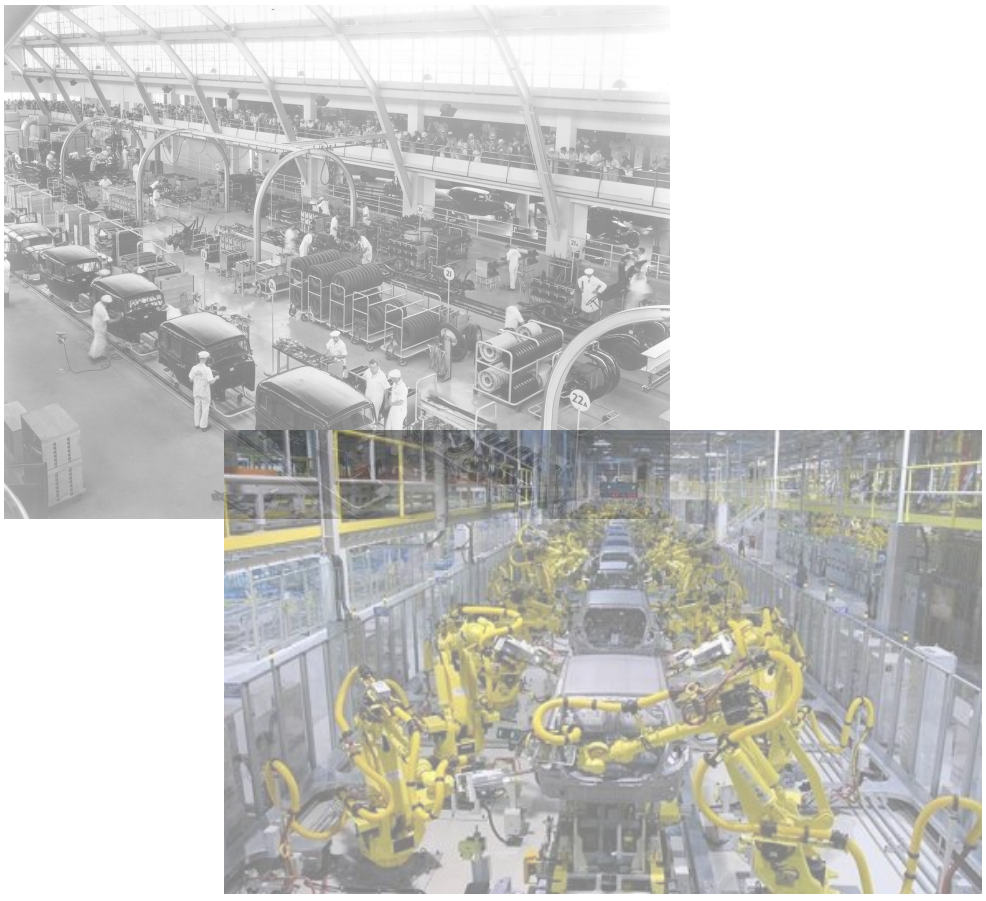
AssemAssist: An Automobile Assembly Line Support System



Software-ontwerp:
project 2021-2022

Iteration 1

Contents

1 Introduction

For the course *Software-ontwerp: project*, you will develop a software system supporting the operations of an automobile assembly line.

In Section 2, we explain how the project is organized, discuss the quality requirements for the software you will develop and how we evaluate the solutions. In Section 3, we show a diagram of the domain model, provide a glossary of automobile industry jargon and explain the problem domain of the application. The use cases are described in detail in Section 4.

2 General Information

In this section, we explain how the project is organized, what is expected of the software you will develop and the deliverables you will hand in.

2.1 Team Work

For this project, you will work in groups of four. Each group is assigned an advisor from the educational staff. If you have any questions regarding the project, you can contact your advisor and schedule a meeting. When you come to the meeting, you are expected to prepare specific questions and have sufficient design documentation available. *If the design documentation is not of sufficient quality, the corresponding question will not be answered.* It is your own responsibility to organize meetings with your advisor and we advise to do this regularly. Experience from previous years shows that groups that regularly meet with their advisors produce a higher quality design. If there are problems within the group, you should immediately notify your advisor. Do not wait until right before the deadline or the exam!

To ensure that every team member practices all topics of the course, a number of roles are assigned by the team itself to the different members at the start of each iteration (or shortly thereafter in case of the first iteration). A team member that is assigned a certain role will give the presentation or demo corresponding to that role at the end of the iteration. That team member is *not supposed to do all of the work concerning his task*! He must, however, take a coordinating role in that activity (dividing the work, sending reminders about tasks to be done, make sure everything comes together, etc.), and be able to answer most questions on that topic during the evaluation. The following roles will be assigned round-robin:

Design Coordinator The design coordinator coordinates making the design of your software.

Testing Coordinator The testing coordinator coordinates the planning, designing, and writing of the tests for the software.

Domain Coordinator The domain coordinator coordinates the maintenance of the domain model.

As already mentioned, the goal of these roles is to make every team member participate in all aspects of the development of your system. **During each presentation or demo, every team member must be able to explain the**

used domain model, the design of the system, and the functioning of your test suite.

2.2 Iterations

The project is divided into 3 iterations. In the first iteration, you will implement the base functionality of the software. In subsequent iterations, new functionality will be added and/or existing functionality will be changed.

