Developing a serious game to aid managers in solving coordination-problems in software development

Gijs F.P. van Veen
g.f.p.vanveen@student.utwente.nl

ABSTRACT
In this paper the development of a serious game based on the Technical and Social Network Analysis, abbreviated to TESNA, decision support system is discussed. This serious game can be used to assist managers to improve their decision making abilities when dealing with coordination problems in software development.

Keywords
Decision support systems, gaming, simulation, evaluation, serious game, problems in software development, decision making.

1. INTRODUCTION
For centuries, gaming has been used for decision making. It was used in ancient Chinese warfare [1, 2] and in the 1950s simulation games became popular to use in urban planning and business management [2, 3]. Lardinois discusses several different ways to classify simulation games [4]. One of these definitions deals with systemic games versus role-playing games. While the former focuses on “interrelationships and interdependencies between elements of an economic, political or social system where players develop their own role behaviours” the latter focuses on “given particular positions in the real world and relationships between these positions” (p. 853, [4]). Angelides and Paul help us further define simulation-gaming, noting that it is a sequential decision-making exercise, where an artificial environment that has some characteristics similar to the real world [5]. In this environment players can learn to respond quickly to the consequences of their actions. This helps them to better understand the real world the simulation game tries to replicate [5]. In recent years these non-entertainment based games have also become known as “serious games” [6].

Another way to assist in decision making is a decision support system (DSS) [7, 8]. Since both decision support systems and simulation games are used to support decision making it would seem like the logical next step to combine both of these methods. Though Khoong already argued for the usage of Decision Support Systems for gaming in 1995 [7], it was not until a decade later that combining DSS with gaming was researched any further. It was then found that there are four different ways in which DSS and gaming could be used complimentary [9]. These four ways are as follows:

1. DSS for game design, where DSS are used to speed up the process of game design by assisting the design choices.

2. DSS for game evaluation, where DSS are either used to evaluate the game itself or to generate results to compare with the results of the game.

3. DSS for game operation, where DSS are used to either play the game or to support running of the game itself.

4. Gaming for DSS research, testing and training, where gaming techniques are used to develop, test and study a new DSS.

It should be noted that these four methods of combining DSS with gaming can be used in combination with other methods and thus are not mutual exclusive [9].

Even though it has been considered to combine DSS with gaming for a while now, examples of these combinations can only be found sporadically. In 2006 such a combination was used in research of a business network [10]. More recently, in 2008, a combination of DSS and gaming was used for decision making in water management [11].

The main objective of this paper is to develop a serious game that aids its players, mainly the managers of software companies, in their decision making when dealing with coordination problems in software development by using the methods found in these researches.

2. MOTIVATION
There have been many researches on the causes of problems in software development. Though the problems cannot be brought down to a single cause. Kraut and Streeter argued that a major factor is the problem of coordinating activities while developing large software projects [12]. According to Brooks software entities are essentially complex, which means that the complexity is a part of the software [13]. Brooks then argued that this complexity would make the communications between team members difficult. This difficulty results in buggy software, which in its turn leads to delays and cost overruns. Not only will the essential complexity lead to technical problems, but managerial problems as well [13]. The complexity of the software makes it hard for a manager to get a proper overview of the developed software and at the same time makes it harder to identify and control software problems.

Amrit developed a tool that detects several different coordination problems in software development [14]. This tool, the TEchnical and Social Network Analysis (TESNA) tool, is a decision support system that analyses the company structure and finds the different problems in software development, helping managers to give better response to these problems [14].

Though the TESNA-tool on itself is capable of detecting different coordination problems for a manager, the visualisations displayed by TESNA need to be understood and interpreted by the manager. To aid the manager in learning the semantics of the coordination problems present in the different visualisations and improve their ability to detect these problems themselves a simulation environment is needed. The aim of this research is to develop such a simulation gaming environment.
In 2007 Pivec discussed the possibilities of using serious games for learning objectives [15]. In this research a serious game based on the TESNA-tool is developed. The aim of the game is to enable players to learn to detect coordination problems themselves and respond to them in a proper way. Mayer and de Jong already showed the possibilities of developing a decision support system into a game and the TESNA-tool could benefit from a similar approach [9].

3. RESEARCH SUBJECT

3.1 Research Question

In this paper the aforementioned serious game based on the TESNA-tool is researched in order to improve the ability of players to detect different coordination problems in software development. The main question that will be answered in this paper is:

*How can we create a serious game based on the TESNA DSS so that players will improve their ability to detect and solve coordination problems in software development?*

To design such a game the design science methodology as well as some principles of game design is used.

3.2 Design Science

The primary research method of this paper is the design science approach. Since the main objective of this research is the creation of an artefact the design science research methodology [16] are used.

Hevner et al described how to perform design science in the field of information systems, allowing their methodology to be used for this system [17]. According to this methodology there are seven different guidelines for performing a design research that have to be used. The different guidelines and their relation to this research are as follows.

