0	0	0	0	



	Group 1	Group 2	Group 3	Group 4	Group 5	Group 6	Group 7	Group 8	Group 9	Group 10	Group 11	Group 12	Group 13	Group 14	Group 15	Group 16	Group 17	Group 18	Group 19
Possible Points	39	39	39	39	39	39	39	39	39	39	51	51	51	51	51	51	51	51	51
Score (Points)	6	8	10	10	0	39	18	15	21	39	22	16	20	15	18	22	18	19	26

Figure D.1: Score (Points)



	Group 1	Group 2	Group 3	Group 4	Group 5	Group 6	Group 7	Group 8	Group 9	Group 10	Group 11	Group 12	Group 13	Group 14	Group 15	Group 16	Group 17	Group 18	Group 19
Score	15%	21%	26%	26%	0%	100%	46%	38%	54%	100%	43%	31%	39%	29%	35%	43%	35%	37%	51%

Figure D.2: Score (Percents)

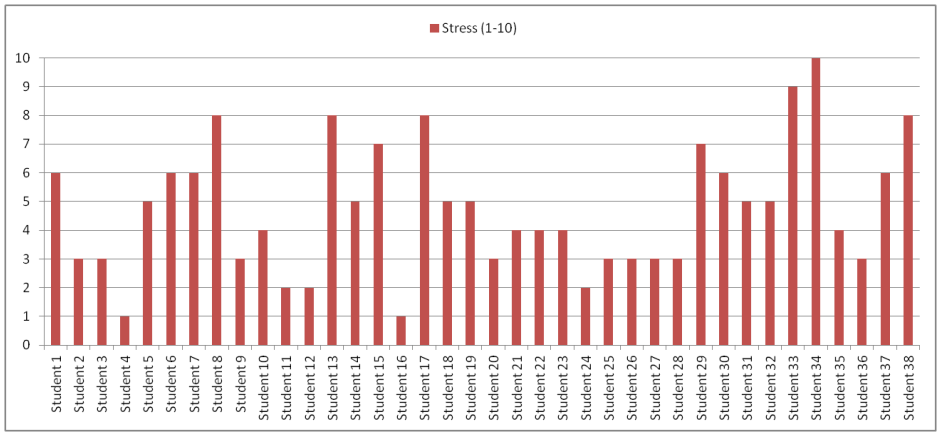


Figure D.3: Stress

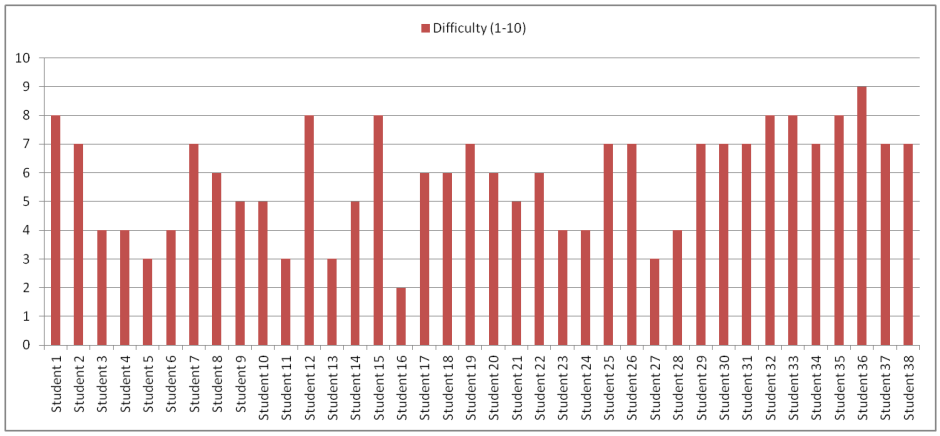


Figure D.4: Difficulty

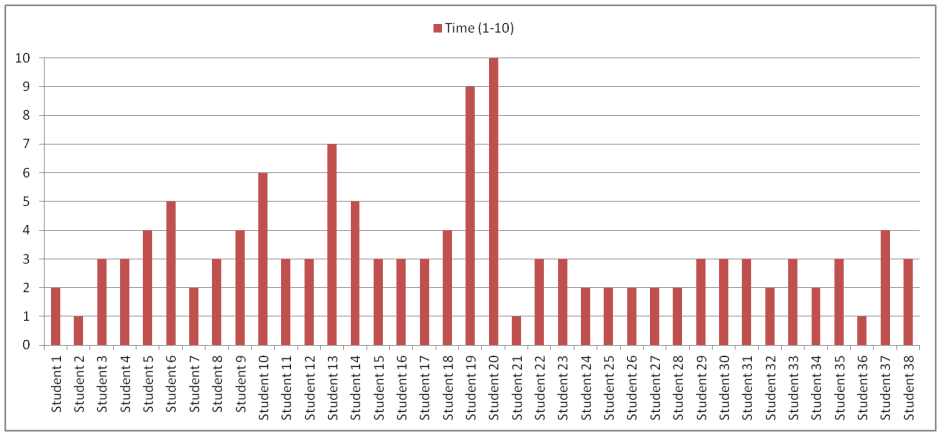


Figure D.5: Time

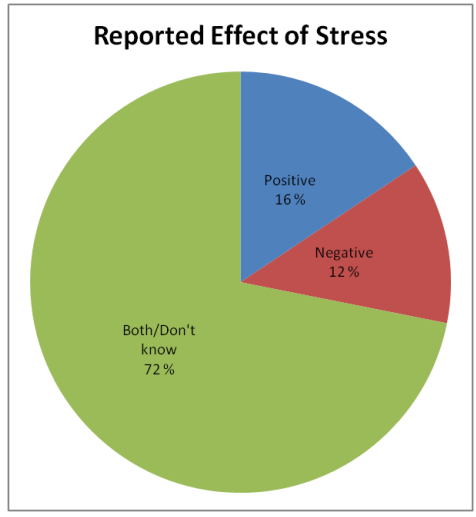


Figure D.6: Reported Effect of Stress

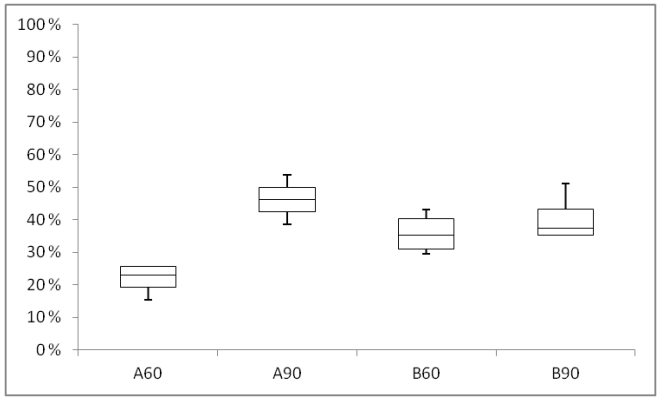


Figure D.7: Boxplot: Score

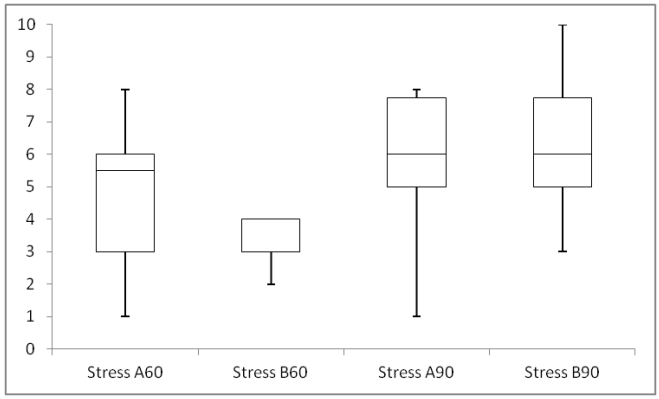


Figure D.8: Boxplot: Perceived Stress

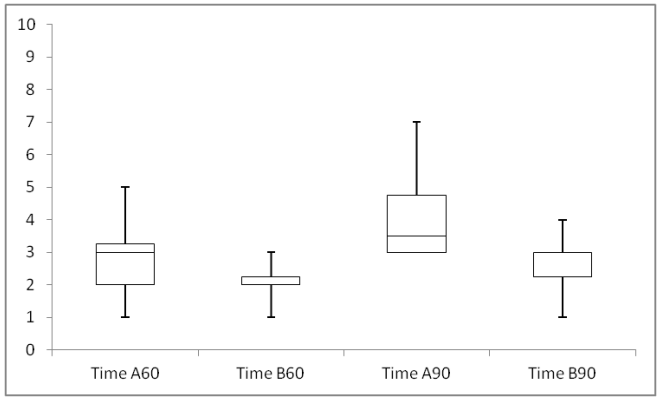


Figure D.9: Boxplot: Perceived Time

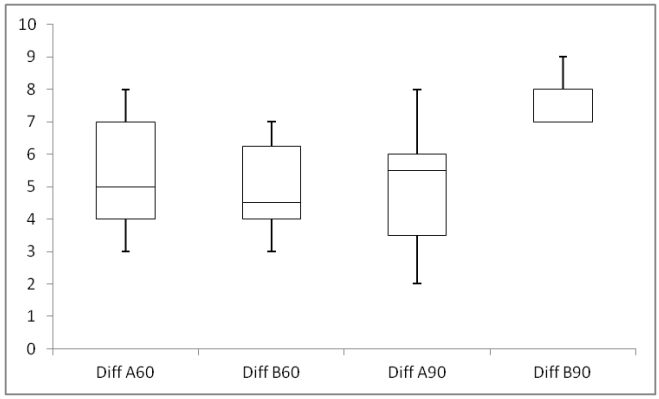


Figure D.10: Boxplot: Perceived Difficulty

	Complexity	Time	CxT	Score	Stress	Rep. Difficulty	Rep. Time
A60	-	-	+	22 %	4,75	5,38	2,88
