I, Nawar H. Obeidat, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:
The Design and Development Process for Hardware/Software Embedded Systems: Example Systems and Tutorials

Student’s name: Nawar H. Obeidat

This work and its defense approved by:

Committee chair: Carla Purdy, Ph.D.
Committee member: Raj Bhatnagar, Ph.D.
Committee member: George Purdy, Ph.D.
The Design and Development Process for Hardware/Software Embedded Systems: Example Systems and Tutorials

A Thesis submitted to the Graduate School of
The University of Cincinnati
In partial fulfillment of the requirements for the degree of

Master of Science

In the
Department of Electrical and Computer Engineering

Of the
College of Engineering and Applied Sciences

November 2014

By

Nawar Obeidat

Committee Chair: Dr. Carla Purdy
Today embedded systems are found in all areas of our lives and have many different applications. They differ in their uses and properties as well as employing both software and hardware components in their implementations. This has made the design and development process for them much more complicated. Learning to use such a process is especially difficult for electrical engineering students, who have not been introduced to the systematic design and testing methodologies familiar to students trained in computer science and computer engineering.

In this thesis, we illustrate the similarities and differences in the design and development design processes in for software systems and for software/hardware embedded systems. We give details for every stage for both types of systems and we develop detailed examples for example embedded systems, using a design process which extends the standard UML-based process used for software. In addition, we include details about project management. The examples and additional exercises and questions provide a set of tutorials which will assist students unfamiliar with complex design procedures in mastering the necessary skills to become well-trained embedded system developers.
To my Children, Husband, Mom, Dad, and all my family …
# Table of Contents

**CHAPTER 1: INTRODUCTION** ................................................................. 1

**CHAPTER 2: BACKGROUND** .............................................................. 4
  2.1 Embedded system areas ............................................................... 4
  2.2 Teaching embedded systems ...................................................... 8
  2.3 Thesis work .................................................................................. 10

**CHAPTER 3: SOFTWARE DESIGN PROCESS IMPROVEMENT** .......... 11
  3.1 Necessity for hardware/software co-design ................................. 11
  3.2 Software and software/hardware development activities ............. 12
    3.2.1 Waterfall and agile software development methods .............. 12
    3.2.2 Software and SW/HW design process development activities .... 14

**CHAPTER 4: DESIGN PROCESS WITH INTELLIGENT VENDING MACHINE EXAMPLE** ................................................................. 28
  4.1 Intelligent vending machine design process ............................... 28
  4.2 Implementation ............................................................................ 50
  4.3 Tutorial ......................................................................................... 53

**CHAPTER 5: CONCLUSION AND FUTURE WORK** .......................... 54

**REFERENCES** .................................................................................... 56

**APPENDIX A: THE DEVELOPMENT PROCESS FOR HARDWARE/SOFTWARE EMBEDDED SYSTEMS: EXAMPLE SYSTEMS AND TUTORIAL** .......................................................... 64
# LIST OF FIGURES

Figure 1: Waterfall and agile development methods [29] ...........................................13  
Figure 2: Typical use case diagram ..................................................................................17  
Figure 3: Weighted constraint chart for low-end and intelligent vending machines .............................................................................................................31  
Figure 4: Use case diagram for an intelligent vending machine ....................................33  
Figure 5: Class diagram for intelligent vending machine ...............................................42  
Figure 6: Deployment diagram for an intelligent vending machine ................................45  
Figure 7: Sequence diagram for an intelligent vending machine, related to customer actor actions ........................................................................................................46  
Figure 8: Sequence diagram for intelligent vending machine, related to employee actor actions ...........................................................................................................47  
Figure 9: Embedded system’s Gantt chart .........................................................................50  
Figure 10: Altera DE2 Board ..............................................................................................51  

Figure 1A: IVM use case diagram ....................................................................................67  
Figure 2A: IVM Sequence Diagram ..................................................................................71  
Figure 3A: IVM Class Diagram ........................................................................................73  
Figure 4A: IVM Deployment Diagram ...............................................................................76  
Figure 5A: IVM State Diagram ........................................................................................77  
Figure 6A: Embedded system’s Gantt chart .....................................................................79
LIST OF TABLES

Table 1: Functional requirements in low-end and intelligent vending machines .......................................................................................................................... 29
Table 2: Non-functional requirements in low-end and intelligent vending machines ...................................................................................................................... 30
Table 3: Intelligent vending machine Hardware/Software classes ................................................................. 43
Table 4: Development schedule for an embedded system .................................................................................. 49

Table 1A: HW/SW IVM classes ........................................................................................................................ 74
Table 2A: Development process schedule for an embedded system .............................................................. 78
CHAPTER 1: INTRODUCTION

Embedded systems are special purpose computing systems that perform a set of predefined tasks and have specific requirements and constraints [1]. Embedded systems shape our world. They exist everywhere around us in many different applications such as cellphones, energy generation and distribution, home appliances, automotive components like antilock brakes, and satellites [2]. Embedded systems are also found in more critical applications. For example, they are found in medical devices, aircraft, trains, special computers for nuclear power plants, and aerospace and defense systems [3].

The embedded systems market is increasing year by year, with an annual growth of 9% [2]. One of the common embedded systems that we see in many different public places is a vending machine. Recently, the vending machine market is growing, which made Intel Corporation develop enhanced solutions to improve on intelligent vending machines [4].

Embedded systems today are typically a mix of hardware and software components [5]. They are usually subject to practical constraints such as code-size, cost, weight, and energy [6]. This makes embedded systems development heavily dependent on sound engineering processes which enable them to meet the design
constraints of their surrounding systems [2]. This means embedded systems design and development are not easy and straightforward processes. It often takes a lot of work and determination for a given embedded system’s development team to meet all the system’s requirements. Because of this, there is increased effort going into developing better design processes for embedded systems in industry as well as into developing more effective teaching techniques for embedded systems classes in universities.

This thesis provides a tutorial on hardware-software co-design for use in a senior/beginning graduate level class at the University of Cincinnati. This tutorial explains a hardware/software development process based on UML [7] and design for testability [6] which can be mastered by students in the class and applied by them to develop their semester projects. The implementation will be demonstrated by running C code on an Altera DE2 board [8] for an intelligent vending machine example. This helps students to be able to study the steps in the method and see how they are carried out for this project. They will then be in a position to apply the same steps to their own projects.

This thesis is organized as follows. Chapter 1 provides an introduction to the thesis. Chapter 2 includes background on embedded systems and on teaching embedded systems, and includes information on projects related to the thesis work. Chapter 3 defines the software design process and shows how it is extended for
software/hardware embedded systems. It also explains the necessity for hardware/software co-design in embedded systems. Chapter 4 shows the result of applying the process to develop a full detailed example of the intelligent vending machine design process, along with an implementation. It also summarizes the tutorial which is included in Appendix A of this thesis. Finally, chapter 5 gives the conclusion and describes possible future work based on this thesis.
CHAPTER 2: BACKGROUND

There are many different design processes which have been employed to develop embedded systems. Approaches include, for example, UML, state charts, Petri nets, and tools based on discrete event simulation [6], and enhanced versions of UML, such as MARTE [9] [10]. The large number of design processes is partly due to the great variety of embedded systems, which range from small simple systems to complex systems with multiple interacting processes. But there are still many embedded systems projects which do not follow a well-documented design process. Embedded systems students need to learn the basic properties of a robust process and to have such a process to use in their projects. This will support them in learning the important properties of a good development process and applying it in projects in the workplace. A good process will also produce a system which meets industry requirements. The goal of this thesis is to define a simple, efficient, and effective design process which can be used to develop a wide range of embedded systems.