1. **Design as Artefact:** The design science must produce a valuable artefact, which can be in the form of a model, construct, method or instantiation. The serious game that is to be designed in this research will be an artefact.

2. **Problem Relevance:** The objective of the research is to give a technical solution to an important business problem. Since our serious game aims to help players in solving problem in software development, the game can be viewed as a technical solution.

3. **Design evaluation:** It must be shown through testing that the designed artefact will achieve its goals. In our research we design an evaluation to test whether the designed serious game indeed improves the decision making abilities of players.

4. **Research contributions:** The research done must give a valuable contribution to the design artefact, foundations and/or methodologies. The developed serious game is the main contribution of this research, though it may also give some insights into the combinations of decision support systems and serious games.

5. **Research rigor:** The research relies on applying rigorous methods in both construction and evaluation. For this methodologies on serious game design, as specified below, will be followed.

6. **Design as a search process:** The design is a search to an effective way to solve the problem specified. By using a serious game, a way to achieve the learning objective of this system was found. Effective ways to achieve the goals of this research will be searched for from within the game.

7. **Communication of Research:** The research done should be communicated to both technological- and managerial-audiences. This communication is done through this paper, which will show the results of this research.

Due to the time constraints of this research, the main focus of the research is on the first four guidelines, though each guideline will be covered.

3.3 Design of Artifact

Since Hevner et al [17] do not deal with the design of the artefact itself in much detail, the Function-Behaviour-Structure (FBS)-framework as specified by Gero [18, 19] is used. In the framework three separate classes of variables, describing different aspects of the artefact, are defined as follows.

- **Function (F) variables**, describing what the artefact is for.
- **Behaviour (B) variables**, describing what the artefact does.
- **Structure (S) variables**, describing the different components of the artefact and their relations to each other.

The FBS-framework represents eight processes that are said to be fundamental in designing an artefact. Figure 1 shows the entire FBS-framework.

**Figure 1. The Function-Behaviour-Structure framework** [18].

Due to the scope and time constraints of this research not all processes are considered equally important. Main focus of this research is on the following processes:

- **Formulation** (process 1): Transforms the requirements into expected behaviour. This is specified before the implementation of the game, in order to be able to test the game properly.
- **Synthesis** (process 2): Transforms the expected behaviour into the solution. This is done by implementing the game itself.
- **Analysis** (process 3): See how the actual behaviour of the artefact is. This is done by testing the game.
- **Evaluation** (process 4): See whether how actual behaviour of the game compares to the expected behaviour. This is done after the testing of the game.

If the results of the evaluation are not satisfied, processes 6 through 8 will be used to adjust the structure, expected behaviour or function respectively until the results of the evaluation are satisfied. Process 5 is used solely for documentation.

3.4 Serious game design

The Function-Behaviour-Structure framework does not discuss the design of games. Since the aim of this research is to design a serious game, some specific requirements for games are needed.
According to Salen and Zimmerman the goal of successful game design is the creation of meaningful play [19].

Meaningful play can be defined in two different ways. In the descriptive definition, “meaningful play in a game emerges from the relationship between player action and system outcome,” so that “the meaning of an action in the game resides in the relationship between action and outcome.” In the evaluative definition “meaningful play is what occurs when the relationships between action and outcome are both discernable and integrated into the larger context of the game.” This means that a player can see the immediate outcome of an action and that the outcome of the action is woven into the game system as a whole [19].

Salen and Zimmerman continue by noting that a game is a system, that is defined as a set of parts that form a complex whole [19]. There are four elements that all systems share [20]:

- Objects, parts elements or variables within the system.
- Attributes, qualities or properties of the system and its objects.
- Internal relationships, relations among the objects.
- Environment, context that surrounds the system.

To identify these elements a game can be framed as either a formal, experiential or cultural system. Though all of these frames exist simultaneously Salen and Zimmerman note that focus should be on just one of the frames at a time when designing a game [19].

To have meaningful play emerge from a system of a game interactivity is required [19]. An interactive context presents the participants with choices that can be split up into micro-choices, which are made of moment-to-moment interactivity and macro-choices, which concern the long-term progress of the game-experience. To give interaction meaning each action should give a certain outcome. It should be noted that the effects of actions should be properly communicated to the player, so that it is possible for the player to understand these effects.

Not only should actions be properly communicated to the player, they should also be consistent within the game. This means that it is necessary to define different rules for a game. Rules help give a game meaning and allow the player to understand the consequences of his actions [19].

Eventually the choices made in the game will have to lead to a quantifiable goal or outcome. In this way, a player has won, lost or received some form of score. This allows the game to have meaningful play [19].

The meaning of the game to be designed in this research is to solve coordination problems in software development. Some of the rules in the game are similar to how the different coordination problems are detected, while the eventual goal of the game is to not have any problems left.