A90	-	+	-	46 %	5,67	5,00	4,17
B60	+	-	-	36 %	3,25	5,00	2,13
B90	+	+	+	40 %	6,30	7,50	2,70
Effect(Score)	0,04	0,14	-0,10	36 %			
Effect(Stress)	-0,43	1,98	1,07		4,99		
Effect(Rep. Difficulty)	1,06	1,06	1,44			5,72	
Effect(Rep. Time)	-1,11	0,93	-0,36				2,97

Table D.3: Effects

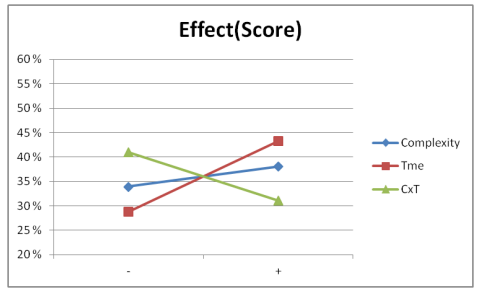


Figure D.11: Effect(Score)

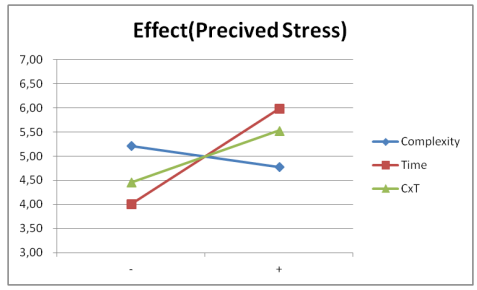


Figure D.12: Effect(Perceived Stress)

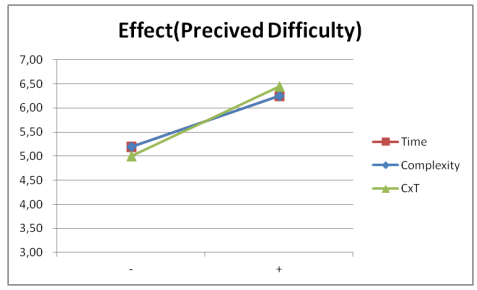


Figure D.13: Effect(Perceived Difficulty)

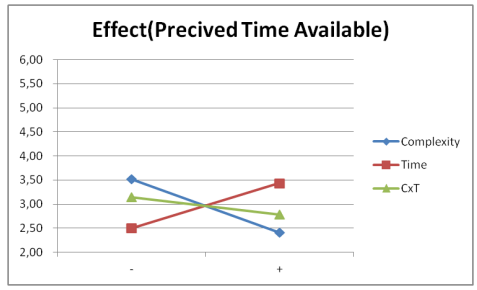


Figure D.14: Effect(Perceived Time)

t-Test: To utvalg med antatt ulike varianser		
	Stress (A60)	*Stress (A90)*
Gjennomsnitt	4,75	5,666666667
Varians	5,071428571	7,066666667
Observasjoner	8	6
Antatt avvik mellom gjennomsnittene	0	
fg	10	
t-Stat	-0,681032027	
P(T$_i$=t) ensidig	0,255656313	
T-kritisk, ensidig	1,812461102	
P(T$_i$=t) tosidig	0,511312625	
T-kritisk, tosidig	2,228138842	

t-Test: To utvalg med antatt ulike varianser		
	Stress (A60)	*Stress (A90)*
Gjennomsnitt	4,75	5,666666667
Varians	5,071428571	7,066666667
Observasjoner	8	6
Antatt avvik mellom gjennomsnittene	0	
fg	8	
t-Stat	1,797434069	
P(T$_i$=t) ensidig	0,054991721	
T-kritisk, ensidig	1,859548033	
P(T$_i$=t) tosidig	0,109983443	
T-kritisk, tosidig	2,306004133	

