Towards a Simulation Environment for Software Managers

For testing managerial response to coordination problems

M. Nijmeijer (Maikel)
m.nijmeijer@student.utwente.nl

ABSTRACT
The goal of this paper is to create a simulation environment in the shape of an abstract game that can be used to test managerial response to coordination problems. Literature review shows that current tools and methods are insufficient for reaching the goal of today’s software engineering. While a simulation environment has proven to be valuable in other scientific fields, such an environment has not been considered in much detail in the field of software engineering. The contribution we make in this paper is such a simulation game in the context of a software engineering environment.

Keywords
Design science, dependency structure matrix, model-view-controller, observer/observable, JUNG, simulation environment, abstract game

1. INTRODUCTION
According to [1], software engineering is the process of solving customers’ problems by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints. This form of software engineering is nowadays getting increasingly complex. [2] To mention a couple of reasons for this:

- Software systems have to be dynamic because they constantly have to adapt to a changing environment with its own requirements. [3]
- Software systems are often very large. This means that there are a lot of states in which the actual program can be. [2]
- Software systems have to deal with the kind of hardware on which it will be used. [4]
- A lot of software engineering projects run over time or budget. [4]

Failure of software engineering causes a lot of trouble. According to a report from the Standish Group in 1995: 31.1% of US software projects were cancelled and 52.7% were completed over allocated time, over budget, or lacked certain functionality. The cost of failed projects in the US in 1995 was $81 billion, while projects that overrun their budgets were good for another $59 billion. [5]

It turns out that there are four categories of software or information system projects failure [6]:

- Correspondence failure: the requirements are not met.
- Process failure: projects run over time or budget.
- Interaction failure: problems related to the use of the system.
- Expectation failure: stakeholders’ expectations cannot be met.

Since the beginning of software engineering there have been many solutions proposed to these failures. Some of these solutions have been successful while others not. Examples of the solutions are the use of high-level languages to write code, unified programming environments, time-sharing or programming techniques like object-oriented programming with the ideas of composition, abstraction and frequently used software patterns. [1], [2]

Even with the available techniques it’s a hard job to make a piece of large, high-quality software. After all, the increased complexity of software engineering has also been influenced by management problems. An example of such a management problem is dealing with coordination problems. These are vital to the success of a project, but cannot be solved through technical solutions. [7]

Therefore, the focus of this research does not lie on the technical problems, but on the management problems. Large software systems get split up into smaller parts by the use of decomposition and different small teams work on different software modules. [8] The management-related questions that arise in such a situation, include:

(i) How is it possible for a software manager to have a solid overview on the overall project?
(ii) How can the coordination among team members or different software teams be arranged?

These are the main questions that play a role in this research.

2. MOTIVATION
This paper suggests the use of a simulation environment for solving the coordination problems that arise in a software project. An in-depth literature review was conducted on the subject of coordination problems in software engineering. This helps in understanding the nature and the short coming of the current solutions. The literature review was conducted according to [9]. The databases considered included ACM Digital Library, Ingenta, Web of Science, and Google Scholar.

Most of the publications found, deal with the existence of communication among members of the software team. See [10] for a discussion of this. This communication can occur face-to-face, via e-mail, via chat, etc. However, there are differences in channel enrichment. Channel enrichment means the amount of information that can be interchanged in a certain time. Talking
to someone in person is generally richer in channel and therefore more useful for coordination. [11]

[12] comes with another solution to coordination problems for software managers. They say that responsibilities among team members should be made clearer and that people should do things in which they are good at.