4. TESNATool

In order to create this serious game, it is required to have a closer look at the TESNATool itself. As specified by Amrit [14], the TESNATool is build to detect coordination problems in software companies by reviewing the relations between persons and software modules. In order to detect these problems, three different matrices should be constructed. By using different patterns on these matrices, the tool will thus determine what problems occur in the system. The three matrices we consider are constructed by the Social Structure, the Technical Structure and the Socio-Technical Structure of the system [14]. Let’s have a closer look at the different structures.

4.1 Dependency Structure Matrix

The matrices used in the TESNATool are in the form of a dependency structure matrix. A dependency structure matrix is a common way for representing dependencies between different tasks within a business process [21]. When the matrix has a value Dependencies below the gray line represent feed forward information, while dependencies above the gray line represent feedback. Figure 2 shows an example of a dependencies structure matrix.

![Figure 2. Example of a dependency structure matrix](image)

In this example, some processes only give feedback, while others only give information. Process 2 and 3 are interdependent however, as they give both information and feedback. In the TESNATool, only interdependency has to be considered due to the nature of the given matrices.

4.2 Matrices in TESNATool

According to Wasserman and Faust the Social Network "consists of a finite set or sets of actors and the relation or relations defined on them" [22]. In the TESNATool, the persons in a company are considered the actors. Relationships between actors can be determined by the TESNATool by mining Chat, Mail and Bug tracker Repositories and can be refined by qualitative interviews and questionnaires [14].

The social network can then be converted into a dependency structure matrix, showing the relations between all actors. In the TESNATool, this matrix is referred to as the social structure of the system [14]. Note that this matrix is interdependent, as communication is a bidirectional process.

According to Curtis et al large software systems are split up into smaller parts by the use of decomposition and different small teams work on different parts of the system [23]. This will create dependencies between some of the different parts, as one module may use functionality of another module.

In the TESNATool, these dependencies between software modules are stored in a second matrix, the Technical Structure. When given the Software Architecture or the software code, the TESNATool will be able to construct this matrix [14]. This matrix is not strictly interdependent, as one process can rely on another without being relied on. Since all other matrices are interdependent however, the matrix can be converted into an interdependent matrix with the same results over the patterns the TESNATool uses.

Since Curtis et al suggested to split software projects into multiple smaller projects with different teams working on them [23], it seems necessary to have a third matrix to determine what person or team is working on each project. We call this third matrix the socio-technical structure of the system.

Even though the socio-technical structure is considered a dependency structure matrix, it differs from this structure to some extent. Since software modules are dependent to persons in this matrix, the diagonal line of the matrix is no longer a filled in area. For example, module one could depend on person one, while module two could depend on person three instead of on person two. The resulting dependency structure matrix is considered interdependent, as a module on which a person depends equally depends on the same person.
In the TESNATool, this matrix can be constructed by the manager, who determines which person or team is assigned to a project.

4.3 Patterns
Since the TESNATool aims to detect different communication problems in the structure of the company, it is necessary to specify the different problems that should be detected. These specific coordination problems the TESNATool focuses on are Socio/Technical Structure Clashes (STSCs). A STSC occurs if and when a Socio/Technical Pattern exists that indicates that the social structure of the system does not match the social/technical dependencies within the software architecture under development. Amrit discusses several patterns that are used in the TESNATool [14]. Though multiple patterns are discussed, this paper will only focus on two of them, due to time constraints of the research.

4.4 Code Ownership Pattern
The Code Ownership STSC is based on the Code Ownership Pattern as described by Coplien [24] and Nordberg [25]. The coordination problem related with this problem is that a developer would find it difficult to cope with the constant changing base of code. Most design knowledge lies in the code, so finding errors in unfamiliar code takes more time than finding them in familiar code. Something similar occurs when a developer who previously was not involved with the software project is given the responsibility of the code. In this case a lot of coordination would be required to get the new developer instructed on the history of the project, which take up a lot of extra development time.

A suggested solution for this problem is to assign a developer to each software module. This developer is then considered the owner of the code [24, 25]. Since only owners are allowed to change the code developers will always be familiar with the code they wrote.

To detect these problems in the system the TESNATool implements the code ownership pattern by checking the socio-technical structure matrix [14]. Since each software module needs to be owned by a developer, the tool checks whether there is at least one developer dependent on each software module. If this is not the case a coordination problem is detected.

4.5 Conway’s Law
Another STSC the TESNATool detects is based on the coordination problem known as Conway’s Law, first described by Conway in his classic paper [26]. Conway stated that organisations are constrained to design systems that copy the communicational structure of the organisation. In short, if software modules are dependent upon each other, there should be communication between the developers that own the software modules. According to Conway [26], if the teams involved in the software development would have shortcomings in their interpersonal relationships, the product they produce would contain flaws as well.

