Opportunities and challenges when introducing interdisciplinarity issues in systems engineering courses: a case study with a constrained project time frame

Preprint · May 2019

DOI: 10.13140/RG.2.2.29133.90081

3 authors:

Ricardo M. Czekster
56 PUBLICATIONS 136 CITATIONS
SEE PROFILE

Thais Webber
University of St Andrews
69 PUBLICATIONS 255 CITATIONS
SEE PROFILE

César A. M. Marcon
Pontifícia Universidade Católica do Rio Grande do Sul
98 PUBLICATIONS 623 CITATIONS
SEE PROFILE

Opportunities and challenges when introducing interdisciplinarity issues in systems engineering courses: a case study with a constrained project time frame

Ricardo M. Czekster[a], Thais Webber[a], César Marcon[b]
[a]*Performanceware Technologies*, Rua Dr Flores 262/55, Porto Alegre, RS, Brazil
[b]PUCRS – *Pontifícia Universidade Católica do Rio Grande do Sul*, Av. Ipiranga, 6681, CEP 90619-900, Porto Alegre, RS, Brazil

Abstract
We propose an educational framework to combine different disciplines such as Software Engineering, Operating Systems, and Simulation to teach project management, modular programming, and system decomposition to students. We present guidelines for the framework implementation and insights as to how plan simulations of virtualized operating systems.

Keywords: *interdisciplinarity, teaching software engineering, simulation, operating systems.*

1. Introduction

Many Computer Science (CS) courses offer conceptually disjointed disciplines to students such as *Software Engineering (SE)*, *Operating Systems (OS)*, and *Simulation*. Those disciplines are usually studied as separated issues that must be present in any CS syllabi. It is the effort of the student to interrelate the concepts and internalize the existing relations. The problem we have been dealing throughout the years is related to how teachers, educators, instructors, or tutors (used interchangeably in this paper) could introduce *interdisciplinary* aspects into those domain applications as well as opportunities and challenges that may become of those practices. We have the impression that it is often difficult to present students with several concepts at once; however, they must be exposed to those kinds of problems because it will be certainly required of them in the working place or research environments.

As time passes, we witness that systems, in general, are being pushed towards users as a *black box* with limited possibilities for interaction, *hacking* (in the good sense of the word, i.e., a curious approach to technology), and tinkering. Vendors are shipping products with embedded software that is seldom openly available. The job of educators is becoming increasingly difficult since CS students are just interacting with electronic gadgets and mobile applications without deepening the fundamental understanding and inherent concepts that lie underneath those technologies.

Today, a few computing courses in undergraduate levels are concerned with presenting students to different topics and expect them to understand difficult and sometimes problematic relations within areas. CS is a demanding field and decision makers working with education must take into account how to develop interdisciplinary abilities. This is not a trivial task because it is expected that students have reached a high maturity level in terms of problem solving, systems modeling, and programming. Another problem that sometimes arrives concerns the fact that students lack a *formal* and *algorithmic* training to define the problem at hand as well as to study the best tradeoffs in terms of performance *versus* maintainability when implementing systems. We hope to fill this gap here since we present a description of the challenge of simulating a virtual system and expect students to provide a valid and timely solution.

The contribution of this paper is to present the necessary steps to construct software consisting of a simulation of an OS and discuss how educators could profit from interdisciplinary concepts to instill students to study requirements elicitation, data structures modeling, simulation execution of multiple scenarios, and OS concepts, all in one project. Another outcome is directed to *team formation* when leveling skills qualifications. In many situations, due to distinct team backgrounds and schooling, they must work as a team, knowing each other strengths and weaknesses to produce *value* in projects.

The paper is organized as follows: Section 2 discusses the formulation of our research problem covering SE, OS, and simulation. We present our framework and modeling discussions in Section 3 with an implementation suggestion and, Section 4 presents our final considerations.

2. Problem formulation

Our approach to developing a working simulation of a virtualized system encompasses notions of different areas such as SE, OS and *Discrete Event Simulation* (DES). The objective is to describe and comment the necessary steps and *general mindset* to build a valid discrete model of a virtualized system with several architectural implementation details.

Interdisciplinarity is a valid approach to cope with the benefits of different ideas and to promote a cross-fertilization among various disciplines. Some authors have discussed how interdisciplinarity works in science contexts [1], where the topic is extensively discussed, and its fruitful implications help problem solving, in general. In this work, we are proposing an educational framework that combines CS disciplines with a mingling of theoretical and practical notions helping students to apply those concepts in real world situations. As they tackle the problem, they understand fundamental notions investigating inner relations, modularity, performance, and other factors common in enterprise settings.

2.1 Software Engineering

SE, in a broad sense, is related to all aspects associated with *software* (SW) design and how to transform concepts and ideas into executable components (e.g., applications) that enhance user interaction, extend learning experience, and increase the student productivity [2]. SE is an active research area within CS, preoccupied with requirements elicitation, definition of the most suitable development process (using a computer language), techniques to model and map requirements as working units (functionalities), testing (to verify if requirements were in fact implemented as well as quality aspects such as security, performance, etc.), and then delivering it to users and stakeholders (deployment and installation).

SE encompasses several disciplines in a CS program because there are numerous concepts to present to students. In this work, we focus the attention on the development process and programming. There are several methodologies to construct SW such as the *Unified Process* (UP) [3], *Waterfall Model*, *V-Model*, *Scrum* [4]*, Agile* [5],[6], *Extreme Programming* [7], among others. The methodologies are very conceptually diverse as different strategies are formulated for each technique, despite having the same general principle that is to develop SW in a proper fashion. UP conjured several methodologies and sought to unify them into one single definition. A refinement of UP is the *Rational Unified Process* (RUP), using a language called *Unified Modeling Language* – UML [8] that captures *Structural* and *Behavioral* arrangements by defining a set of diagrams. They are usually visual and capture user functionalities as *Use Cases*, followed by the derivation of *Activity Diagrams* (describing how processes interact, in what order, and how they synchronize activities), *Object Diagrams*, *Class Diagrams*, and *Sequence Diagrams*.

In 2001, SW process researchers proposed an alternative called *Agile methodologies;* i.e., proactive mechanisms that shift the project responsibility to the team, aiming to cope with frequent requirements changes that occur in many SW projects. Figure 2 exemplifies a Scrum practice, which can be considered an Agile method. The idea of Agile is to focus on process simplification, capturing requirements as *User Stories* (mapping a single functionality into one phrase) that constitutes the *Product Backlog*. Each story has a difficulty level assigned, and the methodology has the following characteristics: a) relying on small teams with different backgrounds and skills; b) having a *Product Owner* on-site (the main project stakeholder, or client); c) learning from bad and good experiences, and; d) ensuring short, focused *stand up meetings* (for brevity). The idea is to offer quick course correction mechanisms to align with the project objectives and adapting fast to requirements change, deliver minor working versions. Therefore, users can form an opinion regarding the SW as well as inspecting team performance in fixed slots (called *sprints*, ranging from one or two to a maximum of four weeks, depending on the project). There is a trend to intertwine concepts from UP and Agile altogether, as they are flexible definitions that are adaptable to each project's needs.

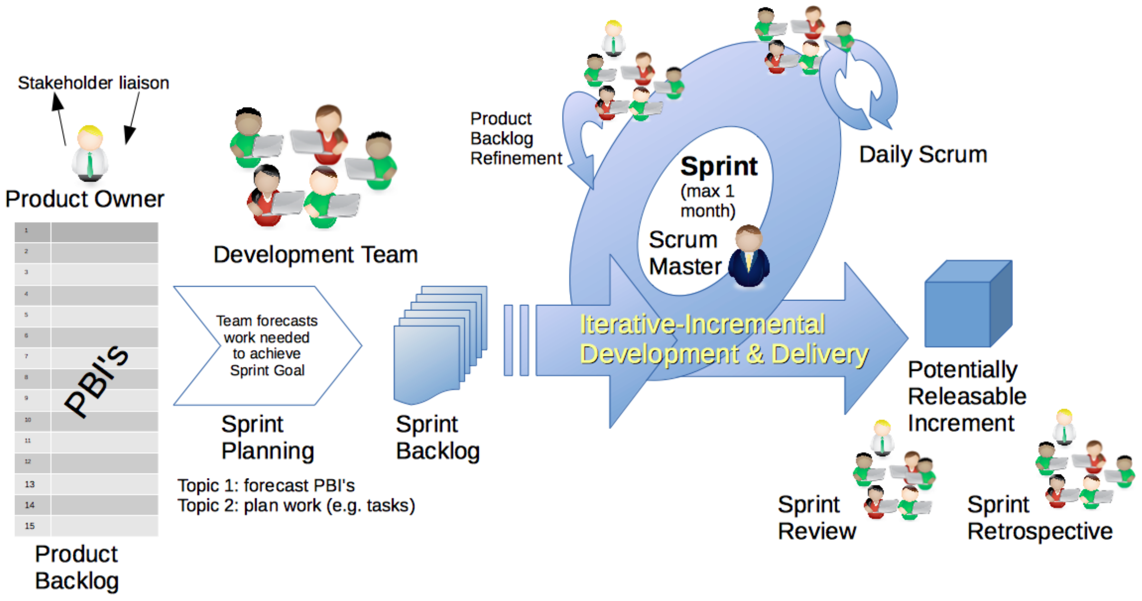

Figure 1: *Scrum practice showing sprints and daily iterations, where programmers choose some stories from the Product Backlog to implement and test*[1].