Sometimes it happens that two different team members have to use the same resources at the same time. Those ‘producer-consumer dependencies’ cause a lot of coordination problems. Therefore, a dependency analysis can be a useful tool to prevent coordination problems. The dependency analysis should be done before the start of the actual project. [13]

Other kind of solutions for coordination problems deal with the tools that are used to develop software. Many of these tools are integrated development environments, or IDE’s. Such tools create an environment for the programming language used and provide several facilities for software engineering. Examples of these facilities are source code editors, compilers, debuggers, etc. [14] At present, lots of plug-ins are built for IDE’s. Some of these plug-ins deal with the coordination among members of a software team. A software manager can use these plug-ins to aid in making coordination decisions. For example, files can be shared, different schedules can be made, or more people can access the same file at the same time. [15] is an example of such a plug-in.

All the solutions mentioned, deal with the prevention of coordination problems. This paper takes another perspective on the problem of coordinating software development. It deals with the visibility of coordination problems for software managers in software projects. In other words, we try and simulate an environment where a project manager can make a decision on the Socio-Technical coordination of the project without affecting the actual software development process. This approach aids the software managers in learning about such Socio-Technical coordination problems. [16] Thus the manager need not learn through trial-and-error, that can be very costly. Wrong decisions can lead to complete failure of the project, or could lead the project in a wrong direction. A simulation environment can help a software manager in making coordination decisions in a more controlled way. [17]

3. RESEARCH SUBJECT
3.1 Research Questions

The subject we wanted to research is the suggested simulation environment. The use of such a simulation game can be valuable for making coordination decisions in the discipline of software engineering. [18],[19]. The main question that will be answered in this paper is:

How can we create a simulation environment to test managerial response to coordination problems?

Such a solution is a decision support system. [20],[21] We develop such a system using the design science methodology.

3.2 Design Science

Design science is a very useful and common way to do research in the discipline of artificial intelligence. Because of the nature of this research, the creation of an artifact, design science can also be useful in this project [22].

A way of performing design science in the discipline of information systems is proposed by Hevner et al. [23] They state seven guidelines for using design science in this context. The scope of this research is actually limited, so not all the guidelines are used. However, the following are considered during this research:

- **Design as an Artifact**: the use of constructs, models, methods, or instantiations. This paper is about constructing a simulation environment, which is an artifact by itself.
- **Design Evaluation**: the designed artifact must be rigorously evaluated. The evaluation of the artifact is partly done in this paper, by comparing the simulation environment to the real world.
- **Communication of Research**: the outcomes of the research must be effectively communicated. This communication happens in the shape of this paper, that mentions the results of the research.

3.3 Designing Artifacts

The Function-Behaviour-Structure framework is used for developing the artifact. [24],[25] This framework comes with the idea of separating three classes of variables, describing different aspects of the artifact:

- **Function (F) variables**: describes where the artifact is for.
- **Behavior (B) variables**: describes what the artifact does.
- **Structure (S) variables**: describes the components and their relationships of the artifact.

The Function-Behaviour-Structure represents eight processes that are claimed to be fundamental for designing artifacts. Figure 1 shows these eight processes.

![Figure 1. The Function-Behaviour-Structure framework based on [25].](image)

Again, due to the scope of the project, not all the processes are considered. We mention two processes in this research:

- **Process 1 (formulation)**: transforms requirements into the solution structure. This happens before the actual implementation of the simulation environment.
- **Process 2 (synthesis)**: transforms the expected behavior into the solution structure. This is the actual implementation phase.

4. DEFINITIONS

This section introduces some definitions that are frequently used in the paper. The goal of these definitions is a better understanding of the whole research.
4.1 Dependency Structure Matrix
A dependency structure matrix is a method for representing task interdependencies within a business process. [26] It can provide information about which tasks depend on which. Dependencies under the gray diagonal represent ‘feed forward information’ and dependencies above the gray diagonal represent feedback. Figure 2 shows an example of a dependency structure matrix.

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

The following information is represented in figure 2:
- Process 2 gives feedback to process 1.
- Process 3 gives feedback to process 2.
- Process 3 gives information to process 2.
- Process 4 gives information to process 1.

There is still another thing that figure 2 represents. Process 2 and 3 give information to each other; process 2 gives information to process 3 and process 3 gives information to process 2. This means that process 2 and process 3 depend on each other.