To detect the communication problems described by Conway’s Law, the TESNATool uses the algorithm as described by Cataldo et al [14, 27]. According to this algorithm [27], the three previously described matrices should be used to detect the coordination problems. By multiplying the socio-technical structure dependency matrix with the technical structure dependency matrix the developer by module matrix is obtained. This matrix represents the tasks a developer should be aware of, given the software modules the developer owns and the dependencies of those modules with other modules. The people by module matrix should then be multiplied with the transpose of the socio-technical structure dependency matrix, creating a developer by developer matrix. This matrix shows which developers each developer should communicate with. This matrix is called the required social structure dependency matrix, as it shows the required communicational structure. By subtracting the actual social structure dependency matrix from the required social structure matrix a final matrix is created. This matrix shows all missing communications between developers. If no missing communications are found, Conway’s Law is satisfied.

5. REQUIREMENTS
The first stage of software design is the formulation of the requirements, equivalent to the first phase in the Function-Behaviour-Structure framework [18]. The formulation of requirements will define a scope in which the project must fit. This gives a good description of what the system should do.

According to Laueson formulating requirements has several advantages [28]. Not only does formulating requirements reduce development time drastically, a main advantage in projects with a strict deadline, like this research, but it also gives the design a goal-directed approach. Clear requirements help in finding the direction of the research and actual implementation of the software [28].

Since the main goal of this research is to design a serious game that implements the TESNATool and helps players learn the coordination problems as specified by this tool, the requirements can be split up into the following three main requirements:

1. The system implements the TESNATool.
2. The system will be a serious game.
3. The system will help players learn to detect coordination problems in software development.

It can be noted that the third requirement only enhances the second. A serious game is defined to teach its players something and requirements three defines what the serious game should be learning to the player exactly. These main requirements can then be split up into multiple smaller requirements, which specify what the system should do and how it should do this.

5.1 TESNATool Requirements
Since the game will only simulate the effects of the TESNATool and not all functionality will be required in the game, the game is required to implement the TESNATool simulation environment as discussed by Nijmeijer [29].

In order to fully implement the TESNATool simulation environment, the game has to be able to set up the simulation properly. The requirements of the simulation environment must thus also be satisfied in the serious game. This gives the following requirements [29]:

- The game contains persons and software modules.
- Persons can be connected to each other, meaning they can communicate with each other. This connection is called a communication line.
- Software modules can be connected to each other, meaning they are dependent upon each other. This connection is referred to as a dependency.
- Persons can be connected with software modules, meaning they are working on the corresponding module. This connection is called a task connection.

The state of the system, i.e. the persons, software modules and connections between them in the simulation environment, at a given time is referred to as the configuration of the software project at this time [29]. Since the goal of the TESNATool is to
detect different coordination problems in software development in a company system, another requirement can be added.

- It is possible to use all of the patterns the TESNATool implements at any given configuration of the software project, resulting in the number of conflicts that occur within the system.

5.2 Game Requirements
In order to achieve the goals of a serious game, it is required to have a proper view of the requirements of a game in general. Specifying the requirements for a system to be a game gives a better definition of the requirements of a specific serious game. The definition of a game given by Salen and Zimmerman [19] give some requirements for the system, as discussed earlier.

- The game should derive meaningful play.
- The outcome of the game should be discernable, meaning that the player can perceive the immediate outcome of an action. This allows the player to evaluate how well or how poorly he or she has performed.
- The outcome should be integrated, meaning that the outcome of an action is woven into the system as a whole.
- The game should consist of Objects, Attributes, Internal Relationships and Environment.
- The game should be interactive, allowing the player to decide what actions should be done.
- The game should allow the player to choose what action to perform.
- The game should have a valid end state or goal that the player must reach.

But most of all, Salen and Zimmerman insist that a game should be fun to play [19].

5.3 Learning Requirements
Now that the requirements for a game in general have been specified, the requirements for the learning purposes of the game can be defined. This can be split into two different categories. First there are the general requirements that have to be satisfied to have a serious game and second are the requirements that have to be satisfied in order to specify what the game should be learning to its players.

Since learning is not an exact science, it is difficult to give clear requirements a system should have in order to learn its users something. In the field of serious games however, some research has been done [30]. However, most of the requirements found by these researches are similar to the requirements of a game in general. This includes the necessity to have clear rules, communicate the outcome of player actions and a clear representation of the real world.

The main requirement for the serious game however, remains the fun-factor. As noted by Bisson and Luckner "Enjoyment and fun as part of the learning process are important when learning new tools since the learner is relaxed and motivated and therefore more willing to learn" [31].

The remaining requirements, that specify what exactly the game should be learning, are mostly specified by the TESNATool requirements. The goal of the game is to teach its players to detect coordination problems in software management and thus the main learning requirement seems to be centred on these coordination problems. This leads to the following requirements:

- The game should let its players detect all conflicts that the TESNATool is able to detect.
- The game should let its players solve all conflicts that the TESNATool is able to detect.

6. GAMEDESIGN
Even though the requirements specify what the system should do, the structure of the game has not been given. The game requirements only specified what requirements should be met by any game, not this game in particular. Similar, the requirements that come from the TESNATool only specify how the TESNATool should be working.