Once the SW team has decided which process will be used for the project, they must decide what programming paradigm will be the best fit: *imperative* or *object oriented* (there are others, such as

[1] Source: https://upload.wikimedia.org/wikipedia/commons/d/df/Scrum_Framework.png

functional, or logical, we will focus on just those two here). This is crucial because if this choice is poorly made, it will reflect on the amount of time to correct, debug, and maintain the produced SW. There are several *strongly typed* (the compiler enforces predefined type usage at *compile* time) programming languages such as C (strictly imperative), and object based such as C++, Java, and C#. There are also *weakly typed* (the type of variables is defined at *runtime*) *scripting* languages: Perl, Python, and others. The way languages are built allow us simple integration and possibilities to combine all approaches.

Developers should consider how they will implement SW, taking into account decisions about SW modules and data structures and how users will interact with the produced application. In SE those concepts are known as *coupling* (dependency among modules) and *cohesion* (how the elements in modules are related), and they are linked to the amount of work that probably will be needed to maintain the SW if poorly designed. Practical programmers aim at *low coupling* and *high cohesion*; i.e., modules are somewhat independent of each other, with high modularity, where each module is self-contained, needing only the elements present in the module they are located. This is a good indication that the SW is reasonably readable and maintainable by other programmers, an objective for SE practices in general.

The challenge when programming and designing SW is to build maintainable, standard (using code conventions for high source code readability), responsive, and scalable applications. Thus, *other* programmers can extend, improve, remove unnecessary parts, and document it effortlessly. Other factors affect timely project delivery such as team building, skill mapping, task allocation, technical leadership, and respectful member interaction. Those are mandatory abilities, and they are very hard to teach and emulate in a class setting, but they usually emerge in real (or very close to real) projects settings.

Experience is also important to make better decisions, especially when strict deadlines are in play. The idea is to work with heterogeneous teams with different backgrounds, skill levels, and experiences. SE encompasses a large set of concerns that is very hard to teach during a semester. It is important to mature concepts with experience, where students are exposed to different decisions, challenges and projects, something that is seldom offered due to lack of interdisciplinary initiatives in CS courses.

2.2 Operating Systems

An OS is an SW layer placed between the user and the *hardware* (HW) that offers several services such as process and memory management, easy access to I/O devices, virtual memory, file system organization, multithreading, security, and protection [9],[10]. It is the primary objective of OS implementations to offer *convenient* mechanisms so users can perform their tasks productively. Throughout the years, the importance of sound implementations of OS has become of great interest to the general computing research field as vendors that must perform reasonably with limited resources, the case of *Smart Phones* and *Embedded Systems*, for instance, introduced many new HWs.

The OS is the most important SW layer under execution inside any computer's *Central Processing Unit* (CPU), responsible for providing several services to access HW capabilities. Another function of the OS is to load, store in memory and execute several processes concurrently, control invalid accesses and references, and ensure process isolation. Examples of OS are numerous, such as GNU/Linux and its several *flavors* (Ubuntu, Red Hat, and Slackware), Microsoft Windows (Windows7, Windows8, XP, Windows Server), Apple Mac OS X, Oracle Solaris, and Berkeley Software Distribution (BSD).

A *process* is an instance of a *program* (i.e., an application) with a list of instructions written in different languages and compiled to target architectures. In terms of execution, they can be one of two types, *I/O-bound* (needing lots of input/output operations), or *CPU-bound* (it computes many data without having to access input or output devices). Since I/O operations are much slower than intra-CPU operation, I/O-bounded processes affect performance directly due to delays and communications problems. CPU-bounded processes are constantly executing in the CPU and seldom need I/O operations thus executing faster due to main memory proximity (normally using a fast cache hierarchy).

It is hard to anticipate the type of processes that will be executing at any time since its instructions could start performing several CPU-bounded instructions and suddenly start behaving like I/O-bounded. Some techniques do exist to anticipate the type in runtime (or even earlier); however, it is a very difficult task to be able to classify it prior to the actual execution of one type or another. From a CPU perspective, a process consists of a list of instructions to be executed from start to end, with loops and instructions flow.

The CPU works with several layers of memories such as *registers*, *caches* with several levels and sizes, *main memory*, a *secondary memory* where each process must be loaded into the main memory.

Once a process is loaded, it is assigned to specially designed *queues* that observe all processes in the CPU and choose the next process according to scheduling policies and rules (such processes are known as *short* and *long-term scheduling*). Examples of processes are the *scheduler* (a SW that chooses the next process to execute), the *dispatcher* (an SW that performs context switches), *services* (or daemons), *virtual machine processes* (processes that need a particular virtual machine for its execution, such as the *Java Virtual Machine* – JVM, for Java programming language or the *Common Language Runtime*, for C#), and *virtual machine managers* (VMM), explained in Section 3.3. A process has an identifier (PID, or *Process Identifier*), the *Program Counter* (or PC), a list of opened files, a state (*executing*, *waiting*, *idle*, and *ready*), its memory limits; i.e. the accessible memory only for the process, so it does not invade other processes memory space [9].

2.2.1 Virtualization and virtual machine processes

IBM researchers were the first to consider the execution of an OS inside another (e.g. a *Virtual Machine*, or VM), naming the technique *virtualization* [11], which is illustrated in Figure 2.

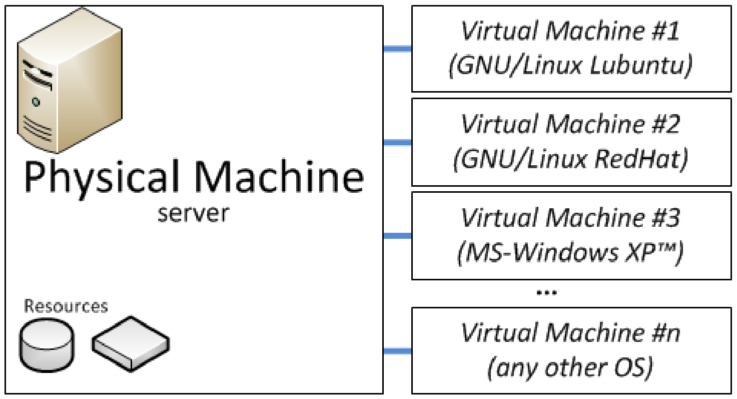

Figure 2: *Virtualization concept, showing the Physical Machine and virtualized platforms.*

This approach offers several rewards such as execution isolation, security, and the ability to run several VM instances. It is very easy to duplicate machines and configurations across clusters of homogeneous equipment. Additionally, it provides high scalability, ability to run legacy SW and economy of resources. The idea behind virtualization is to create a logical infrastructure on top of the OS enabling the installation and configuration of multiple VMs in the same HW infrastructure. As HW becomes extremely cheap, the use of virtualization arises as the remedy for all performance and capacity planning problems in heterogeneous networks.

During the past years, we are witnessing the massive adoption of virtualized solutions in corporate environments as it apparently solves more problems than it causes. However, despite this increased usage, the technology has limits possibly leading to poor performance, inability to meet quality requirements, and even causing SW to fail due to lack of resources as they are shared among the set of VM. The technique usually exhibits poor performance because it is another SW layer running on top of the *bare metal*, i.e., the *Physical Machine* (PM). Despite efforts to circumvent these limitations, performance will always be an issue when choosing virtualization. Another disadvantage concerns the acquisition of powerful machines in terms of processing power and memory size because PMs running virtualization must support the extra load gracefully.