2.1 Embedded system areas

To implement an embedded system, the developer needs to have a good knowledge of many different areas because embedded systems typically include both
hardware and software components. Embedded systems courses must cover all the major topics that students need to understand and apply to produce a successful embedded system project by the end of the course.

The major topics that students need to learn are:

- General familiarity with embedded systems and their properties: students need to know what constitutes an embedded system. They need to examine different examples of embedded systems, and understand typical embedded system characteristics. They need to know, for example, that embedded systems have a combination of hardware/software components that work together to complete a specific task in a reactive and proactive environment, that usually hardware is used for performance and software is used for supporting features and flexibility, and that embedded systems have real time constraints and typically must be low cost, manage power consumption efficiently, and be reliable and fault-tolerant [11].

- Instructions set architectures: these include, for example, the ARM and the SHARC architectures [12], PIC [13] and Intel 8051 [14] microcontrollers. Students need to be aware as well of the basic difference between von Neumann and Harvard architectures and why this is important [15].

- Processors: these range from older 16-bit processors which are low cost and may require an assembly language to newer 32-bit or 64-bit processors [16].
In addition, there are soft-core processors such as the Altera Nios II [17] and the Xilinx MicroBlaze [18].

- Memories and the memory subsystem: students need to learn about the memory hierarchy, caching, and ROM. They need to learn why caching may be inappropriate for an embedded system [19].

- C programming: Each embedded system has its own function that is different from other embedded systems. This means that every embedded system has its own software programs. But there is a common feature among all embedded systems in that most are using the C programming language [19]. Students also need to know why recursion and dynamic storage are not usually good choices for a program in an embedded system [20].

- Operating systems: Students need to understand operating system components and duties, processes, threads, and scheduling, resource sharing mechanisms and deadlock. They also must know the basics of real-time operating systems. They need to know the difference between “hard” and “soft” deadlines. For example, a response to missed deadlines which is crucially important in some systems like an airplane’s flight system is called a “hard” deadline, while other systems such as a satellite communication system may have “soft” deadlines where, if the deadline is missed, damage is limited and not crucial [19].
• General embedded system constraints: these include, for example, power, efficiency, cost, etc.

• System development process: students should learn the general waterfall process [7], with the stages of:
  1. Analysis: Requirements and Specifications
  2. Design (UML)
  3. Implementation
  4. Testing
  5. Maintenance

• Agile processes [7], and DFT or design for testability [21]. We will include more details about these in chapter 3.

Embedded systems students must also learn practical aspects by completing all lab assignments in a course, including a small project which follows the defined development process. At the end, students must:

• Have the ability to make a good selection for embedded system platform, define system requirements and determine hardware/software components.

• Realize and analyze a hardware/software implementation.

• Overcome performance problems with the hardware implementation [12].
2.2 Teaching embedded systems

Because embedded systems are appearing in more and more products, universities are giving more attention to teaching embedded systems classes for engineers and computer scientists.

Unfortunately, some universities teach this class at a low level, covering, for example, teaching how to program in assembly language using an 8-bit processor with a very simple interface and experiments [16].

Some universities focus on teaching hardware design for embedded systems rather than also including software design for engineering students. For example, the basic University of Colorado course focuses on hardware and firmware design. Students learn how to select an appropriate processor for a given application, as well how to program processors in the Intel 8051 microcontroller family, along with details about the 8051 architecture and instruction set. In addition, they learn about hardware/firmware partitioning, glue logic, circuit design, layout, and debugging, development tools, firmware architecture, design, and debugging [22].

An embedded systems class can’t be just a software design class because it has to include hardware components that represent the system. At the end the embedded system is a machine that performs a special task and uses specific I/O devices to complete this task, not just a software program. This requires universities to focus
on both hardware and software components to prepare students and provide them with a good general understanding of embedded systems industry needs.

Because modern real-time embedded systems are based on both hardware and software and often perform demanding tasks, both data-intensive and control-oriented, they often handle such activities as multitasking, concurrency, and interrupts, and thus they need both user and supervisor modes of operation. Other important features typically include, for example, mobility, with wireless capabilities and long lasting low cost battery power, which means designs must be “power-aware" [16] [23].

The term hardware/software co-design is showing up more and more in embedded systems syllabuses in different universities. MIT, for example, focuses on teaching hardware/software co-design for in its basic embedded systems class [24].

There are some difficulties in teaching embedded systems students, especially electrical engineering students. One of the problems in teaching embedded system design is the equipment needed to make the topic practical and interesting. In addition, as mentioned above, today’s embedded systems are becoming very complex, which makes them difficult to teach [6]. In addition, an embedded systems class must cover many topics in a short time, which makes it difficult to go into details. And typically electrical engineering students have very little
experience in software and in the software design process. So they can’t complete the design steps quickly and easily, they need repetition and practice.

2.3 Thesis work

The design process for embedded systems is a very important process that gives students the ability to develop an embedded system from scratch. They must start by collecting requirements from a customer and eventually complete the task of developing the embedded system needed. That’s why teaching the students the design process is an important topic in this class.

This thesis explains how the software development process can be extended for embedded systems by providing a full tutorial for electrical engineering students to have a complete conception and experience with a design process for embedded systems, with full design examples and exercises using UML diagrams. It provides a step-by-step guide that students can follow as they develop their own class projects. Following the steps outlined in the tutorial will save students time in each stage in learning the design process from scratch, and help them to improve their knowledge in this area. If they go through the tutorials, then they will be able to gain good experience in the embedded systems design process in a short time.
CHAPTER 3: SOFTWARE DESIGN PROCESS IMPROVEMENT

The engineering design process in general is a plan with many stages that helps the engineer to perform specific tasks that result in a product that meets specifications. These stages could be repeated many times before starting the final task [25].

This chapter shows the detailed software design process and how it can be modified for software/hardware embedded systems. It includes details for each development activity. This chapter also illustrates the weighted constraint tables and charts which will be used in the next chapter.

3.1 Necessity for hardware/software co-design

Why is it important to use a hardware/software co-design process in designing embedded systems?

- Almost all embedded systems have both HW and SW components
- Embedded systems perform special tasks that need integration of hardware components with software modules
- Combining hardware and software enhances the embedded system’s performance
• This decreases the cost; hardware is costly and software is lower cost [26].

3.2 Software and software/hardware development activities

This section describes the detailed software design process described by Bruegge and Dutoit [7] and Koopman [27], along with extensions to enable the design of hardware/software embedded systems as described by Marwedel [6] and Wolf [28]. In addition, a basic comparison between two widely used software development methods is provided.

3.2.1 Waterfall and agile software development methods

Figure 1 shows one way of comparing one design cycle for the waterfall and agile development methods. In a traditional waterfall development process, the entire product is designed in one cycle. In an agile process, the product is developed over several cycles, each of which makes a noticeable but possibly small change in what will be the final product.
The waterfall method is a fixed method that supports a sequential design process where development flows sequentially from the starting point to the ending point. Agile methods are more flexible because they support an incremental and iterative approach to software design. In agile methods, the designers have the flexibility to respond to requirement changes as they arise during the course of the product development [29].