There are multiple ways to make a game that fulfils all requirements and it is impossible to determine which of these ways is most fitting. The main requirement of a game is to deliver a fun experience. The requirements of such an experience however, differ from player to player. This inherently means that it is impossible to determine what game design is the optimal solution to a problem.

As a result of this, Salen and Zimmerman note that a game designer should not spend a lot of time on designing the game before development [19], but focus on a design by play methodology. The designer should thus spend most of his time on developing the game and optimising it during this design. It should be noted that this methodology fits into the Function-Behaviour-Structure framework, as this framework also suggest optimising requirements and design based on evaluation.

Though this does not necessarily lead to a fun experience for every player, something seemingly impossible, the design by play methodology gives the developer good insights into what does or does not work for the game.

Due to the time limitations of this research, the game design as specified below can thus not be considered fully optimised. Though the game is fully functional, it is still subjected to change in the future. Similarly, some of the design choices mentioned might not have been a part of the original design, while other parts of the original design where removed and are not mentioned due to constraints of this research.

The game requirements as well as Salen and Zimmerman [19] suggest a design of multiple elements of a game. First, the game system itself should be designed. This includes the objects, attributes, internal relationships and environment of the game and is referred to as the System Design. Within the system certain rules should be applied and these are thus designed in the Rules Design. After this, there should be a design on how the player is going to interact according to the rules with the designed system, the Interactivity Design. Since it is a requirement to communicate to the player the effects of their actions as well as what is required of them, a Communication Design should be included. Lastly, the Game flow Design specifies how the game will be played and how the player will be rewarded or punished for his actions.

6.1 System Design
The System Design specifies the game’s objects, attributes, internal relationships and environment. It not only elaborates on the design of these elements, but on their visual representation as well. The system can thus be split up into two sections. On the background there is the simulation environment as specified by Nijmeijer [29]. At the same time, the system also has a visual representation of itself. Objects, attributes, internal relationships and the environment are all added to both systems.

The system or game world should represent a company structure, as this gives players good examples of the problems they might encounter in a real situation and allows for them to feel connected to the situation. A company structure is defined as a set of persons working for the company, a set of software
modules that the company is working on and the relations between these persons and software modules. When considering the requirements of the TESNATool, this suggest that persons and software modules should be the primary objects of the game, while the internal relationships between the objects are equivalent to the relations in the company structure.

Both objects represent something from the real world, and thus the player should be able to identify them as such. Therefore, the person objects should be visually represented as a figure that is reminiscent of a human. Software modules on the other hand are harder to define as they do not represent something physical and can thus be visualised in a more general way.

In the game, each object should have its own name, allowing the game to determine the difference between individual persons and software modules. Other attributes are not necessary however, as the TESNATool considers each person and software module within the system as an equal. This means that no extra visual representation is necessary to show the different attributes, as the player will be able to tell each person and software module apart based only on its location in the environment.

With no other objects in the game than the persons and software modules, all internal relationships are described by the relations that represent the company structure. Within the system, these relations are stored as dependency structure matrices, while visually, simply drawing lines between persons and software modules shows whether there is a dependency between them. To make visualisation easier however, the game will show a different colour when the line represents a social, technical or socio-technical dependency.

Lastly, the environment represents the entire company. All persons, modules and their relationships should be stored within the environment. The environment itself does not need a visual representation other than the representation of its contents, because in the game only one company structure should be evaluated at the same time.

6.2 Rules Design

The rules within the game help define what the player can and cannot do and what he should be rewarded or punished for. The rules should aid the player in learning to detect coordination problems, while at the same time they should encourage the player to keep playing [19].

The rules derived from the TESNATool are all defined by the implemented patterns. That is to say, if one of the patterns detects a coordination problem in the system, the rules require the player to solve this problem. When no coordination problems exist within the system, the rules consider the system safe and the player successful.

In order to solve the coordination problems, the player is allowed to hire new persons, assign a person to a job and ask a person to communicate with another person. A player is not allowed to start a new software module, nor restructure the dependencies between software modules. This is because it should not be allowed to change the technical structure of the company. Changing the technical structure would mean that the company had suddenly made a reorganisation on the software structure, leading to new coordination conflicts and thus delay the project.

When coordination problems consist for a long period in time within a software project, the progress of the software project might become damaged. This could ultimately lead to buggy software, a delay or even a cancelation of the software project. The goal of the game should thus not just be to detect and solve all coordination problems, but solve them before they damage the project too much. The speed at which the software project could become damaged is dependent on the amount of coordination problems present in the system, as well as what coordination problems exist. In the game a coordination problem according to Conway’s Law is considered a bigger problem than a Code Ownership violation.

A player is able to correct any mistakes he might have made during playtime, because making an accidental mistake should not prevent the player from finding all coordination problems. If this would not be allowed, players making accidental mistakes would not have a fun experience. The player are not able to undo any actions the game itself took however, as this would allow them to solve all coordination problems by making the company structure not contain any software modules, meaning there are no coordination problems.