A dependency structure matrix can also be used in the field of software engineering. This is possible because of the modularity of large software systems. These large systems are split up into many different working modules that can depend on each other. Therefore, a technique like dependency structure matrices can be useful for this purpose.

4.2 Model-View-Controller Pattern
The model-view-controller pattern is an architectural pattern used in software engineering. The pattern takes care of the separation of different components of software: the model, the view, and the controller. [27]

The model is the actual business model. It represents the information and data structures behind the program (such as persons, software modules, etc.). The view is the user interface. This can be a graphical user interface or a textual user interface. We are using a graphical user interface in this research, because it renders the model into a suitable form of interaction.

The controller processes and responds to events. Events are actions from the user, like pressing a button, pressing a mouse button, etc. The controller determines what happens when an event occurs.

See figure 3 for a class diagram that shows the model-view-controller pattern.

![Figure 3. Class diagram that shows the model-view-controller pattern.](image)

4.3 Observer and Observable
If the model-view-controller pattern is used, the use of observers and observables can make life easier. An observer is an object that watches an observable and an observable is an object that is watched by an observer. In terms of the model-view-controller pattern: the view is the observer-object and the model is the observable-object. To be more explicit, the view is an observer of the observable model. [28] This can be made clear with the following example.

The model-view-controller pattern can be used for creating a chess game. The model has data structures to represent all the pieces, the board and the actual chess game. The view is just a graphical user interface for the board and the pieces on it. The controller is assigned to handle events. It is possible that a human user moves the white horse from position B1 to position C3 with the keyboard or with the mouse. By pressing the keyboard or clicking the mouse button, an event occurs. The handler of events, the controller, will notice this event and will determine that the human user would like to move the white horse from position B1 to position C3. If this is a legal move, the controller will update the accessible model. The model updates the available data structures and remembers the new position of the white horse. The view, that is an observer, continuously watches the observable model. Every change in the model is notified by the view, also the horse movement. The view is then ready to update the user interface, by drawing the horse in position C3 instead of position B1.

See figure 4 for an updated class diagram that shows the model-view-controller pattern with the observer/observable pattern included.

![Figure 4. Updated class diagram that shows the model-view-controller pattern.](image)

The added dashed lines represents the observer/observable pattern. The model is observable to the view and the view gives the events to the controller.

4.4 Java Universal Network/Graph
Java Universal Network/Graph, or JUNG, is a software library that provides a common and extensible language for modeling, analysis, and visualization of data that can be represent as a graph [JUN08]. JUNG is able to transform a dependency structure matrix into a graph. This graph can be either directional or undirectional. This depends on the type of dependency structure matrix used.

Another advantage of JUNG is the build-in of certain standard graph algorithms from graph theory or social network analysis. This is helpful because it reduces the number of code that has to be written for the simulation environment. Also, JUNG is tested well. Therefore, the used algorithms are supposed to be
working. This means that the actual testing of the simulation environment can also be limited. [1]

See figure 5 for an example of a graph with three nodes, V6, V7 and V8, and unidirectional links between them. This example is created with an application from the JUNG library. [29]

![Figure 5. Unidirectional graph created with JUNG.](image)

### 5. REQUIREMENTS FORMULATION

The first stage of the actual software development process is phrasing the requirements. This is also the first process in the Function-Behaviour-Structure framework, suggested by [25]. The formulation of requirements will define a scope in which the total project must fit. The requirements phase is even more essential in a time-limited research like this. [3]

Another advantage is the goal-directed approach that can be used. Clear requirements help in finding the direction of the research and the actual implementation of the software. [3]

C. Amrit and M. Nijmeijer have formulated the following concrete requirements for the simulation environment that is subject to research:

- The simulation environment contains persons and software modules.
- Persons can be connected to each other. This means that they can communicate. Such a connection is called a communication line.
- Software modules can be connected to each other. This means that they depend on each other. Such a connection is called a dependency.
- Persons can be connected with software modules. This means that these persons are working on the corresponding software modules. Such a connection is called a task connection.