The waterfall design method is used in embedded systems because of its simplicity, and because hardware components are more difficult to redesign. In the waterfall method, as well, the designers can set up a fixed plan so they can launch
software fairly quickly, and this helps in developing, since it gives more accurate estimations for timetables and budgets [29].

In comparison, an agile method is very flexible, which permits changes to be made after the original plan has been determined. That makes an agile method very valuable in situations where the end-goals of a project are not defined clearly. One popular usage for agile methods is in products such as cell phones that frequently have new versions [29].

### 3.2.2 Software and SW/HW design process development activities

Software and SW/HW design process development activities include:

1. **Requirements**

In software, requirements describe everything that the software should or shouldn’t do in addition to the constraints that it is supposed to meet. These requirements should be measurable, accurate, and traceable through the software development process.

Writing requirements is a basic step that prevents you from omitting important features, having a wrong idea of what the system should do, or creating a system that doesn’t work correctly.
There are several types of requirements:

- **Functional requirements**
  These represent the interactions between the system and any other external systems or users independent of its implementation.

- **Non-functional requirements**
  These represent the system’s features that are not directly related to the system’s behavior. They may include, for example, cost, power consumption, system usability, and performance.

- **Constraints**
  These represent the restrictions in building the system. For example, the system may need to meet national or international standards. For example, the system might need to conform to the requirements of IEC 61508 for ISL software. This means that the system is subject to constraints because some of its tasks are safety critical [27].

We can use non-traditional requirements to determine the weighted constraint chart [30] for an embedded system. First we have to weight these non-traditional requirements by giving them different weights depending on how crucial they are.
Weights must add up to a total of 100. So the more crucial non-traditional requirement will get the highest weight. Then the weighted constraint chart could be drawn depending on these weights which could vary for the same system depending on the features that are to be optimized [30]. This chart will be done during the design process example in the next chapter.

To achieve a systematic design process, we will use UML (the “Unified Modeling Language” [10]). UML is a complex graphical language that was originally developed to support the software design process and that is supported by commercial tools. UML supports many features, such as use case diagrams, sequence diagrams, class diagrams, and deployment diagrams, to enable designers to develop and modify their work.

To document specifications, UML provides use cases and use case diagrams. After collecting requirements from customers, a developer should translate the results that describe the system into a use case diagram. A use case diagram has actors and the specific use cases. The actors represent the external objects that interact directly with the system. These may include end users, environment, and computers that the system connects with, for example. Each use case represents some system functionality, i.e., the sequence of interactions between the system and one or more actors.
Figure 2 shows a typical use case diagram.

In HW/SW embedded systems, collecting requirements through informal language is not enough because this may not include key requirements which must be included in the system specification. The specifications should be complete, have no conflicts, and enable the designers to reach the implementation stage in a systemic way. Ideally the specifications should be in a formal language and be machine readable and checkable [6]. But in practice many embedded systems projects do not yet use formal specification. The use case approach we will use
here is not a formal specification technique, but it can easily be learned and used by students and it is a commonly used technique in embedded systems projects [6].

In the software design process, UML language is important and effective. But the commonly used UML version 1.4 was not ideal for use with embedded systems because it lacked a number of features that are important in modeling embedded systems. It has class diagrams, use case diagrams, activity diagrams, state-chart diagrams, sequence diagrams, collaboration diagrams, component diagrams, and deployment diagrams [6] [31]. UML 2.0 was enhanced to include 13 diagrams to better support embedded systems modeling. Added diagram types include communication diagrams, composite structure diagrams, timing diagrams, package diagrams, and Interaction Overview Diagrams [6] [32].

2. Analysis

In the software design process, the analysis is the second stage that formalizes the system requirements and gives specifications, i.e., an analysis model for the system, that should be complete, correct, consistent, and verifiable. This model may not be readable for users and customers, but it should be for developers who make it to discover requirement errors, update them to reflect insights gained, and finally discuss the changes with users and customer.

This analysis model consists of three separate models:
• The functional model: this is represented by use cases and use cases scenarios.

• The analysis object model: this is represented by class diagrams and object diagrams.

• The dynamic model: this is represented by state machine diagrams and sequence diagrams.

3. Design

In the software design process, the design stage represents the code structure without writing the code itself. Code could be represented by flowcharts, pseudocode, and state-charts. Also some UML diagrams, such as sequence, collaboration, and activity diagrams, are useful here. Design doesn’t have any implementation elements, it is not allowed to have executable code. Design is the stage where the developers organize their thoughts and identify any problems before writing the code.

In software there are two types of design, system and object design. These two types of design as well as implementation represent the structure of the systems where the developers connect the requirements and analysis, with the system which the users use at the end.
System design is the first stage, where the system is structured by decomposing it into parts. The challenge is to meet all constraints and resolve conflicting criteria during this decomposition. System design has information about the internal system’s structure and its hardware configuration because it focuses on data structures, processes, and the division into software and hardware components in the implementation stage. System design enables the developers to determine:

- What the design should be to optimize the system’s required and desirable qualities
- What the overall hardware / software architecture should be and how subsystems can be mapped into hardware components
- How the data should be stored, how control flow will be managed, and how access control will be defined, as well as what each subsystem should do
- What the system’s boundaries represented by use cases will be
- How the system starts and shuts down as well as the overall system configuration and its exception cases.
The second design type is called object design. Object design comes after system design to minimize the gap between any off-the-shelf components and application objects. Object design includes:

- Reuse of any off the shelf components which were identified in the system design stage to provide realization of subsystems. This helps in using existing solutions
- Interface specification that accurately describes each class and/or component interface
- Object model restructuring and optimization which represents transforming the object design model to increase its extensibility, understandability, and ability to meet performance criteria

The design stage in software/hardware co-design is usually called “architecture design” [28]. Previous stages show what the system does without providing information about how the system does things. The architecture stage shows how the system implements its functions, as well as the overall system structure and the plan for designing the system components. So the architecture stage defines all components needed, along with the design effort needed to build these components. These components include software components as well as hardware
components such as field programmable gate arrays (FPGAs) and specific I/O devices.

UML is a very efficient visual language that can be used for capturing all design tasks. UML is very efficient because of the ability to add details to the design so there is no need to make a new design at every level of abstraction.

UML is also an object oriented modeling language that has two important concepts:

- The design is represented as a set of interacting objects, not like a set of blocks of code

- The design guarantees that these objects will be a piece of software or hardware in the system.

UML also provides the ability to include the outside world in the design and to show how the system and objects interact with users or other machines. All this helps in better understanding the natural structure of the system.

4. Implementation

In the software design process, the implementation stage consists of translating the solution domain model into code. Implementation includes implementing all system attributes and the methods for each object, then integrating all these objects
together. Implementation maps UML models to code. In hardware/software co-
design, some modules will become software code, while some will become 
hardware components.