t-Test: To utvalg med antatt ulike varianser		
	Stress (A60)	*Stress (B90)*
Gjennomsnitt	4,75	6,3
Varians	5,071428571	4,9
Observasjoner	8	10
Antatt avvik mellom gjennomsnittene	0	
fg	15	
t-Stat	-1,462050395	
P(T$_i$=t) ensidig	0,082180736	
T-kritisk, ensidig	1,753050325	
P(T$_i$=t) tosidig	0,164361471	
T-kritisk, tosidig	2,131449536	

t-Test: To utvalg med antatt ulike varianser		
	Stress (A90)	*Stress (B60)*
Gjennomsnitt	5,666666667	3,25
Varians	7,066666667	0,5
Observasjoner	6	8
Antatt avvik mellom gjennomsnittene	0	
fg	6	
t-Stat	2,169987692	
P(T$_i$=t) ensidig	0,036531594	
T-kritisk, ensidig	1,943180274	
P(T$_i$=t) tosidig	0,073063188	
T-kritisk, tosidig	2,446911846	

t-Test: To utvalg med antatt ulike varianser		
	Stress (A90)	*Stress (B90)*
Gjennomsnitt	5,666666667	6,3
Varians	7,066666667	4,9
Observasjoner	6	10
Antatt avvik mellom gjennomsnittene	0	
fg	9	
t-Stat	-0,490414446	
P(T$_i$=t) ensidig	0,317787351	
T-kritisk, ensidig	1,833112923	
P(T$_i$=t) tosidig	0,635574703	
T-kritisk, tosidig	2,262157158	

t-Test: To utvalg med antatt ulike varianser		
	Stress (B60)	*Stress (B90)*
Gjennomsnitt	3,25	6,3
Varians	0,5	4,9
Observasjoner	8	10
Antatt avvik mellom gjennomsnittene	0	
fg	11	
t-Stat	-4,103304043	
P(T$_i$=t) ensidig	0,000874776	
T-kritisk, ensidig	1,795884814	
P(T$_i$=t) tosidig	0,001749553	
T-kritisk, tosidig	2,200985159	

Table D.4: t-Tests: Stress

t-Test: To utvalg med antatt ulike varianser		
	Time (A60)	*Time (A90)*
Gjennomsnitt	2,875	4,166666667
Varians	1,553571429	2,566666667
Observasjoner	8	6
Antatt avvik mellom gjennomsnittene	0	
fg	9	
t-Stat	-1,637812815	
P(T_i=t) ensidig	0,067942848	
T-kritisk, ensidig	1,833112923	
P(T_i=t) tosidig	0,135885696	
T-kritisk, tosidig	2,262157158	

t-Test: To utvalg med antatt ulike varianser		
	Time (A90)	*Time (B60)*
Gjennomsnitt	4,166666667	2,125
Varians	2,566666667	0,410714286
Observasjoner	6	8
Antatt avvik mellom gjennomsnittene	0	
fg	6	
t-Stat	2,949606076	
P(T_i=t) ensidig	0,012813029	
T-kritisk, ensidig	1,943180274	
P(T_i=t) tosidig	0,025626057	
T-kritisk, tosidig	2,446911846	

t-Test: To utvalg med antatt ulike varianser		
	Time (A60)	*Time (B60)*
Gjennomsnitt	2,875	2,125
Varians	1,553571429	0,410714286
Observasjoner	8	8
Antatt avvik mellom gjennomsnittene	0	
fg	10	
t-Stat	1,513574937	
P(T_i=t) ensidig	0,080539359	
T-kritisk, ensidig	1,812461102	
P(T_i=t) tosidig	0,161078717	
T-kritisk, tosidig	2,228138842	

t-Test: To utvalg med antatt ulike varianser		
	Time (A60)	*Time (B90)*
Gjennomsnitt	2,875	2,7
Varians	1,553571429	0,677777778
Observasjoner	8	10
Antatt avvik mellom gjennomsnittene	0	
fg	12	
t-Stat	0,341907622	
P(T_i=t) ensidig	0,369166835	
T-kritisk, ensidig	1,782287548	
P(T_i=t) tosidig	0,738333669	
T-kritisk, tosidig	2,178812827	

t-Test: To utvalg med antatt ulike varianser		
	Time (A90)	*Time (B90)*
Gjennomsnitt	4,166666667	2,7
Varians	2,566666667	0,677777778
Observasjoner	6	10
Antatt avvik mellom gjennomsnittene	0	
fg	7	
t-Stat	2,083460385	
P(T_i=t) ensidig	0,037848705	
T-kritisk, ensidig	1,894578604	
P(T_i=t) tosidig	0,07569741	
T-kritisk, tosidig	2,364624251	

t-Test: To utvalg med antatt ulike varianser		
	Time (B60)	*Time (B90)*
Gjennomsnitt	2,125	2,7
Varians	0,410714286	0,677777778
Observasjoner	8	10
Antatt avvik mellom gjennomsnittene	0	
fg	16	
t-Stat	-1,666022473	
P(T_i=t) ensidig	0,057581444	
T-kritisk, ensidig	1,745883669	
P(T_i=t) tosidig	0,115162887	
T-kritisk, tosidig	2,119905285	

Table D.5: t-test: Time

t-Test: To utvalg med antatt ulike varianser		
	Difficulty (A60)	*Difficulty (A90)*
Gjennomsnitt	5,375	5
Varians	3,410714286	4,8
Observasjoner	8	6
Antatt avvik mellom gjennomsnittene	0	
fg	10	
t-Stat	0,338630403	
P(T≤t) ensidig	0,37094544	
T-kritisk, ensidig	1,812461102	
P(T≤t) tosidig	0,741890879	
T-kritisk, tosidig	2,228138842	

t-Test: To utvalg med antatt ulike varianser		
	Difficulty (A60)	*Difficulty (B60)*
Gjennomsnitt	5,375	5
Varians	3,410714286	2,285714286
Observasjoner	8	8
Antatt avvik mellom gjennomsnittene	0	
fg	13	
t-Stat	0,444400903	
P(T≤t) ensidig	0,332030485	
T-kritisk, ensidig	1,770933383	
P(T≤t) tosidig	0,66406097	
T-kritisk, tosidig	2,160368652	

t-Test: To utvalg med antatt ulike varianser		
	Difficulty (A90)	*Difficulty (B60)*
Gjennomsnitt	5	2,125
Varians	4,8	0,410714286
Observasjoner	6	8
Antatt avvik mellom gjennomsnittene	0	
fg	6	
t-Stat	3,115921527	
P(T≤t) ensidig	0,010346057	
T-kritisk, ensidig	1,943180274	
P(T≤t) tosidig	0,020692114	
T-kritisk, tosidig	2,446911846	