All the persons, software modules, communication lines, dependencies and task connections together on a single moment in time, say x, is called the configuration of the software project at x.

Besides the elements for the configuration of the software project, the simulation environment has also requirements for the actual simulation part.

- The simulation environment contains a simulation component. This means that the configuration of the software project can be simulated for a week, say week x+1.
- The simulation environment contains a ‘health’ property. This property indicates the probability of the software project to be successful.

The success of a simulated software project depends on the configuration of the software project in each week. This can be expressed in the following equation:

\[ S(P) = \prod_{i=0}^{x} H(x) \]

**Equation 1. Function to calculate the probability of a software project to be successful.**

In words, the probability of a software project (P) to be successful after x weeks (S), is the product of the probabilities of a software project to be successful from week 0 to week x. If a software project runs for four weeks, and the probability of the software project to be successful in weeks zero (the start of the project), one, two and three are 98%, 90%, 85% and 20% respectively, then the probability of the total software project to be successful can be calculated as follows:

\[ S(P) = 0.98 \cdot 0.90 \cdot 0.85 \cdot 0.20 \approx 0.15 \]

This means that the probability of the software project to be successful is 15%. The function to calculate this only makes sense in a simulated software project. In real life, the probability of a software project to be successful in week x does not depend on the probability of a software project to be successful in week x-1.

The question that arises now: how can the simulation environment determine the health of the software project at a certain time, again say x? This should be made explicit in requirements.

- The simulation environment contains so-called constraints. These constraints determine the health of a software project at a certain time.
- The health of the software project can change every next week; it’s not possible to simulate backwards, e.g. from week 3 to week 2.

The simulation environment can be loaded with several constraints. The probability of a software project to be successful at week 0 is 100%. After a simulation, the constraints look at the configuration of the software project. The health of the project will decrease if the configuration of the software project does not satisfy the constraints. The amount of decrease is subject to research. We have to choose a real number that represents reality. Equation 2 gives an example of a one week simulation of a software project. As can be seen, the constraints are not satisfied in the first week and the amount of decrease is 40%.

\[ S(P) = \prod_{i=0}^{x} H(x) = 1 \cdot 0.60 = 0.60 \]

**Equation 2. Calculation of the example mentioned**

Finally, there are requirements dealing with the visual part of the simulation environment. These requirements are quite obvious: an abstract game is not really a game without a graphical user interface.

- Persons and software modules can be drawn on the screen or removed from the screen. This can be done with simple mouse clicks that will create events.
- Communication lines, dependencies and task connections can be drawn on the screen or removed from the screen. This can be done with simple mouse clicks that will create events. All these links are directional.
- The health of the project and the simulation button are constantly visible on the screen. Pressing the simulation button will create an event that updates the current health of the project.

### 6. SYNTHESIS

The stage of synthesis is the second stage in the Function-Behaviour-Structure framework that is used. This stage involves the actual implementation of the simulation environment.
During this stage, requirements were transferred into a working program.

To make things clear, a running example is used in this section. This running example aids the reader in understanding the decisions that are made during the implementation.

### 6.1 The Overall Simulation Environment

The implementation of the simulation environment is done in a programming language called Java. This programming language has the advantage of being a high-level programming language. This means that the programmer is not confronted with hardware programming. Furthermore, there are many libraries available with functions that are already programmed. These libraries can be used by importing it in the program. An example of such a library is already mentioned: the JUNG library.

See figure 7 for a class diagram, showing the simulation environment.

![Figure 7. Class diagram that shows the simulation environment.](image)

In this figure, the class SimulationEnvironment is the model and therefore observable. The class SimulationEnvironmentGUI is the view and therefore observer. The controller is encapsulated in the SimulationEnvironmentGUI. This way, the controller has access to the data structures of the view, because of the encapsulation. Besides, the controller has access to the data structures of the model because of the connection between model and view.