If all work was done correctly in the design process activities, most design issues 
should have been solved by this stage. The system can then be implemented 
depending on the predesigned use cases from the requirements and the design 
activities so far. The most common problem that developers can face in this stage 
is the system’s integration, which means putting all subsystems together. This 
could require the developers to write more code modules to prevent system 
degradation which may become evident during the integration step. A number of 
transformations can be applied at this step to achieve optimal integration [7]. 
These include:

- Class model optimization
- Associations to collections mapping
- Operation contracts to exceptions mapping
- Class model to a storage schema mapping

In embedded systems we must do software/hardware co-design. The challenge is to 
find a perfect combination between software and hardware components so that the 
system meets all specifications. Embedded systems are complex and their
requirements are stringent with respect to time-to-market, which makes reuse of hardware and software components important. This reuse process is called “platform based design” [6]. A platform is a set of architectures with a set of constraints to allow reusing software and hardware components. This can enable a reliable, quick, and derivative design using a platform application programming interface (API).

5. Testing

Testing usually includes a fault detection plan that tries to find faults and create failures and then repair them in order to ensure a fault free system before it is released to the customer. One definition of successful testing is that it identifies faults. Another definition is “it demonstrates that faults are not present.” [7].

Testing should be integrated throughout the system development process. Unit testing is used to find differences between a specification of an object and its components. Structural testing is used to find differences between the system design model and the integrated subsystems. Functional testing is used to find differences between the system and the use case model. Finally, performance testing is used to find differences between nonfunctional requirements and the actual system performance. When these differences are found, developers identify the weakness causing the observed failure and modify the system to correct it [7].
Testing in embedded systems can be partitioned into “verification and validation”. Validation is the process of checking if a certain (possibly partial) design is proper for its purpose and meets all constraints and will work as expected. Validation is important for any design procedure, and it’s very important for safety-critical embedded systems. Validation is required at many phases during the design procedure [6]. Verification ensures that design steps are being followed correctly, and roughly corresponds to the notion of backward traceability [27].

To develop more efficient and faster algorithms to generate test patterns or use design techniques to enhance testability especially for embedded systems, there is a technique called "design for testability." Design for testability techniques offer one approach to improving testing by reducing the complexity of testing [33]. Design for testability keeps testability in mind by adding additional components to ease testing the system. Design for testability ensures that we have ways to validate the system’s behavior without any doubt [30].

For a component-based software system, to make sure that we can run the system properly and effectively, the qualities of basic components have to be assured. Third-party certification is a safe approach to trust when dealing with component software [34]. Third-party testing means the voluntary, contractually needed, or statutorily mandated intervention of an independent testing laboratory for testing the component using known standards, rendering a finding to the requestor [35].
Testing includes many stages, styles, and activities:

- **Test planning.** This includes allocating resources and scheduling the testing. It should be done early in development to have enough time for testing. Independent testers should start testing as soon as the developers state that their models are stable.

- **Usability testing.** This represents finding faults in the user interface design. Sometimes an unfriendly user interface design can cause a system failure.

- **Unit testing.** This includes finding faults in subsystems and participating objects.

- **Integration testing.** This involves finding faults in each subsystem while integrating it with other subsystems. Structural testing includes integration testing for all the system’s components and their interactions.

- **System testing.** This means testing the whole system and its functions with respect to the system’s requirements, design, and goals. It includes:
  - Functional testing that tests that requirements are met
  - Performance testing that tests that nonfunctional requirements and additional design goals are met
  - Acceptance testing that tests the system against the project agreement which was negotiated with the customer.
• Exploratory testing. This includes checking the function of the system. It is done by using system operating context, experience, and judgment to find faults.

• Black box testing. It means testing the behavior of components or units as they interface with other components, without “looking inside” the components themselves.

• White box testing. It means checking the internal behavior of systems components.

Testing should be done as well by measuring system performance parameters and finding out the correlation between the results and these parameters’ weights [30].

Chapter 4 will include an intelligent vending machine example that shows the applicable work for all that has been mentioned above in this chapter.
CHAPTER 4: DESIGN PROCESS WITH INTELLIGENT VENDING MACHINE EXAMPLE

Chapter 3 illustrated the development process for both software and software/hardware, and chapter 2 explained why teaching this process is important in embedded systems design courses for university students.

This chapter will show a full detailed example for an “intelligent vending machine” (IVM). It will show the complete design process along with an implementation. It will also be reformulated as a tutorial that helps embedded system students gain a full understanding of the HW/SW co-design development process.

4.1 Intelligent vending machine design process

The vending machine market is increasing and it includes approximately 18.7 million installed units in 2010 [36]. The first priority for intelligent vending machines is to have an LCD screen to interact with users [36]. The new intelligent machines have other new features such as LCD screens with a touch screen interface, cashless payments option, highly secure transactions, anti-vandal anti-theft devices or cameras, and remote management. Improving energy efficiency is also a factor that is becoming important in the intelligent vending machine market [37].
An intelligent vending machine (IVM) is a good example of an embedded system. This section will give a detailed description of the development process for a “standard” vending machine (VM) and for an IVM. These examples will show how the ideas of the previous chapter can be applied in specific cases. UML diagrams will be used to describe the systematic stages in the design process.

Tables 1 and 2, along with figure 3, compare the low-end VM and the IVM in terms of functional and non-functional requirements and give the weighted constraint graph that represents these comparison quantitatively. Weights in each case total 100 [30].

Table 1 shows some of the functional requirements for intelligent and low-end vending machines.

<table>
<thead>
<tr>
<th>Functional Requirement</th>
<th>Low-end VM</th>
<th>Intelligent VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet connected</td>
<td>Not connected</td>
<td>Connected</td>
</tr>
<tr>
<td>Cashless system</td>
<td>No; Cash only</td>
<td>Yes; Payments by cash, mobiles, credit/debit cards</td>
</tr>
<tr>
<td>Advanced LCD</td>
<td>Does not exist</td>
<td>Exists, with different advertisements</td>
</tr>
<tr>
<td>Has both food/drink</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>System control</td>
<td>Low control</td>
<td>High system control</td>
</tr>
</tbody>
</table>

Table 1: Functional requirements in low-end and intelligent vending machines
Table 2 gives non-functional (NF) requirements for intelligent and low-end vending machines.

<table>
<thead>
<tr>
<th>NF Requirement</th>
<th>Low-end VM</th>
<th>Intelligent VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing speed</td>
<td>Fair</td>
<td>High speed</td>
</tr>
<tr>
<td>Power Consumption</td>
<td>High</td>
<td>Less</td>
</tr>
<tr>
<td>Security</td>
<td>Less secure</td>
<td>High level security</td>
</tr>
<tr>
<td>Weight</td>
<td>Less</td>
<td>Heavy</td>
</tr>
<tr>
<td>Cost</td>
<td>Low cost</td>
<td>More expensive</td>
</tr>
<tr>
<td>Touch screen</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Remote management</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2: Non-functional requirements in low-end and intelligent vending machines

Figure 3 shows the weighted constraint chart [30] for low-end and intelligent vending machines. For each machine, the weights for the different properties add up to 100. As shown in figure 3, the most important requirements for the intelligent vending machine are security, processing speed, and power consumption, while the most important requirements for the low-end vending machine are: low cost and light weight.
Before starting designing by UML modules, we’ll need to have an idea of why we use those modules in designing different embedded systems.

UML has many benefits for use with embedded systems. Some of them are:

- UML includes a set of notations, semantics, and syntax to allow the creation of families of designs for certain applications.
- Use case modeling describes system environments, user interactions, and test cases.
- Extension mechanisms like constraints can be used for certain applications.
- UML supports object-oriented system specification, design and modelling.
- UML supports real-time systems [38].
As mentioned in the previous chapter, UML diagrams like use case diagram are used in the first stage of the design process. These are used in requirements and analysis activities.