t-Test: To utvalg med antatt ulike varianser		
	Difficulty (A90)	*Difficulty (B90)*
Gjennomsnitt	5	2,7
Varians	4,8	0,677777778
Observasjoner	6	10
Antatt avvik mellom gjennomsnittene	0	
fg	6	
t-Stat	2,469014121	
P(T≤t) ensidig	0,024261267	
T-kritisk, ensidig	1,943180274	
P(T≤t) tosidig	0,048522534	
T-kritisk, tosidig	2,446911846	

t-Test: To utvalg med antatt ulike varianser		
	Difficulty (A60)	*Difficulty (B90)*
Gjennomsnitt	5,375	2,7
Varians	3,410714286	0,677777778
Observasjoner	8	10
Antatt avvik mellom gjennomsnittene	0	
fg	9	
t-Stat	3,805474889	
P(T≤t) ensidig	0,002091033	
T-kritisk, ensidig	1,833112923	
P(T≤t) tosidig	0,004182065	
T-kritisk, tosidig	2,262157158	

t-Test: To utvalg med antatt ulike varianser		
	Difficulty (B60)	*Difficulty (B90)*
Gjennomsnitt	2,125	2,7
Varians	0,410714286	0,677777778
Observasjoner	8	10
Antatt avvik mellom gjennomsnittene	0	
fg	16	
t-Stat	-1,666022473	
P(T≤t) ensidig	0,057581444	
T-kritisk, ensidig	1,745883669	
P(T≤t) tosidig	0,115162887	
T-kritisk, tosidig	2,119905285	

Table D.6: t-test: Difficulty

t-Test: To utvalg med antatt ulike varianser		
	Score (A60)	*Score (A90)*
Gjennomsnitt	0,217948718	0,461538462
Varians	0,002410695	0,00591716
Observasjoner	4	3
Antatt avvik mellom gjennomsnittene	0	
fg	3	
t-Stat	-4,80026595	
P(T_i=t) ensidig	0,008602758	
T-kritisk, ensidig	2,353363435	
P(T_i=t) tosidig	0,017205515	
T-kritisk, tosidig	3,182446305	

t-Test: To utvalg med antatt ulike varianser		
	Score (A60)	*Score (B60)*
Gjennomsnitt	0,217948718	0,357843137
Varians	0,002410695	0,004197104
Observasjoner	4	4
Antatt avvik mellom gjennomsnittene	0	
fg	6	
t-Stat	-3,441929681	
P(T_i=t) ensidig	0,006884179	
T-kritisk, ensidig	1,943180274	
P(T_i=t) tosidig	0,013768358	
T-kritisk, tosidig	2,446911846	

t-Test: To utvalg med antatt ulike varianser		
	Score (A60)	*Score (B90)*
Gjennomsnitt	0,217948718	0,403921569
Varians	0,002410695	0,004536717
Observasjoner	4	5
Antatt avvik mellom gjennomsnittene	0	
fg	7	
t-Stat	-4,785844996	
P(T_i=t) ensidig	0,000999358	
T-kritisk, ensidig	1,894578604	
P(T_i=t) tosidig	0,001998716	
T-kritisk, tosidig	2,364624251	

t-Test: To utvalg med antatt ulike varianser		
	Score (A90)	*Score (B60)*
Gjennomsnitt	0,461538462	0,357843137
Varians	0,00591716	0,004197104
Observasjoner	3	4
Antatt avvik mellom gjennomsnittene	0	
fg	4	
t-Stat	1,886410461	
P(T_i=t) ensidig	0,066148894	
T-kritisk, ensidig	2,131846782	
P(T_i=t) tosidig	0,132297789	
T-kritisk, tosidig	2,776445105	

t-Test: To utvalg med antatt ulike varianser		
	Score (A90)	*Score (B90)*
Gjennomsnitt	0,461538462	0,403921569
Varians	0,00591716	0,004536717
Observasjoner	3	5
Antatt avvik mellom gjennomsnittene	0	
fg	4	
t-Stat	1,073677755	
P(T_i=t) ensidig	0,171711123	
T-kritisk, ensidig	2,131846782	
P(T_i=t) tosidig	0,343422245	
T-kritisk, tosidig	2,776445105	

t-Test: To utvalg med antatt ulike varianser		
	Score (B60)	*Score (B90)*
Gjennomsnitt	0,357843137	0,403921569
Varians	0,004197104	0,004536717
Observasjoner	4	5
Antatt avvik mellom gjennomsnittene	0	
fg	7	
t-Stat	-1,041704464	
P(T_i=t) ensidig	0,166092199	
T-kritisk, ensidig	1,894578604	
P(T_i=t) tosidig	0,332184399	
T-kritisk, tosidig	2,364624251	

Table D.7: t-test: Score

P-values(t-Tests)		Score	Stress	Time	Difficulty
A60	A90	0,017	0,511	0,136	0,742
A60	B60	0,014	0,110	0,161	0,664
A60	B90	0,002	0,164	0,738	0,004
A90	B60	0,132	0,073	0,026	0,049
A90	B90	0,343	0,636	0,076	0,017
B60	B90	0,332	0,002	0,115	0,115

Table D.8: p-values (t-tests)