Virtualization offers several tradeoffs for adoption since it has high computational costs, and it often overloads the PM as multiple VM instances are executing and sharing resources (disk, memory, network, and processor time) constantly. Performance analysts and quality engineers must regularly observe and monitor the PM and try to determine the impact of virtualization on the infrastructure.

2.3 Basic concepts of Discrete Event Simulation (DES)

DES is broadly used throughout several domain applications. It has been successfully applied from medical research to nuclear power plants and researchers use it as an important analysis and inference tool for dynamic and complex systems. The central idea of DES is to model any given system by defining the set of states they assume and the list of discrete events that causes changes from state to state. The strength of using DES consists on observing the reality under study and creating a representation (e.g., a model) that closely describes its operational behavior. Modeling involves the expertise of choosing the most important aspects to represent [12],[13],[14],[15].

Once the model is created, a computational technique for performance index derivation is implemented, and the system is repeatedly executed where the statistics are recorded as the simulation progresses. DES has the capability of unveiling hidden relations and offers an interesting analysis of *cause-effect* relations, usually making modelers and performance analysts to rethink certain parts of the system or redesign other parts to meet performance requirements. In contrast with DES advantages, however, some tradeoffs must be considered prior the choice of using simulation. For instance, DES modelers should pay attention to the desired *level-of-detail* because if it is too coarse, it will roughly represent the system whereas if too detailed, it will negatively influence the simulation execution, causing difficulties in the next phases. The analysis phase will be most impacted, because it deals with decision-making and resources allocation (usually financial). Simulation studies quickly deliver results using simple modeling primitives that always must be interpreted to make sense according to the problem.

It is usual to consider other performance techniques to compare the results, e.g., monitoring or analytical modeling [16]. It is worth mentioning that for systems with dynamic behaviors, the best choices are monitoring and simulation, whereas systems governed by known rates are better studied by analytical modeling, e.g., queues and networks of queues, using classic system performance evaluation equations, namely *Little's Law* [17]. DES and other simulation-based techniques allow the creation of *what-if scenarios*, where several parameters variations are made for many replications, with different outputs for analysis. It is important to inspect how the system under study is affected by various conditions and the mapping of the worst and the best scenarios for determining operational performance, best configurations, misalignments to specifications, and detection of underperformances and outages.

2.3.1 General operation

DES uses several concepts and mechanisms for proper operation, mainly discrete events, event rates, global simulation clock, a list of the past and future events, an event scheduler, statistic collection, and multiple executions (replications). The core idea governing any DES study is to decompose a system into discrete observable events. Time is paced according to these event changes, from an initial condition until a stop criteria is met, usually defined as a large number of executed events or a fixed time duration. The idea is that the system changes its global state only in specific moments dictated by the events [18].

Events can be thought as series of *stimuli* applied to the data structures within the simulator responsible for global system state changes. Those events are only fired in specific times, hence the name discrete. The events happen in random instants drawn from different probability distributions, such as *Exponential*, *Poisson*, *Normal*, or *Uniform*. The event list contains the main observations that modelers make on the system where they deem those as the most important ones. Once the event is created, one must assign a rate of event occurrence, a mapping that will be used to compute the time when the event will occur during the simulation.

The set of events is formed where the modeler must associate it with a valid rate in time, observing how often the event occurs in the system. Those values may come from systems operation logs (raw data) or observations. The chosen time or rate must work in similar scales so the results will make sense. The global clock is a simulation variable that can be fast-forwarded and re-initialized at will, allowing executions to have different outputs for analysis. Simulation execution can be stopped at any moment, where indices are computed since the simulation has initiated until it was stopped. There are several stop criteria for simulations where the most common ones are the execution for some distinct events, or for a fixed duration in simulation time. It is possible to run a simulation for a long period, performing what is called *long runs*. After a batch of several simulations is executed, one must compute interesting statistics in the data, helping the analysts understand the produced outputs as well as the data significance. For instance, the simulation could calculate confidence intervals and variance to determine if the produced results are within reasonable margins.

A simulation study comprehends several *phases*, such as *modeling* (assigning states and rates), *implementation* (if one chooses to build a simulator engine) or *tooling* (using a specific tool package for simulation), *setting up parameters* (duration, number of simulation replications, statistics to be stored), *execution*, *analysis*, *refinements and optimizations*, and *reporting*.

Simulation is a powerful technique to model and analyze dynamic systems; however, it has some drawbacks. For instance, one must define the proper *warm-up period* (time prior to statistics capturing) to be consistent, because otherwise it is hard to detect the effect of initial conditions and how much time is needed for simulations to run, as well as for how many times (replications). Another disadvantage is the definition of the *stop criteria*, the decision of when to terminate the simulation according to some rule such as some events or time, and working with large cycled *pseudo-random number generators*. Another problem that arises when simulating is due to modeling. It is important to capture the basic system operation and translate it into discrete events. The distribution of chosen time (or rate) is crucial since it is very sensitive, and it directly affects all discussions and decisions made regarding the constructed model.

2.3.2 Tools

In a simulation, there are several SW suites to model and execute; however, there are only a few options that help to implement a simulator, producing interesting performance indices. We can cite some comprehensive simulation infrastructures specific to networks such as *Network Simulator* – NS [19], and multipurpose such as *OMNET++* [20], and *adevs* [21]. We also highlight the *Arena* suite [22], having a modeling environment that offers various modeling primitives, attributes, and other elements that ease the simulation process in general. Arena also has several statistical inference tools implemented in the environment, such as the *Input Analyzer* tool, that performs data fitting for several distributions, integrating those results with the processes created inside the SW, which greatly simplifies the modeling and analysis of systems. The only disadvantage of using Arena is its high price; however, it is possible to download a version for educational purposes (it limits the number of objects that will be created).

2.4 Related work

There are several works, case studies, and modeling of several simulation applied to process management, health care, agricultural [23] and manufacturing [12]. However, to the best of the author's knowledge, there is no research where a combination of SE, OS, and simulation practices and educational frameworks that aim to teach those interdisciplinary approaches to students. Specific for OS simulation platform we mention [24], where the authors have developed an environment to simulate OS and applications running on diverse HW configurations, though we point out that this work is quite outdated. We cite the work of [25] that suggests teaching simulation using *Object Oriented* mechanisms. The work explained by [26] sought to use DES visual simulation to teach process modeling in management courses. A framework called SUCCESSFUL is discussed in [27] to teach DES with interesting results.

There is a growing interest in teaching Agile in SE courses, specifically targeting Scrum and Agile practices. For instance, we cite [28] that created an educational platform to teach Scrum using LEGO blocks. The work of [29] has studied card games to teach scrum to students, and [30] has defined *SCRUMIA*, an environment to teach Scrum practices. In [31],[32] authors followed a similar approach, where the interest to cope with *Agile/Scrum* learning is directed towards mechanisms. In [33] the authors have focused on *Agile* collaboration aspects within SE projects whereas [34] directed their attention to the simulation of Scrum development. In [35] a framework called *Kaizen* is devised that aims to develop and train high-performance SW teams, whereas [36] presents how to teach Scrum, presenting a discussion in terms of backlog analysis, requirements comprehension, and student feedback to the process.

3. Modeling, designing, and executing simulations in virtual environments

It is not trivial to combine all these disciplines into a working application running in a reasonable amount of time, responsively, and extensible to allow further extensions. There are several problems that must be addressed prior the *actual* programming, such as requirements, designing, functionality mapping, data structures modeling, event mapping, and other issues, described as follows.

3.1 General ideas and initial considerations

Before embarking on any SW development project, one must address a few concerns, so it runs smoothly over the assigned period. Figure 3 shows our suggestion with five phases for a project summing ten 3-hour classes of an interdisciplinary study in an undergraduate CS course.