The model, SimulationEnvironment, is connected to the following classes:

- **SoftwareCallGraph**: this class is responsible for maintaining the dependencies between software modules.
- **SocialNetwork**: this class is responsible for maintaining the communication lines between persons.
- **PersonSoftwareModule**: this class is responsible for maintaining the task connections between persons and software modules.
- **Person**: this class is responsible for maintaining the name of the person.
- **SoftwareModule**: this class is responsible for maintaining the name of the software module.

Besides the SoftwareCallGraph, SocialNetwork and PersonSoftwareModule, there are two more properties for the model. First, the health of the project, that indicates the probability of the software program to be successful. Second, the number of weeks that are simulated. These two components are necessary for the simulation component that is formulated in the requirements.

### 6.2 Modeling the Simulation Environment

Suppose there is a software team of four members, working on a program. These persons are called ‘A’, ‘B’, ‘C’, and ‘D’. Because of the complexity of the program it is split up into five different software modules. We number the software modules ‘1’, ‘2’, ‘3’, ‘4’, and ‘5’.

According to the first stage, the formulation of requirements, it is the responsibility of the model of the simulation environment to remember the persons, software modules, communication lines, dependencies and task connections. The four persons and the five software modules from the example are saved in an array. This is a data structure used in JAVA to save many objects from the same type. In the running example, this can be represented as:


The communication lines are modeled as a dependency structure matrix, that is mentioned earlier. We assume the following facts for the running example:

- Person ‘A’ and person ‘C’ can communicate with each other.
- Person ‘B’ and person ‘C’ can communicate with each other.
- Person ‘B’ and person ‘D’ can communicate with each other.

Figure 8 shows how the assumptions are represented as communication lines in the class SocialNetwork.

![Figure 8. Example of a social network.](image)

The dependencies between software modules can also be modeled as a dependency structure matrix. Again, we assume some facts for the running example:

- Module ‘3’ depends on module ‘1’.
- Module ‘4’ depends on module ‘2’.
- Module ‘1’ depends on module ‘4’.
- Module ‘3’ depends on module ‘5’.

Figure 9 shows how the assumptions about software modules are represented as dependencies in the class SoftwareCallGraph.

![Figure 9. Example of a software call graph.](image)

Finally, there are task connections. These are also modeled as a dependency structure matrix. We make some final assumptions for the task connections:

- Person ‘A’ works on module ‘1’.
The predicates used, have the following meaning:

- **S(x)**: x is a software module.
- **P(y)**: y is a person.
- **SP(x,y)**: software module x is occupied by person y.

The algorithm uses two ‘for-loops’. The first iterates over the different software modules and the second checks for every person whether he, or she, is working on that software module. There will be a counter initialized for every software module. The counter will be incremented when a person is not working on the software module in question. Finally, the counter will be checked when all the persons are considered. If the counter is at the same count as the number of persons, no one is working on the software module in question and the method returns a negative result. Because of the double for-loop used, the algorithm lies in the big-oh of \( n^2 \), where \( n \) stands for the number of software modules or the number of persons. [31]

Let’s see how this works for the running example. See figure 10 for a recall of the situation. At first, the algorithm determines that there are four persons in the simulation environment. It begins considering software module ‘1’ for all the four persons. It can be seen very soon that person ‘A’ is working on software module ‘1’ and the other three persons are not. Thus, the counter increments three times. This means a count of three. Three is less than four, so there is at least one person who is working on software module ‘1’. So far nothing to care about. Also for software module ‘2’ and software module ‘3’, nothing strange happens.

This is different for software module ‘4’. Again, the algorithm determines that there are four persons in the simulation environment. Software module ‘4’ is considered for all the four persons. After four iterations, it is noticed that no one is working on this software module. This means that the counter is incremented four times and that the count is equal to the number of persons that are working. The algorithm breaks immediately because of the fact that there is a software module that is not occupied.