Appendix E

Asserter.java

```
package Test;

public class Asserter {


private static Board clearBoard(Board b){
b.setPiece("a1", null);
b.setPiece("b1", null);
b.setPiece("c1", null);
b.setPiece("d1", null);
b.setPiece("e1", null);
b.setPiece("f1", null);
b.setPiece("g1", null);
b.setPiece("h1", null);

b.setPiece("a2", null);
b.setPiece("b2", null);
b.setPiece("c2", null);
b.setPiece("d2", null);
b.setPiece("e2", null);
b.setPiece("f2", null);
b.setPiece("g2", null);
b.setPiece("h2", null);

b.setPiece("a3", null);
b.setPiece("b3", null);
b.setPiece("c3", null);
b.setPiece("d3", null);
b.setPiece("e3", null);
b.setPiece("f3", null);
b.setPiece("g3", null);
b.setPiece("h3", null);
```

```
b.setPiece("a4", null);
b.setPiece("b4", null);
b.setPiece("c4", null);
b.setPiece("d4", null);
b.setPiece("e4", null);
b.setPiece("f4", null);
b.setPiece("g4", null);
b.setPiece("h4", null);

b.setPiece("a5", null);
b.setPiece("b5", null);
b.setPiece("c5", null);
b.setPiece("d5", null);
b.setPiece("e5", null);
b.setPiece("f5", null);
b.setPiece("g5", null);
b.setPiece("h5", null);

b.setPiece("a6", null);
b.setPiece("b6", null);
b.setPiece("c6", null);
b.setPiece("d6", null);
b.setPiece("e6", null);
b.setPiece("f6", null);
b.setPiece("g6", null);
b.setPiece("h6", null);

b.setPiece("a7", null);
b.setPiece("b7", null);
b.setPiece("c7", null);
b.setPiece("d7", null);
b.setPiece("e7", null);
b.setPiece("f7", null);
b.setPiece("g7", null);
b.setPiece("h7", null);

b.setPiece("a8", null);
b.setPiece("b8", null);
b.setPiece("c8", null);
b.setPiece("d8", null);
b.setPiece("e8", null);
b.setPiece("f8", null);
b.setPiece("g8", null);
b.setPiece("h8", null);


return b;
}
```

```
private static void assertA(){

// assert(false);
assert(PieceColor.WHITE != null);
assert(PieceColor.BLACK != null);
assert(PieceColor.WHITE.getOtherColor() == PieceColor.BLACK);
assert(!(PieceColor.WHITE.getOtherColor() == PieceColor.WHITE));
assert((PieceColor.BLACK.getOtherColor() == PieceColor.WHITE));
assert(!(PieceColor.BLACK.getOtherColor() == PieceColor.BLACK));

}

private static void assertB(){
// Board b = clearBoard(new Board());
Board b = new Board();
Piece p = new Piece() {

@Override
public PieceColor getPieceColor() {
// TODO Auto-generated method stub
return null;
}

@Override
public boolean canTake(String from, String to, Board board) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean canMove(String from, String to, Board board) {
// TODO Auto-generated method stub
return false;
}
};

b.setPiece("a3", p);
assert(b.getPiece("a3").equals(p));

b.setPiece("f4", p);
assert(b.getPiece("f4").equals(p));

b.setPiece("c2", p);
assert(b.getPiece("c2").equals(p));

assert( (b.getPiece("g5") == null || !(b.getPiece("g5").equals(p))));
assert( (b.getPiece("h6") == null || !(b.getPiece("h6").equals(p))));
```

```
assert( (b.getPiece("a1") == null|| !(b.getPiece("a1").equals(p))));
}

private static void assertC(){

Board board = clearBoard(new Board());;

assert(board.isStraight("a1", "a2"));
assert(board.isStraight("a2", "a4"));
assert(board.isStraight("b3", "b6"));
assert(board.isStraight("c5", "c8"));
assert(board.isStraight("a1", "b1"));
assert(board.isStraight("c3", "f3"));
assert(board.isStraight("a7", "a2"));
assert(board.isStraight("a8", "a5"));
assert(board.isStraight("b7", "b1"));
assert(board.isStraight("c8", "c1"));
assert(board.isStraight("f1", "a1"));
assert(board.isStraight("h3", "c3"));
assert(!board.isStraight("h6", "c3"));
assert(!board.isStraight("b3", "a2"));
assert(!board.isStraight("e1", "g8"));

assert(board.isDiagonal("a1", "h8"));
assert(board.isDiagonal("a2", "d5"));
assert(board.isDiagonal("b5", "c4"));
assert(board.isDiagonal("c6", "f3"));
assert(board.isDiagonal("d4", "g7"));
assert(board.isDiagonal("a7", "b8"));
assert(board.isDiagonal("a6", "c8"));
assert(board.isDiagonal("a5", "d8"));
assert(board.isDiagonal("a4", "e8"));
assert(board.isDiagonal("a3", "f8"));
assert(board.isDiagonal("h8", "a1"));
assert(board.isDiagonal("h1", "a8"));
assert(!board.isDiagonal("a3", "d8"));
assert(!board.isDiagonal("a2", "e8"));
assert(!board.isDiagonal("a4", "f8"));
assert(!board.isDiagonal("h6", "a1"));
assert(!board.isDiagonal("h8", "a8"));

assert(board.isOccupiedBetween("a3", "h3") == false);
board.setPiece("d3", new Piece() {

@Override
public PieceColor getPieceColor() {
// TODO Auto-generated method stub
return null;
}
```

```
@Override
public boolean canTake(String from, String to, Board board) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean canMove(String from, String to, Board board) {
// TODO Auto-generated method stub
return false;
}
});
assert(board.isOccupiedBetween("a3", "h3") == true);

assert(board.isOccupiedBetween("b4", "d6") == false);
board.setPiece("c5", new Piece() {

@Override
public PieceColor getPieceColor() {
// TODO Auto-generated method stub
return null;
}

@Override
public boolean canTake(String from, String to, Board board) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean canMove(String from, String to, Board board) {
// TODO Auto-generated method stub
return false;
}
});
assert(board.isOccupiedBetween("b4", "d6") == true);

assert(board.isOccupiedBetween("f5", "c8") == false);
board.setPiece("e6", new Piece() {

@Override
public PieceColor getPieceColor() {
// TODO Auto-generated method stub
return null;
}

@Override
public boolean canTake(String from, String to, Board board) {
```

```
// TODO Auto-generated method stub
return false;
}

@Override
public boolean canMove(String from, String to, Board board) {
// TODO Auto-generated method stub
return false;
}
});
assert(board.isOccupiedBetween("f5", "c8") == true);
}

private static void assertD() {

Board board =clearBoard(new Board());
Pawn p = new Pawn(PieceColor.WHITE);
Pawn p2 = new Pawn(PieceColor.BLACK);
King k = new King(PieceColor.WHITE);
King k2 = new King(PieceColor.BLACK);
Queen q = new Queen(PieceColor.WHITE);
Queen q2 = new Queen(PieceColor.BLACK);
Bishop b = new Bishop(PieceColor.WHITE);
Bishop b2 = new Bishop(PieceColor.BLACK);
Knight kn = new Knight(PieceColor.WHITE);
Knight kn2 = new Knight(PieceColor.BLACK);
Rook r = new Rook(PieceColor.WHITE);
Rook r2 = new Rook(PieceColor.BLACK);

assert(p.getPieceColor() == PieceColor.WHITE);
assert(!(p2.getPieceColor() == PieceColor.WHITE));
assert(k.getPieceColor() == PieceColor.WHITE);
assert(!(k2.getPieceColor() == PieceColor.WHITE));
assert(q.getPieceColor() == PieceColor.WHITE);
assert(!(q2.getPieceColor() == PieceColor.WHITE));
assert(b.getPieceColor() == PieceColor.WHITE);
assert(!(b2.getPieceColor() == PieceColor.WHITE));
assert(kn.getPieceColor() == PieceColor.WHITE);
assert(!(kn2.getPieceColor() == PieceColor.WHITE));
assert(r.getPieceColor() == PieceColor.WHITE);
assert(!(r2.getPieceColor() == PieceColor.WHITE));

assert(!(p.getPieceColor() == PieceColor.BLACK));
assert(p2.getPieceColor() == PieceColor.BLACK);
assert(!(k.getPieceColor() == PieceColor.BLACK));
assert(k2.getPieceColor() == PieceColor.BLACK);
assert(!(q.getPieceColor() == PieceColor.BLACK));
assert(q2.getPieceColor() == PieceColor.BLACK);
assert(!(b.getPieceColor() == PieceColor.BLACK));
```

```
assert(b2.getPieceColor() == PieceColor.BLACK);