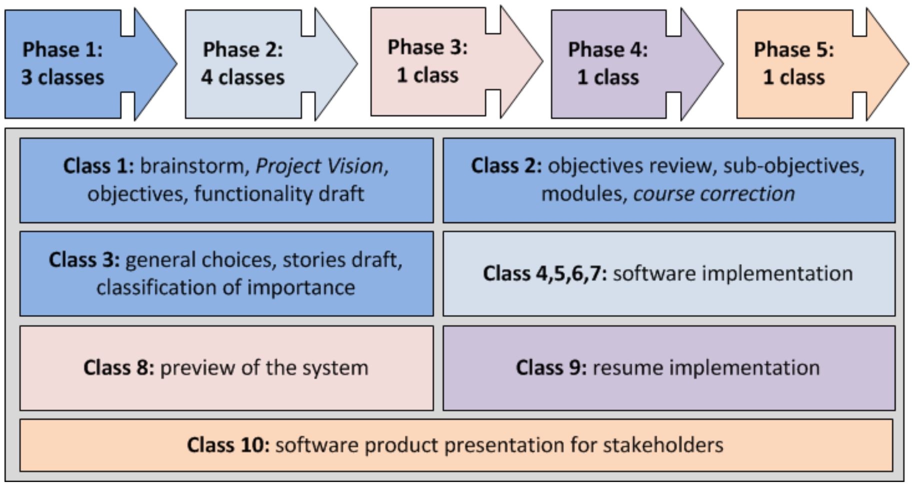
Phase 1: 3 classes
Phase 2: 4 classes
Phase 3: 1 class
Phase 4: 1 class
Phase 5: 1 class
Class 1: brainstorm, Project Vision, objectives, functionality draft
Class 2: objectives review, sub-objectives, modules, course correction
Class 3: general choices, stories draft, classification of importance
Class 4,5,6,7: software implementation
Class 8: preview of the system
Class 9: resume implementation
Class 10: software product presentation for stakeholders

Figure 3: *Project general organization in ten 3-hour classes within a semester.*

- **Phase 1 (Class 1: 3 hours):** In this stage, the students must derive the project *objective*, or, as Scrum/Agile calls it, the *Product Vision* (PV). It represents what the system should embody to stakeholders in terms of general functionalities and other important features that must be present in the final product. Accordingly, as an instructor, or a teacher, it is interesting to initiate a general brainstorm session (with all class), during at least 3 hours, where they: a) listen to the general SW idea and some functions for the future project (by the instructor); b) organize themselves into groups of four or five elements (e.g., by affinity or friendship); where each group c) writes down the PV document. Then, the teacher should equilibrate the teams and *intentionally disturb* the natural pairings by changing several components among the groups. It is important to remind that it is under training the creation of heterogeneous teams with different backgrounds, so this remapping is crucial for the next stages. An important issue is directed at the *brainstorm* practice: there should not be any constraints, promoting that all students could externalize *any idea*. The students must provide a list of requirements both quantitative and qualitative, such as *"software should handle multiple scenarios and parameters using input files"*, or *"system must run on GNU/Linux platforms"*;
- **Phase 1 (Class 2: 2 hours, 1 for discussion):** Next, the groups will start to extract SW objective and sub-objectives, trying to assign effort and time, grouping tasks that are conceptually similar, hopefully designing a draft of the set of existing modules and user interactions. Each group provides an informal

document detailing these ideas in cursive writing (no computers/tablets, only paper, and pen). This procedure should last two hours, and the remainder of the class is dedicated to discussions of alternatives and course correction (i.e., look for change of focus, misinterpretations, and bad decisions that the students usually commit). The idea of this class is to train and study *separation of concerns*;

- **Phase 1 (Class 3: 2 hours, 1 hour for discussions):** This class will handle choosing the most conveniently SW development process, the candidate programming language (according to defined requirements), and a draft concerning the list of *User Stories* (if Scrum/Agile), or *Requirements* (if UP). Students should be able to classify them by importance and effort, and which part of the system will be available for initial testing and debugging;
- **Phase 2 (Class 4, 5, 6, 7: off-site):** Implementation starts. The Phase 2 will take a full month (30 days, or 4 classes); consequently, students can develop functionalities according to priorities. It is interesting to check progress every two weeks, so they do not deviate from the PV. The instructor remains available for questions and project related information;
- **Phase 3 (Class 8: 3 hours):** SW preview presentation (only for the instructor) for each group (a 30-minute exhibition is sufficient), where students highlight activities performed in the past month and indications of new directions, what is left to implement, system user interaction details, problems they are addressing and other project information such as student compromise level, additions, and incorporated technologies (and the reasons why they were missing in the original proposition);
- **Phase 4 (Class 9: *off-site*):** Students resume the project by adding missing details discussed in Phase 3 as well as refinements, optimizations, and testing. The teacher remains available during this period to help students on some issues that may or may not arise, since they lack experience in producing high-quality SW, with performance and responsiveness;
- **Phase 5 (Class 10: 3 hours):** this is the last phase, where students present their solutions in class, showing project details, functionalities implementation, SW demonstration, interesting internal algorithms, input/output analysis, results (if present) in the form of a scientific article, and other information regarding the project. In the end, the instructor constructs an overview reporting with

some final considerations, and according to predefined criteria, evaluates every project, grading them and informing the results to the students, ending the project;

It is interesting to point out that the project must have a specific period (we suggest 30 hours) because the students work better when faced with strict deadlines.

3.2 Software requirements elicitation

Following, we present a project suggestion that instructors may adopt to work with students. Table 1 shows some basic project facts. For this project, we chose to use the Scrum/Agile requirements format using *User Stories*, due to ease of elicitation.

Table 1: *Main specifications and details of the project.*

Objective (one sentence)	*"Implement a DES simulator of a virtualized architecture that allows multiple scenario formulation and advanced statistics analysis"*
Sub-objectives and general decisions	• Use an *Agile* SW development method to describe activities and tasks; • Map the necessary functionalities in a simple format; • Work with the *Product Owner* on-site so that each small release (*sprints* of two weeks) contains useful functionalities accorded by the parties; • Study how processes execute, how much processing they require, what type of processes are possible in these contexts; • Plan the set of events and the simulation core for the project; • Design a modular SW (due to coupling and cohesion) that will allow future extensions and functionality aggregation easily; • Implement a responsive simulator that allows multi-scenarios definition, multiple replications, saving valid statistics for later analysis; • Languages: C/C++ (performance), Perl (*Practical Extraction and Reporting Language*) for transform output to a graphic format; • Supporting SW: GNU/Linux (OS), *Gnuplot* (for graphics)
Project codename	*vProcess Simulator* Release 0.0.1 (each next release, the number is reviewed)
Team specification, pre-requisites and skill requirements	At least three and at most five members with object-oriented and scripting programming skills (at any level), GNU/Linux experience, and SW design (data structures/algorithms), with basic to intermediary level

The decisions set forth earlier were based on simple assumptions about this kind of project, as they reflect real world projects. These decisions must be made very early (at least in the first three classes), so they can meet the deadline properly. Table 2 describes the *Product Backlog* with all the necessary functionalities that must compose the final version. There is only one type (*users*) for this project, and there is no impediment to adding other user types to the specification (*administrators*). Assigning points to activities is hard and depends on the team. The table shows a suggested difficulty for each story, with a recommended time to testing (to finalize the task successfully, diminishing rework). We used a *Fibonacci Scale* (i.e., an arithmetic progression of 1, 1, 2, 3, 5, 8, 13, … where the next element is the sum of the last two, and so on) to address difficulty. However, any other point assigning technique could be used as well.

These are the basic system requirements and user possibilities. They list everything that the user is supposed to inform and how they will perform those several analyzes. Those difficulties depend on the programmer expertise; therefore, they may vary a lot. We propose the use of a *Skill Correction Factor* (F) that is simply a multiplicative factor that could be applied to align the total points according to the team skill level (e.g., F=5 for *beginners*, F=3 for *intermediate* level programmers, and F=1 for *experienced*). It is also possible to break one difficult story in others, reducing and reviewing the assigned difficulty accordingly. A total of 80 points (with testing) is needed for this project. Our Sprint has 20 points, so they should be able to finish the project in the planned four weeks (refer to Phase 2, in Section 3.1).

Table 2: *Product Backlog with all User Stories that must be present in the final release.*

#	***User Stories (US) written in Agile format: "USER … TASK … SO … OBJECTIVE"***	***Difficulty (Fibonacci Scale) + Testing effort***
US1	***USERS*** should be able to set up input file containing simulation parameters such as simulation time, the total number of replications, output folder path, statistics capture, graphic options, and physical machine resources usage ***SO*** they could analyze multiple simulation scenarios	(8 + 3) * F
US2	***USERS*** should create sessions where simulations are run, and their output files are saved (assigned to this session) ***SO*** they can create analysis sessions to inspect different parameters tradeoffs	(8 + 3) * F
US3	***USERS*** should configure simulations by time, a number of events or some stop criteria such as a variable reaching some value, which uses an input properties file containing simulation parameters and events rates ***SO*** they can work with simulation sessions that benefit multiple analysis	(8 + 3) * F
US4	***USERS*** should erase simulation sessions (from the disk) and organize existing sessions by assigning names, date/time, and other related information ***SO*** they can define multiple executions	(5 + 3) * F
US5	***USERS*** should know if the simulator will execute in a valid period according to the input file definition, where the simulator calculates a prediction for the simulation time, and present warnings to users case, those calculations are invalid ***SO*** it is possible to change the input parameters and restart simulation execution	(3 + 3) * F
US6	***USERS*** should inspect generated simulation output with graphics that will show different metrics captured during the execution ***SO*** they enhance the analysis phase with evidence of operation of the simulated parameters	(5 + 3) * F
US7	***USERS*** should combine different outputs from different executions into one graphic, ***SO*** they can analyze the impact of choices more efficiently	(13 + 5) * F
		Total (*suggested*) : (50 + 30)*F = 80F

It is important for the instructor to explain to the students that they must be able to finish the project in two two-week sprints (one-month iteration). Testing is usually neglected; however, on this occasion, this task is as important as the project itself (i.e., the project is 50 points, and testing is 30). We also recommend that students use *Unitary Testing* at length techniques (it is available in many *Object Oriented* languages such as Java, or C++) for these activities, as they catch simple defects very easily and directly affect the final product quality. Next, we present some internals needed by the simulator, such as some OS and DES considerations.