### 6.3.2 Communication Lines & Dependent Software

The second constraint that is considered, is more difficult:

*There has to be a communication line between two persons who are working on software modules that depend on each other.*

Again, the meaning of this constraint is obvious. There has to be communication between persons who are working on dependent software modules. Otherwise, the quality of connections between the dependent software modules cannot be guaranteed. Again, let’s transform the constraint to first-order logic to see what the statements means in formal terms.

\[
\forall w \forall x \bullet S(w) \land S(x) \land \text{DEP}(w,x) \rightarrow \\
(\exists y \forall z \bullet P(y) \land P(z) \land SP(w,y) \land SP(x,z) \rightarrow \text{COM}(y,z))
\]

The predicates used, have the following meaning:

- **S(w)**: w is a software module.
- **DEP(w,x)**: software modules w and x are dependent.
- **P(y)**: y is a person.
- **SP(w,y)**: software module w is occupied by person y.
- **COM(y,z)**: person y and z are communicating.

The algorithm that can be used, comes from [16],[32]. It proposes the multiplying of the matrix in the PersonSoftwareModule class with the matrix in the SoftwareCallGraph class. This means multiplying the dependencies with the task connections. How this is done, can be seen in [33]. Appendix B shows how the algorithm is implemented.

Appendix C shows the way in which algorithms for constraints can be included in the program. It shows how to watch at the
constraints while simulating a week of the software project. More constraints can be loaded into the program by checking other algorithms on the place of the dots. Note that the algorithms that are used to represent constraints are static methods. This means that the constraints can always be represented and are not dependent from any state of the program. The only precondition is the delivering of a suitable array with working connections.

6.4 Exceptions
Of course there are things that can go wrong while running the simulation environment. These are called ‘exceptions at run-time and throw an exception. This means that a class signals a condition that prevents the program from continuing normally. [28] The following events will throw such an exception:

- A person or software module that already exists is added to the simulation environment.
- A person or software module that does not exist is deleted from the simulation environment.
- A communication line is made between two persons, while one of these persons does not exist.
- A dependency is made between two software modules, while one of these software modules does not exist.
- A task connection is made between a person and a software module, while the specified person or software module does not exist.

For example, if one of the persons in the simulation environment is named ‘A’ and the human user tries to add a person called ‘A’, the program will throw an exception. This exception takes care of the mistake that is made and returns a decent message that mentions the fault.

For clarity, the mechanism for throwing exceptions is split up into two parts. The first part is a PersonException and the second part is a SoftwareModuleException. This separation makes things clearer and makes the program code more readable.

7. ENVIRONMENT VALIDATION
The phase of implementation is followed logically by the phase of testing and validating. This is also one of the guidelines that [23] proposes. Note that testing the program is not the same as validating the program. Testing is checking whether there are some errors in the program code. These errors come from the implementation phase. The process of validating checks whether the program corresponds with reality or not. [3] Validating deals also with the following question: did we make what we wanted to make?

In this research, testing occurred by using test classes. This means that there are certain classes in the program, or parts of the program, that are used for testing the whole program. The following tests are included in the research:

- Transforming matrices to graphs.
- Configuring the model.

The last test means the adding of components to the simulation environment, and the removing of components from the simulation environment.

The process of validating is more tricky for the simulation environment, because it involves a lot of experiments and research. This paper deals only with the first step of validating. This first step involves the actual using of the program and seeing whether the program behaves according to the requirements. Other steps for the process of validating are mentioned in the section about future work.

8. CONCLUSION & FUTURE WORK
This section contains some final remarks about the research that is done. Also, it comes with suggestions to improve the simulation environment in the future.

First, the current solutions for coordination problems in software engineering are insufficient. Software managers have to learn by trial-and-error and there is no possibility to foresee consequences of decisions related to coordination problems.

Second, a simulation environment aids software managers in learning about Socio-Technical coordination problems. Such a tool can simulate decisions that are made by software managers to see the consequences of these decisions.