assert(!(kn.getPieceColor() == PieceColor.BLACK));
assert(kn2.getPieceColor() == PieceColor.BLACK);
assert(!(r.getPieceColor() == PieceColor.BLACK));
assert(r2.getPieceColor() == PieceColor.BLACK);


//Pawn

assert(p.canMove("e2", "e4", board));
board.setPiece("e3", p2);
assert(!p.canMove("e2", "e4", board));
assert(p.canTake("f2", "e3", board));
assert(p.canMove("f3", "f4", board));
assert(p2.canMove("e7", "e5", board));
board.setPiece("e6", p);
assert(!p2.canMove("e7", "e5", board));
assert(p2.canTake("e3", "f2", board));
assert(p2.canMove("f4", "f3", board));

board.setPiece("e3", null);
board.setPiece("e6", null);

assert(!p.canTake("f3", "f4", board));
assert(!p.canMove("g4", "g3", board));
assert(!p.canMove("g4", "h3", board));
assert(!p.canTake("g4", "g3", board));
assert(!p.canTake("g4", "f3", board));
assert(!p2.canTake("f3", "f4", board));
assert(p2.canMove("g4", "g3", board));
assert(!p2.canMove("g4", "h3", board));
assert(!p2.canTake("g4", "g3", board));
assert(p2.canTake("g4", "f3", board));

assert(!p2.canMove("c4", "d4", board));
assert(!p.canMove("c4", "d4", board));

//King

assert(k.canMove("e1", "e2", board));
assert(!k.canMove("e1", "e3", board));
assert(!k.canMove("e1", "c3", board));
assert(!k.canMove("e1", "d3", board));
assert(k.canMove("e1", "d2", board));
assert(k.canMove("d5", "e5", board));

assert(k.canTake("e1", "e2", board));
assert(!k.canTake("e1", "e3", board));
assert(!k.canTake("e1", "c3", board));
```

```
assert(!k.canTake("e1", "d3", board));
assert(k.canTake("e1", "d2", board));
assert(k.canTake("d5", "e5", board));

//Queen

assert(q.canMove("e1", "e2", board));
assert(q.canMove("e1", "e3", board));
assert(q.canMove("e1", "c3", board));
assert(!q.canMove("e1", "d3", board));
assert(q.canMove("e1", "d2", board));
assert(q.canMove("d5", "e5", board));

assert(q.canTake("e1", "e2", board));
assert(q.canTake("e1", "e3", board));
assert(q.canTake("e1", "c3", board));
assert(!q.canTake("e1", "d3", board));
assert(q.canTake("e1", "d2", board));
assert(q.canTake("d5", "e5", board));
assert(q.canTake("a1", "h1", board));
assert(q.canTake("a1", "h8", board));
assert(!q.canTake("a1", "h6", board));
assert(!q.canMove("a1", "h6", board));
board.setPiece("b1", k);
assert(!q.canTake("a1", "h1", board));

board.setPiece("b1", null);

//Bishop

assert(b.canMove("c1", "f4", board));
assert(b.canMove("f8", "h6", board));
assert(b.canTake("c1", "f4", board));
assert(b.canTake("f8", "h6", board));

assert(!b.canMove("c1", "f5", board));
assert(!b.canMove("f3", "h6", board));
assert(!b.canTake("c2", "f4", board));
assert(!b.canTake("f7", "h6", board));
assert(b.canMove("a1", "h8", board));
assert(!b.canMove("a1", "h1", board));

//Rook

assert(r.canMove("h1", "a1", board));
assert(r.canMove("a8", "a1", board));
assert(r.canTake("a4", "e4", board));
board.setPiece("d4", p);
assert(!r.canTake("a4", "e4", board));
```

```
board.setPiece("d4", null);
assert(!r.canMove("a1", "h8", board));
assert(!r.canMove("f4", "d2", board));

//Knight

assert(kn.canMove("g1", "f3", board));
assert(kn.canTake("f6", "d5", board));
assert(!kn.canMove("a1", "h8", board));
assert(!kn.canMove("a1", "a2", board));
assert(!kn.canMove("g8", "h8", board));
assert(!kn.canMove("e6", "c8", board));
assert(!kn.canMove("f1", "d3", board));
}

private static void assertE1(){

Board board = new Board();

assert (board.getPiece("a1") instanceof Piece);
assert (board.getPiece("b1") instanceof Piece);
assert (board.getPiece("c1") instanceof Piece);
assert (board.getPiece("d1") instanceof Piece);
assert (board.getPiece("e1") instanceof Piece);
assert (board.getPiece("f1") instanceof Piece);
assert (board.getPiece("g1") instanceof Piece);
assert (board.getPiece("h1") instanceof Piece);
assert (board.getPiece("a2") instanceof Piece);
assert (board.getPiece("b2") instanceof Piece);
assert (board.getPiece("c2") instanceof Piece);
assert (board.getPiece("d2") instanceof Piece);
assert (board.getPiece("e2") instanceof Piece);
assert (board.getPiece("f2") instanceof Piece);
assert (board.getPiece("g2") instanceof Piece);
assert (board.getPiece("h2") instanceof Piece);

assert (board.getPiece("a3") == null);
assert (board.getPiece("b3") == null);
assert (board.getPiece("c3") == null);
assert (board.getPiece("d3") == null);
assert (board.getPiece("e3") == null);
assert (board.getPiece("f3") == null);
assert (board.getPiece("g3") == null);
assert (board.getPiece("h3") == null);
assert (board.getPiece("a4") == null);
assert (board.getPiece("b4") == null);
assert (board.getPiece("c4") == null);
assert (board.getPiece("d4") == null);
assert (board.getPiece("e4") == null);
```

```
assert (board.getPiece("f4") == null);
assert (board.getPiece("g4") == null);
assert (board.getPiece("h4") == null);
assert (board.getPiece("a5") == null);
assert (board.getPiece("b5") == null);
assert (board.getPiece("c5") == null);
assert (board.getPiece("d5") == null);
assert (board.getPiece("e5") == null);
assert (board.getPiece("f5") == null);
assert (board.getPiece("g5") == null);
assert (board.getPiece("h5") == null);
assert (board.getPiece("a6") == null);
assert (board.getPiece("b6") == null);
assert (board.getPiece("c6") == null);
assert (board.getPiece("d6") == null);
assert (board.getPiece("e6") == null);
assert (board.getPiece("f6") == null);
assert (board.getPiece("g6") == null);
assert (board.getPiece("h6") == null);

assert (board.getPiece("a7") instanceof Piece);
assert (board.getPiece("b7") instanceof Piece);
assert (board.getPiece("c7") instanceof Piece);
assert (board.getPiece("d7") instanceof Piece);
assert (board.getPiece("e7") instanceof Piece);
assert (board.getPiece("f7") instanceof Piece);
assert (board.getPiece("g7") instanceof Piece);
assert (board.getPiece("h7") instanceof Piece);
assert (board.getPiece("a8") instanceof Piece);
assert (board.getPiece("b8") instanceof Piece);
assert (board.getPiece("c8") instanceof Piece);
assert (board.getPiece("d8") instanceof Piece);
assert (board.getPiece("e8") instanceof Piece);
assert (board.getPiece("f8") instanceof Piece);
assert (board.getPiece("g8") instanceof Piece);
assert (board.getPiece("h8") instanceof Piece);

assert (board.getPiece("a1") instanceof Rook);
assert (board.getPiece("b1") instanceof Knight);
assert (board.getPiece("c1") instanceof Bishop);
assert (board.getPiece("d1") instanceof Queen);
assert (board.getPiece("e1") instanceof King);
assert (board.getPiece("f1") instanceof Bishop);
assert (board.getPiece("g1") instanceof Knight);
assert (board.getPiece("h1") instanceof Rook);
assert (board.getPiece("a2") instanceof Pawn);
assert (board.getPiece("b2") instanceof Pawn);
assert (board.getPiece("c2") instanceof Pawn);
assert (board.getPiece("d2") instanceof Pawn);
```

```
assert (board.getPiece("e2") instanceof Pawn);
assert (board.getPiece("f2") instanceof Pawn);
assert (board.getPiece("g2") instanceof Pawn);
assert (board.getPiece("h2") instanceof Pawn);

assert (board.getPiece("a8") instanceof Rook);
assert (board.getPiece("b8") instanceof Knight);
assert (board.getPiece("c8") instanceof Bishop);
assert (board.getPiece("d8") instanceof Queen);
assert (board.getPiece("e8") instanceof King);