3.3 Operating System considerations and choices

It is rather difficult to simulate every aspect of an OS; hence, one must abstract some fundamental operations and then study the implications in terms of metrics. We have decided to represent the OS as having a working CPU, with a list of the running processes (the *Ready List*) and other details. Each process is abstracted into a list of instructions that must be handled by the CPU sequentially. The number of instructions is an indication of the effort to run the process; i.e., a process with 100 instructions will execute much faster than a 1000 instructions process. Evidently, this is a very simplistic and abstract view of a process because loops and other repetition commands could affect these measures enormously.

The objective of the CPU in this context is to execute processes, manage the memory and other components, such as I/O operations (that are slower than CPU operations). There is also a distinction in terms of the process types under execution, for instance, normal processes are different from more heavy processes, such as VMMs (Oracle's *VirtualBox*, or *Xen*), or VM processes (JVM).

It is worth mentioning that our approach has simplified other important aspects of many modern OS implementations such as threads, concurrent behavior, virtual memory, instruction localization, *Translation Lookaside Buffer* implementations that impact hit/miss ratios for memory, and cache memory hit/miss. We stress the fact that we are working with a valid approximation of a system.

3.4 Simulation aspects and events

Figure 4 shows that after the data structures are initialized in the *"Start"* process, the simulation keeps running until the stop criteria is achieved. If not, the event must be executed (process *"Execute Event"*), its statistics collected (process *"Collect Statistics"*) and then it resumes the operation iteratively.

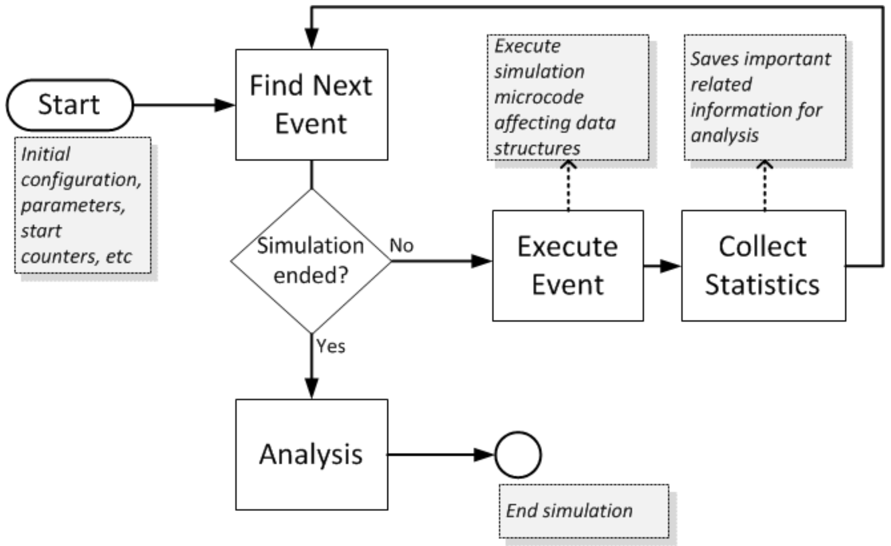
Start
Initial configuration, parameters, start counters, etc
Find Next Event
Simulation ended?
No
Yes
Execute simulation microcode affecting data structures
Execute Event
Saves important related information for analysis
Collect Statistics
Analysis
End simulation

Figure 4: *Simulation general execution, processes, and considerations.*

The *"Execute Event"* procedure determines the next event from the list of events to run according to the assigned rate. The SW picks a random number from the Uniform distribution; convert it to an Exponential figure and then selects the one with the least value, simulating the fact that this event happens earlier. Then, it should be triggered before all others (only one is picked at each global time advancement). This simulation procedure dispenses the simulation scheduler and provides the same results. If the simulation has come to an end (because simulation time or the total number of events has reached a predefined value or other user criteria), the analysis phase starts where the SW produces output that promotes studies with graphics showing system statistics and performance metrics. Table 3 displays the list of procedures needed by the simulator to initialize internal data structures.

Table 3: *List of procedures for the simulation and descriptions.*

Procedure	***Description***
Start Simulation	Choose number of processes already executing and process type. Start the simulation *Global Clock* and internal structures. Initialize the *event queue*, instantiate the CPU, create processes and assign them to CPU, update CPU characteristics (e.g., memory)
CPU	Initialize the list of processes, the memory size, and other characteristics needed

Process Creation	Assign PID (*Process Identifier*) and the total number of instructions, choose process execution type (CPU or I/O bound), assign process type - whether it will be a *Normal Process* (P), a *Virtual Machine Process* (PVM), or a *Virtual Machine* (VM)
Statistics Computations	Saves number of executed events, updates the *Global Time*, saves internal statistics, stores data structures (in our case, CPU and memory) according to events
End Simulation	Transform output data to textual format readable by external tools (e.g., *Gnuplot*), produce graphics and plots for analysis

The *"Start Simulation"* procedure instantiates the CPU data structure with a list of randomly chosen processes, depending on user preferences (more information on Table 5). It is often required to start the system with some processes, simulating a server with only services and no new processes arrive for execution. This simulation is DES-based. Therefore, after the system knows which events are more important, the simulation needs to know the time occurrence of each event (to decide the next event to run), the probability distribution (in our case, the majority of events are Exponential, drawn from Uniform and converted [12]), and how all the simulation data structures are changed when the event is triggered and run. Table 4 shows the list of events (particular to this project) and its description.

Table 4: *List of events, descriptions and simulation microcode.*

Event name		***Description***
Arrival Events	`PA_PVM`	The arrival of a process virtual machine (e.g., JVM), a process that uses a considerable amount of resources to be executed. For an application server, for example, this process could impair memory and CPU
	`PA_P`	The arrival of a '*normal'* process, a lightweight process that starts execution and ends healthily. It simulates user processes such as text editors, browsers, OS basic processes that must be executing, and so forth.
	`PA_VM`	The arrival of a virtual machine, a more *heavy* process, that consumes many resources and uses CPU more often, influencing performance. It corresponds to a VMM running on memory
Other Events	`PE`	Represents the process being executed in the simulator; picks a list of processes and execute N instructions each. This event indicates that the simulator should choose one or more processes to execute
	`PD`	Process departure: chooses M processes to exit the system, simulating that some processes are interrupted, exit with error, or simply stop executing because they have ended operation
	`PI`	Process in an invalid state: picks a process and assign a flag indicating that it is behaving abnormally; i.e., it simulate a virus, a deadlock, a *zombie* process, and so forth.

The simulation runs for a precise duration (set by users) where it is possible to define the total number of replications; i.e., how many times the execution will be executed, and statistics collected. The idea is to study the parameters influence on the simulated OS structures such as memory, disk, and other CPU characteristics (e.g., CPU utilization). The general operation of the simulator is to keep running processes (decreasing the number of instructions) as the simulation time passes, capturing statistics regarding the processes, the memory, and collecting events information. When the simulation ends, the SW must be able to transform the huge sized *raw data* concerning replications to meaningful graphics.

3.5 System architecture and main data structures

It is relatively easy to implement a DES. The programmers should choose the right data structures and algorithms to perform the necessary tasks. *Object Oriented* (OO) languages are natural choices due to simplicity, modularity, rapid prototyping, testing, and decomposability when coding. Figure 5 details the *Class Diagram* (defined in UML) for our suggested project. Note the extensive use of *enumerations* as data types used that provides sensible maintainability as well as readability to the SW.