Lastly, there are rules to encourage the player to find the optimal solution. This makes the game more challenging while also encouraging the player to play the game for a second time. To do this, the game shows a score that is raised every time a puzzle is completed. The game then determines whether the player has made more moves than the optimal strategy. If this is the case, the player will be rewarded with a negative score, showing that too many moves where done.

6.3 Interactivity Design

Interactivity is one of the key components of a game, as the player is asked to interact with the system. Interactivity determines what actions the player can perform and how he can perform these actions. As defined in the rules, the player can perform three actions to solve coordination problems, as well as some actions to undo his previous activity.

To ensure a good accessibility to the game interaction is as intuitive as possible. When only a few buttons are used, the player will have to spend less time figuring out how to do something and can instead spend more time on figuring out what to do. Ideally, the player will do things automatically the way he has to do them.

To hire a new person in the game, the player has to press the left mouse button on the screen. The game will then draw a person on the screen at the position of the mouse cursor and add a new person to the simulation environment. To delete a created person, the player has to click the right mouse button when hovering the mouse cursor over the person on screen.

To create dependencies between persons and software modules, the player has to hover over either a person or software module on screen. Holding the left mouse button and dragging the mouse to another person or software module will be sufficient to create a dependency between the two. Again, deleting can be done by clicking the right mouse button, this time when hovering over the newly drawn line.

The interactivity of the game allows the player to detect and solve all coordination problems with just the left mouse button. Only when the player makes an error he wants to correct the use of the right mouse button is necessary. Nevertheless, only a maximum of two buttons are used to perform multiple actions.

Due to all the different dependency lines in the system, the player might be unhappy with the places of persons and software modules on screen. Deleting these and placing them somewhere else, requires too many actions however. Therefore a last action is allowed in the game. When hovering over a person or software module with the mouse cursor, holding the right mouse button allows the player to drag it to somewhere else. This prevents the player to become frustrated with the visual representation of the system or the possible misplacement of a new person, allowing the player to have more fun.
6.4 Communication Design
In the requirements of the game, communication is an important factor. Since the player will be rewarded or punished for his actions and because the player has to learn what he is doing wrong or right, the game needs to communicate the effects of actions to the player. This is especially essential for this game, as the goal is to learn players solve coordination problems on their own.

The most important communication within the game is showing how many problems there are and showing the effect of performed actions on the number of communication problems. Since the rules already defined the usage of health in the game, the game will simply show how much health is going to be lost if the problems are not solved on time. Health that will be lost after a given time, will blink while the rest will stay constantly visible. When an action is performed and the health loss grows or shrinks, so will the amount of blinking health.

The player also needs to know when the coordination problems have consisted long enough to damage the project. For this, a time bar is shown, which decreases in length over time. Once the length reaches zero, the health is decreased, communicating the time limit the player has.

The game also implements rules to encourage the player to find an optimal strategy. This is communicated by showing the current score of the player. Once the player makes too many moves, the score gets decreased with every move, showing the player he did not choose the optimal solution.

6.5 Game Flow Design
Having a system with interaction, rules and communication does not make a game, as there is no meaningful play in the game without a clear goal. Giving the player a goal to reach will give the game meaning, allowing the player to learn and have fun [19, 30].

Since the goal of the game itself is to teach its players to detect and solve coordination problems, the main goal for the player should be to solve coordination problems. The game will use a puzzle structure for this, as this allows the player to encounter many different examples of coordination problems in a company structure and allows the game to become progressively more challenging.

With all elements of the game defined, the general flow of the game can be designed. The game flow considers all user interaction, the rules and the goal of the game to construct a state chart of the game. Figure 3 shows the game flow of the game.

![Figure 3. Game Flow Design](image)

The game flow shows that the game starts by loading a puzzle. Once the puzzle is loaded, a time bar will start running. If the player does not perform any actions, eventually the time bar will run out and subtracts the health loss from the health of the puzzle. If the puzzle runs out of health, the player has failed and the puzzle will be reloaded.

If an action is performed, the score will be updated if the player used more moves than the optimal solution. The action is then evaluated by the rules, to determine if the health loss will be changed. If this is the case, the health loss will be changed. If there is no more health loss, the puzzle is considered solved and a new puzzle is loaded. In any other case, the game will wait for the next action.

7. SOFTWARE DESIGN
The second process of software design according to the Function-Behaviour-Structure framework is the process of synthesis [18]. In this phase the expected behaviour, as described by the game design, is transformed into the actual solution.

According to Lethbridge and Laganière the implementation of a system should be done by following a class diagram that shows the general structure of the system [32]. Using a programming language, the system should then be fully implemented.