assert (board.getPiece("f8") instanceof Bishop);
assert (board.getPiece("g8") instanceof Knight);
assert (board.getPiece("h8") instanceof Rook);
assert (board.getPiece("a7") instanceof Pawn);
assert (board.getPiece("b7") instanceof Pawn);
assert (board.getPiece("c7") instanceof Pawn);
assert (board.getPiece("d7") instanceof Pawn);
assert (board.getPiece("e7") instanceof Pawn);
assert (board.getPiece("f7") instanceof Pawn);
assert (board.getPiece("g7") instanceof Pawn);
assert (board.getPiece("h7") instanceof Pawn);

}

private static void assertE2(){

Board board = clearBoard(new Board());

//movePiece

Piece piece1 = new Piece() {

@Override
public PieceColor getPieceColor() {
// TODO Auto-generated method stub
return null;
}

@Override
public boolean canTake(String from, String to, Board board) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean canMove(String from, String to, Board board) {
// TODO Auto-generated method stub
return false;
}
```

```
};
Piece piece2 = new Piece() {

@Override
public PieceColor getPieceColor() {
// TODO Auto-generated method stub
return null;
}

@Override
public boolean canTake(String from, String to, Board board) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean canMove(String from, String to, Board board) {
// TODO Auto-generated method stub
return false;
}
};

board.setPiece("e5", piece1);
board.setPiece("g8", piece2);

assert(board.getPiece("e5") == piece1);
assert(board.getPiece("g8") == piece2);

assert(board.getPiece("g8") != piece1);
assert(board.getPiece("e5") != piece2);

assert(board.getPiece("g8") != null);
assert(board.getPiece("e5") != null);

board.movePiece("e5", "g8");

assert(board.getPiece("e5") == null);
assert(board.getPiece("e5") != piece1);
assert(board.getPiece("e5") != piece2);
assert(board.getPiece("g8") == piece1);
assert(board.getPiece("g8") != piece2);

//isCheck

board = clearBoard(board);

King k = new King(PieceColor.WHITE);
King k2 = new King(PieceColor.BLACK);
Bishop b2 = new Bishop(PieceColor.BLACK);
```

```
Rook r2 = new Rook(PieceColor.BLACK);
Knight kn2 = new Knight(PieceColor.BLACK);
Queen q2 = new Queen(PieceColor.BLACK);
Pawn p2 = new Pawn(PieceColor.BLACK);
Pawn blocker = new Pawn(PieceColor.WHITE);

board.setPiece("e4", k);
//Pawn
board.setPiece("e5", p2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("e5", null);

board.setPiece("f5", p2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("f5", null);

board.setPiece("d3", p2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d3", null);

board.setPiece("d6", p2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d6", null);

board.setPiece("d5", p2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d5", null);

//King
board.setPiece("e5", k2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("e5", null);

board.setPiece("f5", k2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("f5", null);

board.setPiece("d3", k2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d3", null);

board.setPiece("d6", k2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d6", null);

board.setPiece("d5", k2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d5", null);
```

```
board.setPiece("f3", k2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("f3", null);

board.setPiece("d4", k2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d4", null);

board.setPiece("f4", k2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("f4", null);

board.setPiece("c4", k2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("c4", null);

//Knight
board.setPiece("f6", kn2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("f6", null);

board.setPiece("d6", kn2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d6", null);

board.setPiece("g5", kn2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("g5", null);

board.setPiece("g3", kn2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("g3", null);

board.setPiece("c3", kn2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("c3", null);

board.setPiece("c5", kn2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("c5", null);

board.setPiece("d2", kn2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d2", null);

board.setPiece("f2", kn2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("f2", null);
//---------------------------------------
```

```
board.setPiece("e5", kn2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("e5", null);

board.setPiece("f5", kn2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("f5", null);

board.setPiece("d3", kn2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d3", null);

board.setPiece("d5", kn2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d5", null);

board.setPiece("f3", kn2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("f3", null);

board.setPiece("d4", kn2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d4", null);

board.setPiece("f4", kn2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("f4", null);

//Queen
board.setPiece("e5", q2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("e5", null);

board.setPiece("f5", q2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("f5", null);

board.setPiece("d3", q2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d3", null);

board.setPiece("d6", q2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d6", null);

board.setPiece("d5", q2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d5", null);
```

```
board.setPiece("f3", q2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("f3", null);

board.setPiece("d4", q2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d4", null);

board.setPiece("f4", q2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("f4", null);

board.setPiece("c4", q2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("c4", null);

board.setPiece("c6", q2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d5", blocker);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d5", null);
board.setPiece("c6", null);

board.setPiece("e8", q2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("e6", blocker);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("e6", null);
board.setPiece("e8", null);

board.setPiece("h1", q2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("g2", blocker);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("g2", null);
board.setPiece("h1", null);

board.setPiece("g3", q2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("f3", blocker);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("f3", null);
board.setPiece("g3", null);

//Bishop
board.setPiece("e5", b2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("e5", null);
```

```
board.setPiece("f5", b2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("f5", null);

board.setPiece("d3", b2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d3", null);

board.setPiece("d6", b2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d6", null);

board.setPiece("d5", b2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d5", null);

board.setPiece("f3", b2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("f3", null);

board.setPiece("d4", b2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d4", null);