The use case diagram is the simplest representation of how the user/s interact with the system and describes the specifications of a use case [39]. Use case diagrams have use cases that represent the system functionalities, actors that represents entities that interact with the system, and the connections between them [40].

Figure 4 shows the use case diagram for the intelligent vending machine, followed by the flow of events and the system level test cases.
Figure 4: Use case diagram for an intelligent vending machine

Flow of events for use cases:

Participating actors: Customer, Manager, and Employee.
Use cases:

- **Name: Select product**

  **Participating actors:** Customer.

  **Flow of events:**

  1. A message will appear on the keypad LCD: “Please select a product”.
  2. Customer hits a button from the keypad related to his selection.
  3. A product advertisement or a related products advertisement will appear on the Adv. LCD.
  4. Processor checks the product availability.
  5. Processor updates the database and sends the product price.
  6. The product price will be displayed in the keypad LCD.

  **Entry condition:** Customer hits a button from the keypad related to his selection.

  **Exit condition:** A message will appear on the keypad LCD “Please select payment option: 1.Cash 2.Credit/Debit Card”.

  **Quality requirements:** Speed of response to all actions, nicely behaved buttons, and high resolution LCDs.
- **Name**: Pay for product, Select payment method, and Insert cash OR slide credit/debit card

*Participating actors:* Customer.

*Flow of events:*

1. A customer hits “1”.
2. A price will be verified from coin sensor. If he wants to pay cash, or “2” if he wants to pay by credit/debit.
3. If the customer hits “1” a message will appear on the keypad LCD “Please insert coins”.
4. The processor will update the database and send the product to the customer.
5. The advertisement on the Adv. LCD will be updated.
6. A message will appear on the keypad LCD “Please select a product”.
7. If the customer hits “2” a message will appear on the keypad LCD “Please slide your card”.
8. The cashless sensor will sense the card info. If the card is a debit card a message will appear on the keypad LCD “Please insert a 4 digit code for your card”.

9. The cashless sensor will send all information to Ethernet for verification.

10. The Ethernet will verify the information and send ACK to the processor.

11. The processor will update the database and send the product to the customer.

12. The advertisement on the Adv. LCD will be updated.

13. A message will appear on the keypad LCD “Please select a product”.

Entry condition: A customer hits “1” if he want to pay cash, or “2” if he want to pay credit/debit on the keypad buttons.

Exit condition: A message will appear on the keypad LCD “Please select a product.

Quality requirements: Fast information verification by Ethernet, fast response to all actions, nicely behaved buttons.

- Name: Collect cash

  Participating actor: Employee.

  Flow of events:
1. The employee presses the employee button (which is selected by the owner company, for example: F1).

2. A message will appear on the keypad LCD “Please insert Employee ID”.

3. The employee will insert his ID from the keypad buttons.

4. The processor will verify the ID and open all locks.

5. The employee will collect cash and lock the machine.

**Entry condition:** The employee presses the employee button.

**Exit condition:** The employee locks the machine.

**Quality requirements:** Fast verification for EmpID, and fast unlock/lock for the locks

- **Name:** Restock products

  **Participating actor:** Employee.

  **Flow of events:**

  1. The employee presses the employee button (which is selected by the owner company, for example: F1).

  2. A message will appear on the keypad LCD “Please insert Employee ID”.

37
3. The employee will insert his ID from the keypad buttons.
4. The processor will verify the ID and open all locks.
5. The employee will restock the products and lock the machine.

*Entry condition:* The employee presses the employee button.

*Exit condition:* The employee locks the machine.

*Quality requirements:* Fast verification for EmpID, and fast unlock/lock for the locks.

**Acceptance Tests Developed During Analysis and Specification:**

- **Select product:**

1. When customer didn’t press any button from keypad, a message “Please select a product” should appear on the keypad LCD.
2. When customer presses a button from the keypad to select a product, a new advertisement will appear on the Adv. LCD, a price will be shown on the keypad LCD, and a message will appear on the keypad LCD “Please select payment option: 1.Cash 2.Credit/Debit Card”.
3. (Quality) when customer presses any button, the system should immediately respond.
• Pay for product, Select payment method, and Insert cash OR slide credit/debit card:

1. When customer hits “1” from keypad, a message should appear on the keypad LCD “Please insert coins”.

2. When customer gets the product, a new advertisement should appear on the Adv. LCD, and a message should appear on the keypad LCD “Please select a product”.

3. When customer hits “2” from keypad, a message should appear on the keypad LCD “Please slide your card”.

4. If customer slides a debit card, a message should appear on the keypad LCD “Please insert a 4 digit code for your card”.

5. When customer inserts a 4 digit code for the debit card, the system should send the product to the customer.

6. When customer gets a product, a new advertisement should appear on the Adv. LCD, and a message should appear on the keypad LCD “Please select a product”.

7. (Quality) the system should immediately respond to the customer insertions from keypad.
8. (Quality) the system should verify payment information very quickly, to send the product quickly to customer.

9. (Quality) the system should refresh the advertisements on the Adv. LCD, depending on the customer selection, immediately.

• **Collect cash:**

  1. When the employee presses the employee button, a message should appear on the keypad LCD “Please insert Employee ID”.
  
  2. When customer inserts the ID from the keypad, the system should open all locks.
  
  3. (Quality) the system should open immediately after the employee inserts the ID.

• **Restock products:**

  1. When the employee presses the employee button, a message should appear on the keypad LCD “Please insert Employee ID”.
  
  2. When customer inserts the ID from the keypad, the system should open all locks.
  
  3. (Quality) the system should open immediately after the employee inserts the ID.
Another important diagram in UML is the class diagram that models the static view of the application. It represents the structure of the system by showing the system’s classes and attributes, operations/methods, and the relationships between objects [41] [42].

The purpose of the class diagram is to:

- Design and analyze the static view of the application.
- Define responsibilities of a system.
- Be a base to design component and deployment diagrams.
- Support both reverse and forward engineering [42].