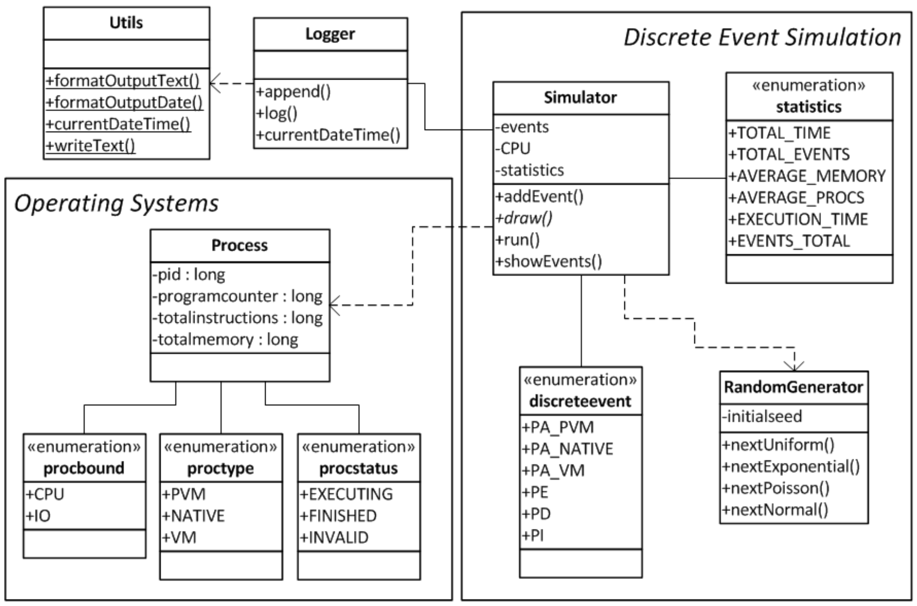

Figure 5: *Class Diagram for the suggested project.*

Additionally, the figure shows the main data structures (e.g., Simulator, Process, CPU, enumerations) and their connections. Some auxiliary variables are also present, such as the statistics array (of *long* data type), other objects and counters, specific to each object. We chose to aggregate a log writer called *Logger* to facilitate the analysis phase. This module handles creating output files containing information for future investigations. We suggest the implementation of a simulation *core* because it contains several standard commands and methods such as random number generators (according to probability distributions), next event selection and execution, and saving important simulation data (the global clock, data structures internals, and other related information).

This interdisciplinary project requires educators' expertise crossing several domains because they should capable of helping students, interact, correct/advise, help redefining the project to align with its needs, and many other factors. The instructors should also have a broad experience in programming and testing, managing real world projects of sizeable scopes, as well as client interaction and requirements elicitation, thus the students can profit and learn from an experienced *mentor/guide*.

We suggest the C++ programming language for speed and convenience since it provides a large community offering support, as well as other forms of help such as books and reference. We recommend the Perl scripting language for data transformation and the *Gnuplot* package for graphics and plots. The SW should be thoroughly tested for the presence of defects because it executes several times, performs intensive computations for statistics purposes and deals with floating point calculations intensively. We also note that it is not sufficient for students to implement the DES for the OS and virtualized processes, it is crucial to raise awareness to usually neglected non-functional requirements such as performance, timely execution, maintainability, responsiveness, user interaction (friendliness), decision-making capabilities, security, and invalid accesses. Those requirements are usually inspected after the defect discovery and testing phase and it is related to the general SW quality aspects, those items that directly affect user experience and broad adoption.

3.6 Results and discussion

Our implementation uses input files and has no *Graphical User Interface* (GUI), only textual interaction and visual plot analysis. Table 5 shows general simulation parameters (present in an input file), easily available to users, where they insert the project parameters, the event rates and output choices.

Table 5: *List of properties and general parameters for the simulation.*

Property/Parameter	*Description*

General Simulation Parameters	`Number of initial processes=10   // system start state`
	`Memory=5                         // scale: gigabytes`
	`Min instructions=1000            // values per process`
	`Max instructions=100000          // values per process`

	`Max events=infty                 // infinity or number`
	`Total Simulation Time=1000       // scale: hours`
	`Replications=25                  // between 1 and 1000`

Events (rate of occurrence)	`PA_PVM=100                       // these numbers are` ***`rates`***
	`PA_P=50`
	`PA_VM=15`
	`PE=75`
	`PD=50`
	`PI=5`

Output decisions	`Graphic format=png                // choices: png, jpg, pdf`
	`File data format=DDMMAAAA_HHMM.txt`
	`Generate PDF report=yes`
	`Clear previous sessions=no`
	`Combine replications=yes`

The table shows the parameters that the simulator could be started to analyze different virtualized systems. It is important to notice that the event list, for this case, is fixed to those present in the file. However, the system is flexible enough to allow another programmer to add simply other events and increase the event rates in the properties file accordingly. Figure 6 shows the PM memory size of a simulation having 25 replications (one for each line) in 10000 seconds (2.7 hours) of *simulation time*.

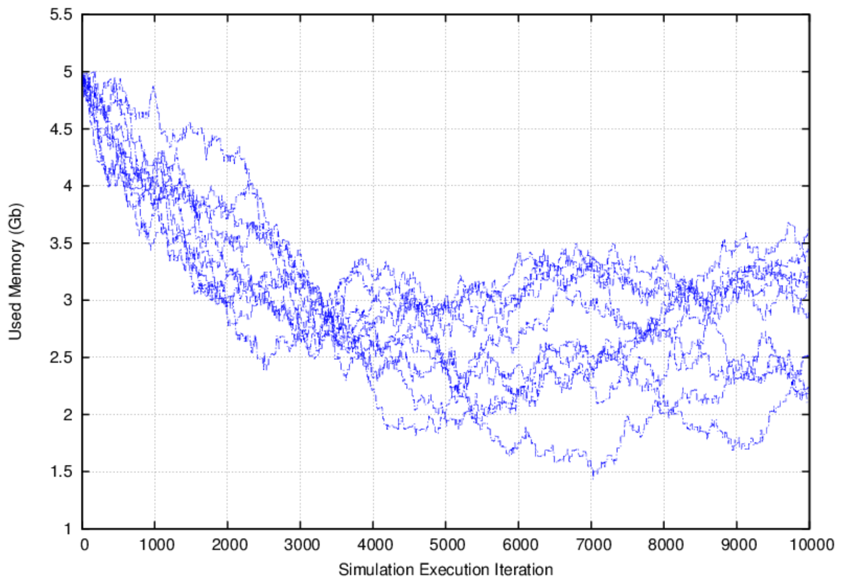

Figure 6: *Memory results for a simulation comprised of 25 replications.*

According the figure, we notice that there is a trend for the memory to keep usage at 3 Gb (according to the simulation). The parameters used are described in Table 5, explained earlier. This is some analysis that could be done with our framework. To build such graphical output representation, the students will have to work with numerous log files, *Gnuplot* files, and several results from the replications.

3.7 Application of a questionnaire to actual practitioners

Implementing scientific software is very difficult, especially using modern development methodologies where the people involved are not used to designing systems with strict performance and usability requirements. We have devised a questionnaire (Appendix A), where our objective was to address how to develop the scientific software presented here using Scrum, and what are the difficulties when working with this methodology. Our questions aim to:

- How to measure effort for each User Story (US) according to a broad problem definition;
- Attest programming efforts according to each coder's experience and background;
- Investigate whether the methodology has an impact on software project decisions;
- Compute the total number of required professionals for the size of this project (Team Formation);
- Measure the necessary effort to design software projects before actual implementation, relying only on US descriptions;
- Raise awareness as to the decision impacts on necessary knowledge for this project;
- Discover the knowledge lacking from students of system engineering in terms of formation;
- Apply student knowledge on a real world setting of a scientific software development project;
- Consider the roles of implementation and testing in the project, where they must assign different effort levels for each dimension;

We had limited the scope when we applied the questionnaire to the software analysts and programmers with considerable experience (from one to five years), with different roles across software development companies, as well as students and researchers from Computer Science and Systems Engineering. We have prepared a short version of the software requirements as supplemental material for consultation. The questionnaire had open and closed questions, and was applied to 29 responders where the main results and observations are summarized as follows:

- A vast majority of responders (58.6%) have answered that is a professional programmer for more than three years, whereas 24.1% between one and three years. So, 82.7% of our sample is experienced in programming, only 17.1% considered their level of programming 'low'

- The responders' occupation within companies has varied broadly, from analysts, architects, coders, students, and university researchers
- The responders did not reach consensus on several key issues of the software development and testing, such as:
    - The effort for each US ranged widely between 1 point (the minimum value) to 13 points (the maximum amount) for development and testing, where the experience has varied as well (from beginner to experienced)
    - Question 11 *"In your opinion, how many software developers are required to code this system in less than 3 months?"* has not raised consensus as well, responders have varied their answers from two programmers to 20. However, the most cited value (mode) was three
    - Question 12 (*"In your opinion, what should be the profile of professionals to code such systems?"*) was also divergent: responders have argued that the system could involve several (more than 5) beginners, or just two senior developers, and many answers have considered intermediary level programmers and entry level testers
- We discovered that several professionals (24.1%) have selected the option *"Combination of Scrum & RUP"*; i.e., responders are mixing different methodologies, extracting different strengths and weaknesses and adapting it to their reality
    - 13.8% have responded "Other agile methodology"
- A large sum of responders (65.5%) work on personal coding projects, enlarging experience and learning continuously.