Since the game will implement the TESNATool simulator developed by Nijmeijer [29], the structure of this simulator should first be discussed, so that the class diagram for the game can be constructed. The game is implemented with the programming language JavaFX Script, which should be discussed, as it is an uncommon and fairly new programming language.

7.1 TESNATool Simulator Classes
The TESNATool Simulation Environment consists of a set of Person and Software Module classes, the Social, Technical and Socio-Technical matrices and methods for calculating coordination problems by using the different patterns. The majority of the Simulation Environment was developed by Nijmeijer [29], though it was slightly changed to have some extra functionality. Figure 4 shows the class diagram of the Simulation Environment. In the figure, operations have been hidden due to their length.

![Figure 4. Class Diagram of the Simulation Environment](image)

7.2 Class Diagram
Following the class specification of the simulation environment, a specification on the user interface, the visual part of the game, should be specified. This includes the specified system and all of its specified elements, the interaction and the communication, as well as an implementation of the game flow. The class diagram showing all classes with its operations can be seen in figure 5.
The different classes can be split up into four different groups.

The TESNAGame class controls the entire game, including the TESNAGenerator, PuzzleManager, health, time limit and score. The TESNAGenerator is responsible for constructing and managing the set of persons, software modules and their relations, which are described by TESNANodes and TESNALines. The TESNANode, which can be either of the type Person or Module, is the visual representation of a node in the game, while the TESNALine is the visual representation of a link between two nodes. Lastly, the Puzzle Manager manages the loading, reloading and saving of puzzles.

7.3 JavaFX Script
As described by Nijmeijer, the Simulation Environment is specified in Java [29], making the implementation of the game in the same language a seemingly logical choice. However, anyone experienced with programming in Java will realise that it is hard to make programs that use heavy graphics and interaction in Java.

According to Weaver UI design is relatively easy in the programming language JavaFX Script, due to the use of triggers, functions that automatically execute when a variable is changed [33]. This allows the designer to spend less time on making code that shows something on screen and gives more time to develop what should be shown on screen. Since JavaFX can import any code programmed in Java [33] the language is an ideal choice for the development of the game.

7.4 TESNAGame Class
The TESNAGame class is responsible for the managing of the entire game. It’s variable array content holds all the visual data for the game, and health, time and score are measured in this class. The class can be split up into two sections.

The first section holds the playing field, on which TESNANodes can be created by clicking on it. For the actual creation and management of the TESNANodes, the TESNAGame class refers to the TESNAGenerator. Whenever a Node or Line is created, the function update() is called. When update is called, the game checks whether there is still health left or if the health loss has reached zero, loading the next puzzle.

The second section holds the sidebar of the game, which shows the score, healthbar and timebar. On the healthbar a blinking part shows the health that is to be lost after the timebar, which depletes over time, reaches zero. Figure 6 shows the entire game as it appears on screen.

7.5 TESNAGenerator Class
The TESNAGenerator class is responsible for holding all the information on all TESNANodes and TESNALines. When the player clicks on the playing field in TESNAGame, the function createNode() is called on the location of the mouse, adding a new TESNANode object on that location. For lines a variable is hold so that the player can see the line that is currently drawn. When the mouse is released, the generator checks if the drawn line was drawn between two TESNANodes and if so adds them to the proper dependency matrix in the Simulation Environment.

7.6 TESNANodes and TESNALines
Since both persons and software modules would hold much of the same functionality in the game, the class TESNANode was created. The TESNANode class draws a circle on a colour on screen and makes sure this circle can respond to user interaction properly. This includes moving the node, deleting it and starting to draw a line. The subclasses extend the visual representation of the node and hold an instance of the Person or SoftwareModule class they represent in the SimulationEnvironment. The visuals of Person are extended to make is look more like a human being, as can be seen in figure 6.

The TESNALine class holds the visual representation of a dependency between two TESNANodes. Depending of the type of dependency the line is representing, the colour is either Red (Social Dependency), Blue (Technical Dependency) or Green (Socio-Technical Dependency). The TESNALine class also contains the functionality to delete the line.

7.7 Puzzle Manager Classes
In the game, there are three classes responsible for managing and constructing puzzles. A puzzle is formatted as follows:

1. A line with the initial health of the system.
2. The number of moves the optimal solution requires.
3. Multiple lines, with the description of every node in the puzzle.
4. Multiple lines, with the description of each dependency in the system.

The PuzzleManager class only determines whether a puzzle should be loaded or saved. If a puzzle is loaded, the system searches for a file named puzzlex.tgp, where the x is the number of the expected puzzle. Once the file is found, the puzzle description is formatted and inserted into the TESNAGame. The PuzzleSaver class, which is only used for constructing puzzles and is thus never called by the player, constructs a new
puzzles.tgp file out of the situation as represented in the TESNAGame class.

8. EVALUATION
The forth process of the Function-Behaviour-Structure framework is that of evaluation. Evaluation is a necessity for the designed game, as its teaching effects should be properly tested. For this, multiple students and faculty members were asked to play the game.