Figure 5 represents the object model which is the class diagram for the intelligent vending machine.
In Figure 5 we have both the hardware and software classes. Table 3 shows whether each class is a hardware and/or a software class and why these choices were made.
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Hardware</th>
<th>Software</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Yes</td>
<td>Yes</td>
<td>Processor is the hardware part that controls all software parts</td>
</tr>
<tr>
<td>Cashless sensor</td>
<td>Yes</td>
<td>No</td>
<td>Hardware part that isn’t responsible for running software functions</td>
</tr>
<tr>
<td>Coins sensor</td>
<td>Yes</td>
<td>No</td>
<td>Hardware part that isn’t responsible for running software functions</td>
</tr>
<tr>
<td>Keypad</td>
<td>Yes</td>
<td>No</td>
<td>It’s a hardware part that has good connections with software classes</td>
</tr>
<tr>
<td>Ethernet</td>
<td>Yes</td>
<td>No</td>
<td>Hardware part that isn’t responsible for running software functions</td>
</tr>
<tr>
<td>Adv. LCD</td>
<td>Yes</td>
<td>No</td>
<td>It’s a hardware part that has good connections with software classes</td>
</tr>
<tr>
<td>Product</td>
<td>Yes</td>
<td>No</td>
<td>Hardware that represents the product itself</td>
</tr>
<tr>
<td>EmpIDSys</td>
<td>No</td>
<td>Yes</td>
<td>Software only; it has a database for all employee IDs</td>
</tr>
<tr>
<td>PwrMngt</td>
<td>No</td>
<td>Yes</td>
<td>Software only that receives data from hardware sensors</td>
</tr>
<tr>
<td>Coin_Controller</td>
<td>No</td>
<td>Yes</td>
<td>Software module that controls coins sensor</td>
</tr>
<tr>
<td>Security_Mngr</td>
<td>No</td>
<td>Yes</td>
<td>Software module that is responsible for keeping the systems secure, it needs to be updated regularly</td>
</tr>
<tr>
<td>Testing_HW</td>
<td>No</td>
<td>Yes</td>
<td>Software module that is responsible to keep testing all HW components</td>
</tr>
<tr>
<td>Cashless_Controller</td>
<td>No</td>
<td>Yes</td>
<td>Software module that controls cashless sensor</td>
</tr>
<tr>
<td>Product_Controller</td>
<td>No</td>
<td>Yes</td>
<td>Software module that controls products</td>
</tr>
<tr>
<td>Controller</td>
<td>No</td>
<td>Yes</td>
<td>Software module that controls all system’s components</td>
</tr>
</tbody>
</table>

Table 3: Intelligent vending machine Hardware/Software classes
Deployment diagrams are used to describe the system’s hardware components. They are used for visualizing the physical components’ topology. That’s why deployment diagrams are useful for describing the system’s static deployment view. Deployment diagrams include nodes and the connections among these nodes [43].

Most UML diagrams are designed to focus on software components of the system but a few diagrams like deployment diagrams, which are designed to focus on both software and hardware components, make it a good model in embedded systems. The importance uses of the deployment diagrams are:

- Visualizing system’s hardware topology
- Describing system’s hardware components that are used to deploy software components.
- Describing nodes’ runtime processing [43]

Figure 6 shows the deployment diagram for the intelligent vending machine, including the main hardware components.
A sequence diagram represents the time view of interactions among the system’s objects to achieve a behavior goal of the system. It describes classes and objects in a specific scenario and the sequence of the exchanged messages between the objects to give the scenario’s functionality. It’s related to the use case realization of the system [44] [45].
Figure 7 shows the sequence diagram for the select product and pay for product use case for the intelligent vending machine.

![Sequence Diagram](image)

Figure 7: Sequence diagram for an intelligent vending machine, related to customer actor actions.

Figure 8 shows the sequence diagram for employee actor action for the intelligent vending machine.
As a result of our analysis, we can now decide what components we can use to implement each kind of the vending machines.

For the low-end vending machine we can decide to have:

- A simple PIC microcontroller or FPGA
- Simple software or hardware only
- No OS, just a FSM

While we can design the intelligent vending machine using:

- Enhanced microcontroller
➢ More complex software (especially for security and controllability)

➢ Security software which should be up-to-date and will need frequent updating

➢ Power management software

In every project in industry or in any class in the universities there are “deadlines” to complete tasks. Writing a development plan is an important part in completing the system and meeting deadlines. Table 4 shows a general development plan schedule example using the waterfall model for an embedded system, while figure 9 shows a Gantt chart for this schedule.

A Gantt chart is a chart that is most used in project management to represent activities (tasks) against time in a useful and simple way. The left side of the chart represents a list of all tasks. Every task is represented by a bar with a particular position for a start date, and a particular length for time duration to complete the task [46].
<table>
<thead>
<tr>
<th>Task</th>
<th>Start date</th>
<th>Duration (Days)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Define Requirements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Define customer needs</td>
<td>1-Jan</td>
<td>1</td>
</tr>
<tr>
<td>Define available resources</td>
<td>2-Jan</td>
<td>1</td>
</tr>
<tr>
<td>Create req. Documents and use cases</td>
<td>3-Jan</td>
<td>3</td>
</tr>
<tr>
<td>*Milestone: Requirements defined</td>
<td>6-Jan</td>
<td>0</td>
</tr>
<tr>
<td><strong>2. Create UML description</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create use case diagrams</td>
<td>7-Jan</td>
<td>1</td>
</tr>
<tr>
<td>Create class diagram</td>
<td>8-Jan</td>
<td>1</td>
</tr>
<tr>
<td>Create deployment diagram</td>
<td>9-Jan</td>
<td>1</td>
</tr>
<tr>
<td>Create sequence diagrams</td>
<td>10-Jan</td>
<td>1</td>
</tr>
<tr>
<td>Revise UML diagrams. Complete test cases</td>
<td>11-Jan</td>
<td>3</td>
</tr>
<tr>
<td>*Milestone: UML Description completed</td>
<td>14-Jan</td>
<td>0</td>
</tr>
<tr>
<td><strong>3. Implementation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start coding. Start testing</td>
<td>15-Jan</td>
<td>8</td>
</tr>
<tr>
<td>Build up all components together</td>
<td>24-Jan</td>
<td>5</td>
</tr>
<tr>
<td>Test after integration</td>
<td>30-Jan</td>
<td>2</td>
</tr>
<tr>
<td>*Milestone: Implementation completed</td>
<td>1-Feb</td>
<td>0</td>
</tr>
<tr>
<td><strong>4. Testing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Define a test plan</td>
<td>2-Feb</td>
<td>1</td>
</tr>
<tr>
<td>Test the whole system</td>
<td>3-Feb</td>
<td>3</td>
</tr>
<tr>
<td>Complete testing. Write final report</td>
<td>7-Feb</td>
<td>5</td>
</tr>
<tr>
<td>*Milestone: System is ready to use.</td>
<td>12-Feb</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4: Development schedule for an embedded system
4.2 Implementation

The implementation we choose for our basic vending machine is the Altera DE2 Board and NIOS II Processor.

Figure 10 shows the Altera DE2 board [8].
In this traditional vending machine, we will use the DE2 board switches to model coin insertion. The number we will add via these switches will be the coins’ amount. We will use the LCD to display system messages such as the price and other messages. And finally, we will use the push buttons to determine the product that we need (as a total of four different products).

We need to determine the system’s components first. In our design we have these components:

*Input:* Switches (To insert price amount/coins). A hardware component.
Input: Push buttons (To choose a product). A hardware component.

Output: LCD Screen (Displays messages on screen). A hardware component.

VM Controller: Implemented using software to control the system. A software component.

Most of the modules for actually implementing the vending machine have already been developed for a similar project completed in the University of Cincinnati's course EECE 6017C, Embedded Systems, in Fall 2013. The one exception is the module to send and receive Bluetooth signals to enable wireless communication and also to provide a high level of security for transactions. Implementing this module and the related security algorithms is the subject of a separate University of Cincinnati M.S. thesis project, which is being carried out in parallel to with this project. A prototype of a similar system has been completed by Silva et al. [47].

For the intelligent vending machine, we can develop a more complex Controller which will include security modules and we can use the DE2 board’s USB connector to implement communication via Bluetooth, to provide remote communication.
4.3 Tutorial

This thesis includes a tutorial as part of the thesis work. It is designed to enable embedded systems students to get a better understanding of the design and development process. The tutorial includes a detailed example of the intelligent vending machine and some exercises to motivate students to practice more in this area as well as to further develop and modify the system that’s already given. This tutorial will be very useful for students since they won’t need to search in different references to have a full idea of the process.