Table 6, as follows, selected 19 valid answers from responders (excluding empty responses), relating the experience (Question 1: *"How long are you a software programmer?"*). Each cell has a pair [D, T], where D stands for "Development" and T for "Testing". The table shows the lack of consensus when assigning efforts, regardless of experience. We also observe that several tasks are greater than four (4), indicating that the task is sufficiently complex to justify a breakdown of the User Story in less difficult tasks.

Table 6: *Experience versus effort according to responders. Each cell marks a pair consisting of [Development,Testing] values.*

#	***How long you are a software programmer?***	**US1**	**US2**	**US3**	**US4**	**US5**	**US6**	**US7**
1	*Less than one year*	2,2	3,3	1,1	5,5	2,3	2,3	2,5
2		5,5	8,5	5,5	3,5	8,8	13,13	3,3

3		5,5	8,8	8,8	3,3	5,5	5,5	5,3
4		8,3	5,5	3,1	2,1	3,2	5,2	1,1
5	*Between one and three years*	13,8	8,5	8,5	5,3	8,5	13,5	8,5
6		2,1	3,2	3,2	2,1	3,2	3,1	3,1
7		3,5	8,5	8,5	5,5	8,8	3,3	5,3
8		5,2	3,2	5,3	2,1	3,2	8,3	8,3
9		8,13	5,8	8,8	8,8	8,8	8,8	13,8
10	*More than three years*	1,1	1,1	1,1	2,2	5,5	13,13	8,8
11		13,8	5,3	8,5	8,8	5,3	13,13	13,13
12		13,8	3,2	13,13	13,5	5,3	2,2	2,2
13		2,1	13,8	2,1	5,2	8,5	8,5	8,5
14		3,3	5,3	5,5	5,5	5,5	8,13	13,13
15		3,3	5,3	5,3	2,2	2,2	3,2	3,2
16		5,13	8,13	5,8	3,13	13,13	13,13	8,13
17		5,3	13,13	13,13	8,5	3,3	5,5	5,5
18		5,3	3,2	3,2	2,2	5,3	8,5	3,2
19		5,3	3,3	5,5	3,3	5,5	8	8

It was interesting to analyse required software efforts for scientific applications, as many professionals and beginners have completely different views when estimating software activities. These findings urge us to investigate how to estimate software soundly and deeply, as broad variations for effort are in place. We know that the developers must agree on effort somehow in the Scrum practices, by discussing why each task was assigned high or low values, however, it is interesting to detect that initially, people perceptions to difficulty are very diverse. Our approach here suggested a Skill Correction Factor for each story, and our findings based on the responses reflect the importance when trying to estimate effort to User Stories.

4. Final considerations and future works

Interdisciplinarity concerns are usually very difficult to cope since it crosses several application domains and combines them into an objective that is greater than the sum of its parts. Those issues are recently receiving a generous amount of interest from different research communities. We point out that the present work offered an educational framework to tackle scientific requirements. The present paper discussed an educational framework to implement efficient DES tool for virtualized environments.

The main difficulties that students may present are related to discovering the list of discrete events required for the simulation, how to break down the project into parts and modules, how to prioritize tasks, and teamwork. There are specific moments where educators must be present to help students because difficulties should be addressed as they happen, diminishing defects and misinterpretations. We emphasize that each project has its needs and characteristics, so practitioners should feel free to choose the proper process, languages, and applications for each assignment.

Essential skills could be thought such as abstraction (a difficult concept to convey to students), designing SW before implementation, good programming practices, teamwork, technology transfer among team members (due to different expertise and background), the need for performance and responsiveness, user-friendly interaction, and data presentation. A large part of the project is directed at planning and testing, activities that are usually neglected in many projects. They directly affect the quality of the final product as it relates to low reworking levels and unnecessary meetings to correct defects.

4.1 Next advancements

We are currently offering an extension course having as pre-requirements notions of programming, Scrum/Agile methodologies, OS, and DES. This project is suitable for both beginners and intermediate students who would like to be exposed to Scrum/Agile methodologies, work with a programming team, or expand the knowledge of OS and DES. It is also possible to use this educational framework as a benchmark to teach Scrum or other Agile methodology to heterogeneous teams.

As future works, we plan to document the project findings with several teams and capture the deficiencies and advantages of our approach, to understand team dynamics, stories creation, and other interesting considerations such as the emergence of new structures and concepts (e.g., the need for consistent testing, required extra modules), performance considerations and the limits of computations; i.e., how many events could be defined without impairing the actual simulation execution speed.

References

[1] F. Morillo, M. Bordons, I. Gómez. Interdisciplinarity in science: A tentative typology of disciplines and research areas. *J. Assoc. Inf. Sci. Technol. 54*, 2003, no. 13, pp. 1237-1249.

[2] I. Sommerville. *Software Engineering*, Pearson, 2010. 9th edition, 792p.

[3] I. Jacobson, G. Booch, J. Rumbaugh. *The unified software development process*. Addison-Wesley, Vol. 1, 1999.

[4] L. Rising, N. Janoff. The Scrum software development process for small teams. *IEEE Software* 4, 2000, pp. 26-32.

[5] R. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2003.

[6] M. Fowler, J. Highsmith. The agile manifesto. *Software Development 9*, 2001, no. 8, pp. 28-35.

[7] K. Beck. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[8] J. Rumbaugh, I. Jacobson, G. Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[9] A. Silberschatz, P. Galvin, G. Gagne. *Operating System Concepts*, Wiley, 9th Edition, 2012, 944p.

[10] A. Tanenbaum, H. Bos. *Modern operating systems*. Prentice Hall Press, 2014, 4th Edition, 1136p.

[11] J. Smith, R. Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann, 2005, 1st Edition, 656p.

[12] W. Kelton, A. Law. *Simulation modeling and analysis*. Boston: McGraw Hill, 2000.

[13] G. Wainer. *Discrete-event modeling and simulation: a practitioner's approach.* CRC Press, 2010.

[14] J. Sokolowski, C. Banks. *Principles of modeling and simulation: a multidisciplinary approach*, John Wiley & Sons, 2011.

[15] W. Stewart. *Probability, Markov chains, queues, and simulation: the mathematical basis of performance modeling*. Princeton University Press, 2009.

[16] R. Jain. *The Art of Computer Systems Performance Analysis – Techniques for Experimental Design, Measurement, Simulation, and Modeling. Digital Equipment Corporation*, John Wiley & Sons, 1991.

[17] E. Lazowska, J. Zahorjan, G. Graham, K. Sevcik. *Quantitative system performance: computer system analysis using queueing network models.* vol. 84. Englewood Cliffs-Prentice-Hall, 1984.

[18] S. Ross. *Simulation*, 5th Edition, Academic Press, 2012.

[19] G. Riley, T. Henderson. The ns-3 network simulator. *Modeling and Tools for Network Simulation*, pp. 15-34, 2010.

[20] A. Varga. The OMNeT++ discrete event simulation system. *Proceedings of the European simulation multiconference (ESM'2001)*, Vol. 9, No. S185, pp. 65, 2001.

[21] J. Nutaro. *ADEVS (A Discrete EVent System simulator)*. Arizona Center for Integrative Modeling & Simulation (ACIMS), University of Arizona. Available at http://web.ornl.gov/~1qn/adevs/, Accessed July/2015.

[22] *Rockwell Automation: Arena Simulation Software*, Available at: http://www.arenasimulation.com/, Accessed July/2015.