board.setPiece("f4", b2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("f4", null);

board.setPiece("c4", b2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("c4", null);

board.setPiece("c6", b2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d5", blocker);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d5", null);
board.setPiece("c6", null);

board.setPiece("e8", b2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("e6", blocker);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("e6", null);
board.setPiece("e8", null);

board.setPiece("h1", b2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("g2", blocker);
```

```
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("g2", null);
board.setPiece("h1", null);

board.setPiece("g3", b2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("f3", blocker);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("f3", null);
board.setPiece("g3", null);

//Rook
board.setPiece("e5", r2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("e5", null);

board.setPiece("f5", r2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("f5", null);

board.setPiece("d3", r2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d3", null);

board.setPiece("d6", r2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d6", null);

board.setPiece("d5", r2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d5", null);

board.setPiece("f3", r2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("f3", null);

board.setPiece("d4", r2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d4", null);

board.setPiece("f4", r2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("f4", null);

board.setPiece("c4", r2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("c4", null);

board.setPiece("c6", r2);
```

```
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d5", blocker);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d5", null);
board.setPiece("c6", null);

board.setPiece("e8", r2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("e6", blocker);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("e6", null);
board.setPiece("e8", null);

board.setPiece("h1", r2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("g2", blocker);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("g2", null);
board.setPiece("h1", null);

board.setPiece("g3", r2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("f3", blocker);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("f3", null);
board.setPiece("g3", null);

//Multiple

board.setPiece("a3", p2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("g3", r2);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("f3", blocker);
assert(!board.isCheck(PieceColor.WHITE));
board.setPiece("d6", kn2);
assert(board.isCheck(PieceColor.WHITE));
board.setPiece("d6", null);
board.setPiece("f3", null);
board.setPiece("g3", null);

//isLeagalMove

board = clearBoard(board);

//Pawn
board.setPiece("e7", p2);
assert(!board.isLegalMove(p2.getPieceColor(), "e7", "e8"));
assert(!board.isLegalMove(p2.getPieceColor(), "e7", "e7"));
```

```
assert(board.isLegalMove(p2.getPieceColor(), "e7", "e6"));
assert(board.isLegalMove(p2.getPieceColor(), "e7", "e5"));
board.setPiece("e6", blocker);
assert(!board.isLegalMove(p2.getPieceColor(), "e7", "e5"));
board.setPiece("e6", null);
board.setPiece("e5", blocker);
assert(!board.isLegalMove(p2.getPieceColor(), "e7", "e5"));
board.setPiece("e5", null);
assert(!board.isLegalMove(p2.getPieceColor(), "e7", "d6"));
board.setPiece("d6", blocker);
assert(board.isLegalMove(p2.getPieceColor(), "e7", "d6"));
board.setPiece("d6", null);
assert(!board.isLegalMove(p2.getPieceColor(), "e7", "f6"));
board.setPiece("f6", blocker);
assert(board.isLegalMove(p2.getPieceColor(), "e7", "f6"));
board.setPiece("f6", null);
assert(!board.isLegalMove(p2.getPieceColor(), "e7", "d8"));
assert(!board.isLegalMove(p2.getPieceColor(), "e7", "f8"));
assert(!board.isLegalMove(p2.getPieceColor(), "e7", "d5"));
assert(!board.isLegalMove(p2.getPieceColor(), "e7", "f5"));

board.setPiece("e8", k2);
board.setPiece("e2", new Rook(PieceColor.WHITE));
board.setPiece("f6", blocker);
assert(!board.isLegalMove(p2.getPieceColor(), "e7", "f6"));
board.setPiece("d6", new Knight(PieceColor.WHITE));
assert(!board.isLegalMove(p2.getPieceColor(), "e7", "e6"));
assert(!board.isLegalMove(p2.getPieceColor(), "e7", "d6"));
assert(board.isLegalMove(p2.getPieceColor(), "e8", "d7"));
board.setPiece("e2", null);
assert(board.isLegalMove(p2.getPieceColor(), "e7", "d6"));
board.setPiece("d7", null);
board.setPiece("e8", null);
board.setPiece("e2", null);
board.setPiece("f6", null);

board.setPiece("e7", null);

//King
board.setPiece("f4", k);

board.setPiece("d5", b2);
assert(!board.isLegalMove(k.getPieceColor(), "f4", "e4"));
assert(board.isLegalMove(k.getPieceColor(), "f4", "f5"));
board.setPiece("e1", q2);
assert(!board.isLegalMove(k.getPieceColor(), "f4", "e3"));
board.setPiece("h2", kn2);
assert(!board.isLegalMove(k.getPieceColor(), "f4", "g4"));
assert(board.isLegalMove(kn2.getPieceColor(), "h2", "g4"));
```

```
board.setPiece("f4", null);

//Knight

board.setPiece("h1", k2);
assert(board.isLegalMove(kn2.getPieceColor(), "h2", "g4"));
board.setPiece("h6", new Rook(PieceColor.WHITE));
assert(!board.isLegalMove(kn2.getPieceColor(), "h2", "g4"));
board.setPiece("h6", null);
board.setPiece("f1", new Rook(PieceColor.WHITE));
assert(!board.isLegalMove(kn2.getPieceColor(), "h2", "g4"));
assert(board.isLegalMove(kn2.getPieceColor(), "h2", "f1"));


}

public static void main(String[] args) {
System.out.println("Starting assertion...");


// System.out.print("Asserting A...");
// assertA();
// System.out.println("Done");
// System.out.print("Asserting B...");
// assertB();
// System.out.println("Done");
System.out.print("Asserting C...");
assertC();
System.out.println("Done");
// System.out.print("Asserting D...");
// assertD();
// System.out.println("Done");
// System.out.print("Asserting E:");
// System.out.print("E1...");
// assertE1();
// System.out.print("Done.");
// System.out.print("E2...");
// assertE2();
// System.out.print("Done.");
// System.out.println("Asserting E Done");

System.out.println("Assertion Done");

// String a = "a5";
// System.out.println((int)a.charAt(0) - 'a');

}

}
```

Bibliography

[1] Bell, J., and Holroyd, J. Review of human reliability assessment methods. Tech. rep., Health and Safety Laboratory for the Health and Safety Executive 2009 (UK), 2009.

[2] Gertman, D., Blackman, H., Marble, J., Byers, J., and Smith, C. The spar-h human reliability analysis method. Tech. rep., Idaho National Laboratory, 2005.

[3] Hollnagel, E. Reliability analysis and operator modelling. *Reliability Engineering and System Safety 52* (1996), 327–337.

[4] Kirwan, B. *A Guide to Practical Human Reliability Assessment.* Taylor & Francis Ltd, 1994.

[5] Stanton, N. A. Hierarchial task analysis: Development, applications, and extensions. *Applied Ergonomics 37* (2006), 55–79.

[6] Swain, A. D. Human reliability analysis: Need, status, trends and limitations. *Reliability Engineering and System Safety 29* (1990), 301–313.