The tutorial is included in appendix A in this thesis.
CHAPTER 5: CONCLUSION AND FUTURE WORK

CONCLUSION

This thesis focuses on the design and development process for embedded systems. It describes in detail how the software design process differs from the software/hardware co-design system design process needed for embedded systems. A full detailed embedded system design example has been illustrated using the example of an intelligent vending machine. This helps in better understanding the design process. It also includes some process management basics which will help developers to manage time and meet deadlines. This thesis helps people who work in software/hardware co-design to have all details needed for the embedded system design process to learn to implement a better design for both industry and university students. The process is described in a tutorial that is part of this thesis.

FUTURE WORK

Future work can include improving the design process by adding annotations to UML diagrams to provide a better overview of the system’s functionality. It can also include extending the design given here to a more complicated intelligent vending machine, for example, an IVM that has voice recognition, mobile payment
option, and thief detector. In addition, other examples of embedded systems can be created.

Another useful addition would be to include details about using the C programming language in embedded systems. There are a number of features of the C language which are not really appropriate for use in embedded systems. For example, recursion is not recommended for embedded systems programs due to the unpredictability of storage requirements for the stack and the overhead in making the recursive calls. The material on the C language could be included as a tutorial to help students and beginning workers in industry learn how to better implement embedded systems.
REFERENCES


[29] Agile Vs Waterfall methods. [Online]. Available: 


[31] UML 1.4 diagrams. [Online]. Available: 


Design for testability. [Online]. Available:


Intelligent vending machine white paper. [Online]. Available:


Intelligent vending machine market. [Online]. Available:

[38] Applications of UML. [Online]. Available:

[39] Use case diagram. [Online]. Available:

[40] Use case diagram. [Online]. Available:

[41] Class diagram. [Online]. Available:

[42] Class diagram. [Online]. Available:


[44] Sequence diagram. [Online]. Available:

[45] UML 2.0 tutorial. [Online]. Available:


[46] Gantt chart. [Online]. Available:


[47] HDDlock project. [Online]. Available:

APPENDIX A: THE DEVELOPMENT PROCESS FOR HARDWARE/SOFTWARE EMBEDDED SYSTEMS: EXAMPLE SYSTEMS AND TUTORIAL

The tutorial:

This tutorial provides a step-by-step guide that students can follow as they develop their own class projects. It uses the intelligent vending machine (IVM) as a helping example for improving their understanding of the process. In the exercises we will also use a simpler (standard) vending machine (VM).

A.1 Design process steps

Step 1. Requirements gathering and specification writing. Development of system acceptance tests.

Step 2. Design.

Step 2A. Development of sequence diagrams for functional use cases, identification of components.

Step 2B. Definition of components and class/component diagram showing interconnections. Development of unit tests.

Step 2C. State diagram defining overall system control.
Step 3. Implementation and testing.

Step 3A. Development of each component (hardware or software) and low-level testing.

Step 3B. Integration of components and unit testing.

Step 3C. System testing.

There are some design methods that are commonly used for embedded systems development. The steps above are used by what is known as the waterfall method. The waterfall method is a fixed method that supports a sequential design process where development flows sequentially from the starting point to the ending point. But we can modify our method to use newer agile methods. Agile methods are more flexible since they support an incremental and iterative approach to software design. In agile methods, the designers have the flexibility to respond to requirement changes as they arise during the course of the product development. We will use the waterfall method in this tutorial.

A.2 Design process details:

Step 1. Requirements gathering and specification writing. Development of system acceptance tests.
For your project, you need to gather requirements from the customer and from potential users of your project. You can gather these requirements through discussions and possibly surveys. Once you have a set of requirements, you will use them to develop the project specifications. Here we will employ use cases for functional specifications and text descriptions for nonfunctional specifications. At this stage we will also develop system tests from the specifications. Once the customer agrees that these system tests capture the requirements they had in mind, these system tests can serve as acceptance tests for the system.

➢ What is a use case diagram?

The use case diagram is the simplest representation of how the user/s interact with the system. A use case diagram includes use cases that represent the system functionalities, actors that represent users (animate, such as a human, or inanimate, such as another system) that interact with the system, and the connections between them.

Example:

Figure 1A shows the use case diagram for the intelligent vending machine (IVM). Each functional use case is shown in an oval, with connections to the participating actors. A rectangle representing the systems boundary contains all the separate use cases. The dashed arrows represent an extended connection between two use cases.
For example, “Select payment method” is extended from “Pay for product” use case. We will not enter “Select payment method” use case unless we have entered “Pay for product” use case.

Figure 1A: IVM use case diagram
Each functional use case is also described verbally, as shown below.

*Flow of events for use cases:*

Participating actors for the set of use cases: Customer, Manager, and Employee.

Use cases:

- **Name: Select product**

  *Participating actors: Customer.*

  *Flow of events:*

  8. A message will appear on the keypad LCD: “Please select a product”.

  9. Customer hits a button from the keypad related to their selection.

  10. A product advertisement or a related products advertisement will appear on the Advanced LCD.

  11. Processor checks the product availability.

  12. Processor updates the database and sends the product price.

  13. The product price will be displayed in the keypad LCD.


  *Entry condition: Customer hits a button from the keypad related to their selection.*
Exit condition: A message will appear on the keypad LCD “Please select payment option: 1.Cash 2.Credit/Debit Card”.

Quality requirements: Speed of response to all actions, nicely behaved buttons, and high resolution LCDs.

Acceptance Tests Developed During Analysis and Specification:

- Select product:

4. When customer has not pressed any button on the keypad, a message “Please select a product” should appear on the keypad LCD.

5. When customer presses a button on the keypad to select a product, a new advertisement will appear on the Advanced LCD, a price will be shown on the keypad LCD, and a message will appear on the keypad LCD “Please select payment option: 1.Cash 2.Credit/Debit Card”.

6. (Quality) when customer presses any button, the system should respond promptly.

Exercises:

1. What changes can we make in the use case diagram for an IVM? What are the differences between IVM use cases and VM use cases?
2. What modifications could be done to better support the security requirement?

3. Write the flow of events for the (Pay for product, Select payment method, and Insert cash OR slide credit/debit card) use cases.

4. Write the acceptance tests for the (Pay for product, Select payment method, and Insert cash OR slide credit/debit card) use cases. Be sure to include tests for quality requirements/specifications.

Step 2. Design.

Step 2A. Development of sequence diagrams for functional use cases, identification of components.

Use cases are: Select product, Pay for product, Select payment method, Insert cash OR Slide credit/debit card, Remotely control the system, Collect cash, and Restock products.

➢ What is a sequence diagram?

A sequence diagram represents the time view of interactions between the system’s objects to achieve the behavior goal of the system. It describes classes and objects in a specific scenario and the sequence of the exchanged messages.
between the objects to give the scenario’s functionality. It’s related to the use case realization of the system.

Figure 2A shows the sequence diagram for Select product and Pay for product for the intelligent vending machine.

Figure 2A: IVM Sequence Diagram
Exercises:

1. Draw a sequence diagram for unexpected cases, for example: The customer inserts less cash than the product’s price, the credit card was invalid, or the product is sold out.

Step 2B. Definition of components and class/component diagram showing interconnections. Development of unit tests.