[23] R. McCown et al, G. L. Hammer, J. N. G. Hargreaves, D. P. Holzworth, D. M. Freebairn. APSIM: a novel software system for model development, model testing and simulation in agricultural systems research. *Agricultural systems 50*, 1996, no. 3, pp. 255-271.

[24] M. Rosenblum, S. Herrod, E. Witchel, A. Gupta. Complete computer system simulation: The SimOS approach. *Parallel & Distributed Technology: Systems & Applications*,1995, IEEE 3, no. 4, pp. 34-43.

[25] J. Tyszer. *Object-oriented computer simulation of discrete-event systems*. *Springer Science & Business Media*. 2012, vol. 10.

[26] S. Taj, E. Mousavidin. Using discrete event visual simulation to teach process modelling in MBA operations management courses. *Int. J. of Simulation & Process Modelling 10*, no. 1, 2015, pp. 45-64.

[27] H. Garcia, M. Centeno. SUCCESSFUL: a framework for designing discrete event simulation courses. *Winter Simulation Conference*, 2009, pp. 289-298.

[28] M. Paasivaara, V. Heikkilä, C. Lassenius, T. Toivola. Teaching students scrum using LEGO blocks. *Companion Proceedings of the 36th Int. Conference on Software Engineering*, 2014, pp. 382-391.

[29] J. Fernandes, S. Sousa. Playscrum-a card game to learn the scrum agile method. 2nd *Int. Conf. on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, 2010, pp. 52-59. IEEE, 2010.

[30] C. von Wangenheim, R. Savi, A. Borgatto. SCRUMIA – An educational game for teaching SCRUM in computing courses. *Journal of Systems and Software 86*, 2013, no. 10, pp. 2675-2687.

[31] V. Devedžić, R. Saša. Teaching agile software development: A case study. *IEEE Transactions on Education*, 54, no. 2, 2011, pp. 273-278.

[32] G. Rodriguez, Á. Soria, M. Campo. Virtual Scrum: A teaching aid to introduce undergraduate software engineering students to scrum. *Computer Applications in Engineering Education* 23, no. 1, 2015, pp. 147-156.

[33] M. Kropp, A. Meier, M. Mateescu, C. Zahn. Teaching and learning agile collaboration. *IEEE 27th Conference on Software Engineering Education and Training (CSEE&T)*, 2014, pp. 139-148.

[34] S. Ramingwong, L. Ramingwong. Plasticine Scrum: An Alternative Solution for Simulating Scrum Software Development. *Information Science and Applications*, 2015, pp. 851-858.

[35] B. Estácio, R. Prikladnicki, M. Mora, G. Notari, P. Caroli, A. Olchik. Software Kaizen: Using Agile to Form High-Performance Software Development Teams. *IEEE Agile Conference*, 2014, pp. 1-10.

[36] V. Mahnic. Teaching Scrum through team-project work: students' perceptions and teacher's observations. *International Journal of Engineering Education* 26, 2010, no. 1, pp. 96-110.

Appendix A – Questionnaire

Questionnaire – Part A

1. How long are you a software programmer?

☐less than 1 year ☐between 1 and 3 years ☐more than 3 years

2. How long are you a software analyst?

☐less than 1 year ☐between 1 and 3 years ☐more than 3 years

3. How do you consider your level of expertise in programming?

☐ inexperienced ☐ intermediary ☐ experienced ☐ Other: ______________________________

4. Select your undergraduate major degree (mark whether currently attending or not)

☐ Computer Science ☐ Systems Engineering ☐ No formal education/autodidact

☐ Information Systems ☐ Similar undergraduate course in technology

☐ Other: _____________________

5. Which software development methodology do you use on a daily basis?

☐ Scrum ☐ *Rational Unified Process* (RUP) ☐ Combination of Scrum & RUP

☐ Other: ___________________

Instruction: if you answered "*Scrum*" (or "*Combination of Scrum & RUP*") in Question 5, read the above text, otherwise go to **Part B**.

Please, read the supplement material attached to this questionnaire. It contains a description of a problem statement that must be mapped using Scrum User Stories and implemented within a specific timeframe according to a difficulty level assigned to each User Story (described below). We ask you to read the material carefully, and address the key issues behind the problem, in a system design perspective, i.e., we are interested in inferring the necessary effort to implement a scientific system with several requirements from different knowledge areas. For this questionnaire, you are allowed to use notes and to write down any remarks that you consider important. We only ask to answer the questions without interference from other audiences (participants, etc.).

We have defined seven (7) *User Stories* (US) as follows. Using a *Fibonacci* Scale (e.g. 1,1,2,3,5,8,...), assign a value (column '*Value'*) for each US, breaking it down to *Development (D)* and *Test (T)* efforts required. *As a comparison basis, consider a US defined as a file reading, processing, and writing counts as 2 effort points.*

	User Story **(US) description**	**Value**
US1	***USERS*** should be able to set up input file containing simulation parameters such as simulation time, the total number of replications, output folder path, statistics capture, graphic options, and physical machine resources usage, ***SO*** they could analyze multiple simulation scenarios	***D*** ***T***
US2	***USERS*** should create sessions where simulations are run, and their output files are saved (assigned to this session), ***SO*** they can create analysis sessions to inspect different parameters tradeoffs	***D*** ***T***
US3	***USERS*** should configure simulations by time, some events or some stop criteria such as a variable reaching some value, which uses an input properties file containing simulation parameters and events rates, ***SO*** they can work with simulation sessions that benefit	***D*** ***T***

	multiple analysis		
US4	***USERS*** should erase simulation sessions (from the disk) and organize existing sessions by assigning names, date/time, and other related information, ***SO*** they can define multiple executions	***D***	***T***
US5	***USERS*** should know if the simulator will execute in a valid period according to the input file definition, where the simulator calculates a prediction for the simulation time, and present warnings to users case, those calculations are invalid, ***SO*** it is possible to change the input parameters and restart simulation execution	***D***	***T***
US6	***USERS*** should inspect generated simulation output with graphics that will show different metrics captured during the execution, ***SO*** they enhance the analysis phase with evidence of operation of the simulated parameters	***D***	***T***
US7	***USERS*** should combine different outputs from different executions into one graphic, ***SO*** that they can analyze the impact of choices more efficiently	***D***	***T***

6. Based on the problem statement, and the values chosen in the table, it is possible to infer that:

☐ These kinds of systems are best described using Scrum methodology

☐ It would be more comprehensive if described with RUP or other similar methodology

☐ The problem statement is poorly described, making it impossible to design and implement a system with these requirements

☐ Some fundamental User Stories are missing from the table, making it impossible to design and implement a software with such specification

☐ The use of a *Fibonacci Scale* has negatively influenced the time estimates for this project

☐ The test efforts are significantly variable for each User Story

☐ The test efforts are the same for each User Story

7. Supposing that you are the management of a big company with three different teams, separated according to the programming level: inexperienced, intermediary, and experienced. How do you think you could estimate the coding efforts for teams given such restrictions?

____

____

8. Do you believe that the estimated efforts vary according to the selected programming language (let's suppose that your team is proficient in many languages)? ☐ Yes ☐ No
9. To design this system, which fundamental knowledge is required for a smooth implementation? (*choose at least two*)

☐ OS – *Operating Systems*

☐ DES – *Discrete Event Simulation*

☐ OOP – *Object-Oriented Programming*

☐ A&P – *Algorithms and Programming*

☐ SDM – *Software Development Methodologies*

☐ Other: ______________________________

10. This type of scientific system would be best developed using an imperative language or an object oriented paradigm? ☐ Imperative language ☐ Object oriented language
11. In your opinion, how many SW developers are required to code in less than 3 months? ____
12. In your opinion, what should be the profile of professionals to code such systems?
13. Case your major stakeholder had chosen to consider *performance* as the key measure of quality, where he had a full case study describing considerable time information from a concurrent system, would this fact influence your decisions as to the time estimates and programming languages? ☐ Yes ☐ No

Questionnaire – Part B

14. Current occupation or position held in company: ___________________________________

15. Experience with Software Functional Testing *(requirements)*:
☐ None☐ Initial ☐ Intermediary ☐ Advanced

16. Experience with Software Automated Testing:
☐ None☐ Initial ☐ Intermediary ☐ Advanced

17. Number of hours dedicated to software development:
☐ Less than 2 ☐ Between 2 and 20 ☐ Between 20 and 40 ☐ More than 40

18. Number of hours dedicated to design/specification of software (using a methodology):
__________/____________

19. Do you work on personal software development projects (unrelated to your job)?
☐ Yes ☐ No

20. Date: _______/___________/__________