Because it should also be tested whether the game has a positive effect on decision making abilities involving coordination problems, just testing how the game gets played will not be enough, as improving skills in a game does not necessarily mean the user has learned the underlying problem the game is teaching.

In order to test the teaching abilities properly, the game was extended to include a questionnaire with two questions, one for each coordination problem, that would be given to players at both the beginning and end of the game. The first question focussed on the code ownership pattern, while the second question gave an example of the Conway’s law pattern. Neither of the patterns where explained and both of the questions only gave the player a single case with four possible solutions each. To prevent any direct link with the game, the cases where represented in a textual form.

After the first questionnaire, the players could choose a short tutorial, that would explain the controls of the game. However, neither Conway’s law nor the code ownership pattern where explained, since an explanation of these patterns would help the players determine the correct answer, without having them play the game.

A total of 24 people were asked to answer the questionnaire and play the game. Their answers and a log containing their actions in the game, where stored and analysed. Table 1 shows the answers given by the respondents to both questions the first and second time (correct answers are marked in grey), the amount of questions they answered correctly the second time, the number of answers they improved, the number of answers they worsened and the total difference in answers to the questions between the first and second attempt.

Table 1. Answers to questions by players

<table>
<thead>
<tr>
<th>Player</th>
<th>Question 1</th>
<th>Question 2</th>
<th>Cor</th>
<th>Ch</th>
<th>Imp</th>
<th>Wor</th>
<th>Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1st</td>
<td>2nd</td>
<td>1st</td>
<td>2nd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>D</td>
<td>C</td>
<td>C</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>D</td>
<td>C</td>
<td>C</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>C</td>
<td>C</td>
<td>D</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>A</td>
<td>D</td>
<td>B</td>
<td>B</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>C</td>
<td>C</td>
<td>B</td>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>A</td>
<td>D</td>
<td>B</td>
<td>C</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>D</td>
<td>D</td>
<td>C</td>
<td>C</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>C</td>
<td>D</td>
<td>B</td>
<td>C</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>C</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

On average, users answered 1.17 questions correctly the second time, while only 0.96 questions were answered correctly the first time. However, table 1 clearly shows not all players showed an improvement and for some the number of correct answers even declined.

Based on analysis of the play sessions that were logged as well as the response gathered from some of the users after they had played, multiple reasons can be found as to why the game showed no significant improvements.

Analysis of play sessions shows that users would regularly solve puzzles by drawing as many lines as possible between the different nodes. This would eventually lead to the puzzle being solved, even though some users stated they had no idea why the puzzle was actually correct. As a reason for this users stated that the game did not explain what the coordination problems actually where. Thus, users where unable to actually detect the problems.

Another reason given by the users was the time pressure given by the game. Users felt the time bar decreased too quickly, forcing them to perform actions as quick as possible, without them evaluating the effects of these actions. This would again lead to puzzles being completed without the users knowing why they were correct.

A final reason the users gave was the formulation of the questions that where asked. Some stated that the questions where “vague”, making the correct answer unclear to them.

9. CONCLUSIONS
With the game developed and tested, the research question can now be answered. The question was:

How can we create a serious game based on the TESNA DSS so that players will improve their ability to detect and solve coordination problems in software development?

The research showed that to create a serious game, one should follow the design rules given by Salen and Zimmerman [19]. This includes designing a system, rules, interactivity, communication and game flow. In order to teach the players of the game to detect coordination problems, it was decided to make detecting these problems a central piece in the rules and game flow.

When looking at the evaluation results, it cannot be concluded that the game successfully taught its players to detect coordination problems, as the results only show a minor improvement. However, it can’t be concluded that the game did not work either, as there was an improvement nevertheless.

10. FUTURE RESEARCH
Though the conclusion showed the game does not fully improve the player’s ability to detect coordination problems, user response gave some final remarks that will be of assistance in further research to improve the game.

First, the game should be improved to give a better explanation of the communication problems themselves. Currently,
users claimed the goal of the game was unclear. Instead of letting the players figure out what the different coordination problems are, the game should tell the players beforehand and thus puzzles should be used as an explanation of the rules. Secondly, testing has to be done on the time limit of the puzzles. Player response suggested the time limit was too strict and pressed the player too much to do an action. Adjustment on the time limit could potentially improve user response.

Third, the form of evaluation should be done in another way, because once the game explains the coordination problems, it is impossible to determine whether questions are answered correctly based on the tutorial of the game or on playing the game. A better evaluation could be done by interviewing the users on the knowledge they gained from playing the game, instead of a questionnaire.

ACKNOWLEDGEMENTS
I would like to thank the following people for their support during the months spend on the research that resulted in this paper: Chintan Amrit, for giving me this interesting research and his continuing support on making the game great, Maikel Nijmeijer for his work on the TESNA Simulation Environment, Erik Dekker for helping me bugfix the game and of course my partner Kirsten Maartense for her support through thick and thin.

REFERENCES