Through developing the sequence diagram(s), we identify the components needed in this system: Customer, Keypad, Processor, Product, Coin Sensor, Cashless Sensor, Adv. LCD, and Ethernet.

➢ What is the class diagram?

The most popular diagram in UML is the class diagram that models the static view of the application. It represents the structure of the system by showing the system’s classes and attributes, operations/methods, and the relationships between objects.

Example:

Figure 3A shows the class diagram for the intelligent vending machine (IVM).
This diagram can also be used to develop unit tests to test the connections among the modules. These tests often use preconditions, postconditions, and invariants to test the module links and communication.

Figure 3A: IVM Class Diagram

Table 1A shows each class with a determination of whether to implement it in hardware or software and the reasons for the choice.
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Hardware</th>
<th>Software</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Yes</td>
<td>Yes</td>
<td>Processor is the hardware part that controls all software parts</td>
</tr>
<tr>
<td>Cashless sensor</td>
<td>Yes</td>
<td>No</td>
<td>Hardware part that isn’t responsible for running software functions</td>
</tr>
<tr>
<td>Coins sensor</td>
<td>Yes</td>
<td>No</td>
<td>Hardware part that isn’t responsible for running software functions</td>
</tr>
<tr>
<td>Keypad</td>
<td>Yes</td>
<td>No</td>
<td>It’s a hardware part that has good connections with software classes</td>
</tr>
<tr>
<td>Ethernet</td>
<td>Yes</td>
<td>No</td>
<td>Hardware part that isn’t responsible for running software functions</td>
</tr>
<tr>
<td>Adv. LCD</td>
<td>Yes</td>
<td>No</td>
<td>It’s a hardware part that has good connections with software classes</td>
</tr>
<tr>
<td>Product</td>
<td>Yes</td>
<td>No</td>
<td>Hardware only that represents the product itself</td>
</tr>
<tr>
<td>EmpIDSys</td>
<td>No</td>
<td>Yes</td>
<td>Software only; it has a database for all employee IDs</td>
</tr>
<tr>
<td>PwrMngt</td>
<td>No</td>
<td>Yes</td>
<td>Software only that receives data from hardware sensors</td>
</tr>
<tr>
<td>Coin_Controller</td>
<td>No</td>
<td>Yes</td>
<td>Software module that controls coins sensor</td>
</tr>
<tr>
<td>Security_Mngr</td>
<td>No</td>
<td>Yes</td>
<td>Software module that is responsible to keep the system secure; it must be an up to date module</td>
</tr>
<tr>
<td>Testing_HW</td>
<td>No</td>
<td>Yes</td>
<td>Software module that is responsible to keep testing all HW components</td>
</tr>
<tr>
<td>Cashless_Controller</td>
<td>No</td>
<td>Yes</td>
<td>Software module that controls cashless sensor</td>
</tr>
<tr>
<td>Product_Controller</td>
<td>No</td>
<td>Yes</td>
<td>Software module that controls products</td>
</tr>
<tr>
<td>Controller</td>
<td>No</td>
<td>Yes</td>
<td>Software module that controls all system’s components</td>
</tr>
</tbody>
</table>

Table 1A: HW/SW IVM classes
Exercises:

1. What classes could we add to the class diagram?

2. What functions and attributes should be in each class?

3. Are there classes that could be in hardware or software?

➢ What is a deployment diagram?

Deployment diagrams are used to describe the system’s hardware components wherever software components are deployed. They are used for visualizing the physical components’ topology. That’s why deployment diagrams are used for describing the system’s static deployment view. Deployment diagrams include nodes and the connections among these nodes.

Figure 4A shows the deployment diagram for the intelligent vending machine. This shows the main hardware components.
Step 2C. State diagram defining overall system control.

Figure 5A shows a basic state diagram for an intelligent vending machine.
Step 3. Implementation and testing.

Step 3A. Development of each component (hardware or software) and low-level testing.

For this step we need to employ strategies such as white box testing (software) and scan testing (hardware).

Step 3B. Integration of components and unit testing.

Step 3C. System testing.
Process Management

In every project in industry or in any class in the universities there are “deadlines” to complete tasks. Writing a development plan is an important part in completing the system and meet deadlines. Table 2A shows a general development schedule example using the waterfall model for an embedded system, and it is followed by Figure 6A that represents a Gantt chart for the waterfall example for this schedule.

<table>
<thead>
<tr>
<th>Task</th>
<th>Start date</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Define Requirements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Define customer needs</td>
<td>1-Jan</td>
<td>1</td>
</tr>
<tr>
<td>Define available resources</td>
<td>2-Jan</td>
<td>1</td>
</tr>
<tr>
<td>Create req. Documents and use cases</td>
<td>3-Jan</td>
<td>3</td>
</tr>
<tr>
<td>*Milestone: Requirements defined</td>
<td>6-Jan</td>
<td>0</td>
</tr>
<tr>
<td><strong>2. Create UML description</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create use case diagram</td>
<td>7-Jan</td>
<td>1</td>
</tr>
<tr>
<td>Create class diagram</td>
<td>8-Jan</td>
<td>1</td>
</tr>
<tr>
<td>Create deployment diagram</td>
<td>9-Jan</td>
<td>1</td>
</tr>
<tr>
<td>Create sequence diagram</td>
<td>10-Jan</td>
<td>1</td>
</tr>
<tr>
<td>Revise UML diagrams. Complete test cases</td>
<td>11-Jan</td>
<td>3</td>
</tr>
<tr>
<td>*Milestone: UML Description completed</td>
<td>14-Jan</td>
<td>0</td>
</tr>
<tr>
<td><strong>3. Implementation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start coding. Start testing</td>
<td>15-Jan</td>
<td>8</td>
</tr>
<tr>
<td>Build up all components together</td>
<td>24-Jan</td>
<td>5</td>
</tr>
<tr>
<td>Test after integration</td>
<td>30-Jan</td>
<td>2</td>
</tr>
<tr>
<td>*Milestone: Implementation completed</td>
<td>1-Feb</td>
<td>0</td>
</tr>
<tr>
<td><strong>4. Testing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Define a test plan</td>
<td>2-Feb</td>
<td>1</td>
</tr>
<tr>
<td>Test the whole system</td>
<td>3-Feb</td>
<td>3</td>
</tr>
<tr>
<td>Complete testing. Write final report</td>
<td>7-Feb</td>
<td>5</td>
</tr>
<tr>
<td>*Milestone: System is ready to use.</td>
<td>12-Feb</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2A: Development process schedule for an embedded system
A Gantt chart is a chart that is used in project management to represent activities (tasks) versus time in a useful and simple way. The left side of the chart represents a list of all tasks. Every task is represented by a bar with a particular position for a start date, and a particular length for time duration to complete the task.

![Gantt Chart](image)

**Figure 6A: Embedded system’s Gantt chart**

**Exercises:**

1. Define a schedule and draw a Gantt chart for designing a traditional ATM machine.
Additional questions and exercises:

1. Draw a use case, class, deployment, and sequence diagram for IVM that has a voice recognition module and has a mobile payment method.

2. Draw a use case, class, deployment, and sequence diagram for a traditional ATM machine.

3. Define the flow of events and the acceptance test for the use cases shown in exercise 2.

4. Show which classes should be HW, SW or both for exercise 2.