Third, the design of a simulation environment includes components like persons, software modules, communication lines, dependencies and task connections. Besides, the simulation environment contains a ‘health’ property that indicates the probability of a software project to be successful. A simulation button can be used to simulate a software project for one week. After that, we can see whether the configuration of the software project satisfies the constraints that are loaded and how this affects the health of the project.

Fourth, the simulation environment that is considered in this paper, is partly validated. This process is done by using the program and seeing whether the program behaves like the requirements that are formulated. It turned out that the program behaves according to the requirements.

Fifth, there are many things that can be done to improve the simulation environment. The simulation environment is built from scratch in limited time. This raises the opportunity for improvement. Below, we suggest some of these improvements:

- Extending the process of validation. There are many experiments that can be conducted to test whether the simulation environment can be compared with real life software engineering. An example of such an experiment, is a session with real software managers. These software managers can play with the simulation environment and determine whether the environment is useful or not.

- There are a lot of constraints that can be added to the simulation environment. A field study can be done to determine the constraints that are useful. Adding more constraints leads to a better representation of real life software engineering. An example of a constraint that can be added to the simulation environment is checking whether there is a deadlock in the dependency structure of the software modules.

- A suitable graphical user interface can be added. The user interface is independent from the model thanks to the model-view-controller pattern. This makes it easy to replace the user interface without affecting the model. The shape of a new graphical user interface can be suggested by actual software managers. This makes the tool even more useful.

ACKNOWLEDGMENTS
I would like to thank the following people who play a significant role in making this research possible: Chintan Amrit, my supervisor in this research. Corrie Hujs and Herman Koppelman, coordinators of the Bachelor Referaat course. Fons Wijnhoven, coach of the Information Management group, and
REFERENCES


[22] Cross, N. Science and Design Methodology: A review, Research in Engineering Design vol. 5 issue 2, 1993, p 63-69


APPENDIX A: WORKING ON ALL SOFTWARE MODULES CONSTRAINT

```java
public static boolean validConstraint(personSoftwareModule) {
    int numberOfPersons = personSoftwareModule.length;
    int counter;

    for(int i = 0; i < personSoftwareModule[0].length; i++) {
        counter = 0;
        for(int j = 0; j < personSoftwareModule.length; j++) {
            if(personSoftwareModule[j][i] == 0) {
                counter++;
            }
        }
        if(numberOfPersons == counter) {
            return false;
        }
    }
    return true;
}
```

APPENDIX B: COMMUNICATION LINES AND DEPENDENCIES CONSTRAINT

```java
public static boolean validConstraint (taskConnections, dependencies, socialNetwork) {
    for(int i = 0; i < dependencies.length; i++) {
        for(int j = i; j < dependencies[i].length; j++) {
            if(dependencies[i][j] == 1) {
                List<Integer> firstSoftwareModule = new ArrayList<Integer>();
                for(int k = 0; k < taskConnections.length; k++)
                    if(taskConnections[k][i] == 1)
                        firstSoftwareModule.add(k);

                List<Integer> secondSoftwareModule = new ArrayList<Integer>();
                for(int k = 0; k < taskConnections.length; k++)
                    if(taskConnections[k][j] == 1)
                        secondSoftwareModule.add(k);

                if(!getCommunication(first, second, socialNetwork))
                    return false;
            }
        }
    }
    return true;
}
```

APPENDIX C: SIMULATING THE SOFTWARE PROJECT

```java
public void simulate() {
    if(!WorkingOnAllSoftwareModulesConstraint.getWorkingOnAllSoftwareModules(taskConn)) {
        this.projectHealth = projectHealth * 0.8;
    }
    if(!ComLinesAndDepSoftwareConstraint.getComLinesAndDepSoftware(taskConn, dep, socNetwork)) {
        this.projectHealth = projectHealth * 0.6;
    }
    this.weeks++;
}
```