2.3 The Software

The focus of this course is on the *quality* (maintainability, extensibility, stability, readability,...) of the software you write. We expect you to use the development process and the techniques that are taught in this course. One of the most important concepts are the General Responsibility Assignment Software Principles (GRASP). These allow you to talk and reason about an object oriented design. **You should be able to explain all your design decisions in terms of GRASP**.

You are required to provide class and method documentation as taught in previous courses (e.g. the OGP course), as appropriate. When designing and implementing your system, you should use a *defensive programming style*. This means that the *client* of the public interface of a class cannot bring the objects of that class, or objects of connected classes, into an inconsistent state.

Unless explicitly stated in the assignment, you do not have to take into account persistent storage, security, multi-threading, and networking. If you have doubts about other non-functional concerns, please ask your advisor.

2.4 Testing

All functionality of the software should be tested. **For each use case, there should be a dedicated scenario test class.** For each use case flow, there should be at least one test method that tests the flow. Make sure you group your test code per step in the use case flow, indicating the step in comments (e.g. `// Step 4b`). Scenario tests should not only cover success scenarios, but also negative scenarios, i.e., whether illegal input is handled defensively and exceptions are thrown as documented. You determine to which extent you use unit testing. The testing coordinator briefly motivates the choice during the evaluation of the iteration.

Tests should have good coverage, i.e. a testing strategy that leaves large portions of a software system untested is of low value. If your IDE's test coverage functionality reports that only 60% of your code is covered by tests, this indicates there may be a serious problem with (the execution of) your testing strategy. However, be careful when drawing conclusions from both reported high coverage and reported low coverage (and understand why you should be careful). The testing coordinator is expected to briefly report the results of running the test suite with coverage tracking during the evaluation of the iteration.

2.5 UML Tools

There are many tools available to create UML diagrams depicting your design. You are free to use any of these as long as it produces correct UML. One of these UML tools is Visual Paradigm. You can find a link to a website where you can download the program and find the license key. However, we recommend UMLet for its simplicity.

2.6 What You Should Hand In

Exactly *one* person of the team hands in a ZIP-archive via Toledo. The archive contains the items below and follows the structure defined below. **Make sure that you use the prescribed directory names.**

- directory `groupXX` (where XX is your group number (e.g. 01, 12, ...))
  - `doc`: a folder containing, for each API you define in your system, a subfolder with the Javadoc for that API. For example, if you implemented a library of GUI components, include a subfolder with the Javadoc for this library's API.
  - `diagrams`: a folder containing UML diagrams that describe your design (at least one structural overview of your entire design, and sufficient detailed structural and behavioural diagrams to illustrate every use case)
  - `src`: your system's source code
  - `system.jar`: your system's compiled executable
  - `design.pdf` (optional): a document that clarifies your design and the main decisions; this document can be a prose text or a slide deck, or any other form that you deem appropriate.

When including your source code into the archive, make sure to *not include files from your version control system.* Make sure you choose relevant file names for your analysis and design diagrams (e.g. `SSDsomeOperation.png`). You do **not** have to include the project file of your UML tool, only the exported diagrams. We should be able to start your system by executing the JAR file with the following command: `java -jar system.jar`.

Needless to say, the general rule that anything submitted by a student or group of students must have been authored exclusively by that student or group of students, and that accepting help from third parties constitutes exam fraud, applies here.

2.6.1 Late Submission Policy

If the zip file is submitted N minutes late, with $0 \leq N \leq 240$, the score for all team members is scaled by $(240 - N)/240$ for that iteration. For example, if your solution is submitted 30 minutes late, the score is scaled by 87.5%. So the maximum score for an iteration for which you can earn 4 points is reduced to 3.5. If the zip file is submitted more than 4 hours late, the score for all team members is 0 for that iteration.

2.6.2 When Toledo Fails

If the Toledo website is down – and *only* if Toledo is down – at the time of the deadline, submit your solution by e-mailing the ZIP-archive to your advisor. The timestamp of the departmental e-mail server counts as your submission time.

2.7 Evaluation

After iteration 1, and again after iteration 2, there will be an intermediate evaluation of your solution. An intermediate evaluation lasts 35 minutes and consists of: a presentation about the design and the testing approach, accompanied by a demo of the system and the test suite.

The intermediate evaluation of an iteration will cover only the part of the software that was developed during that iteration. Before the final exam, the *entire* project will be evaluated. It is your own responsibility to process the feedback, and discuss the results with your advisor.

The evaluation of an iteration is planned in the week after that iteration. Immediately after the evaluation is done, you mail the PDF file of your presentation to Prof. Bart Jacobs & Tom Holvoet and to your advisor.

2.7.1 Presentation Of The Current Iteration

The main part of the presentation should cover the design. The motivation of your design decisions *must* be explained in terms of GRASP principles. Use the appropriate design diagrams to illustrate how the most important parts of your software work. Your presentation should cover the following elements. Note that these are not necessarily all separate sections in the presentation.

1. A discussion of the high level design of the software (use GRASP patterns). Give a rationale for all the important design decisions your team has made.

2. A more detailed discussion of the parts that you think are the most interesting in terms of design (use GRASP patterns). Again we expect a rationale here for the important design decisions.

3. A discussion of the testing approach used in the current iteration.

4. An overview of the project management. Give an approximation of how many hours each team member worked. Use the following categories: group work, individual work, and study (excluding the classes and exercise sessions). In addition, insert a slide that describes the roles of the team members of the current iteration, and the roles for the next iteration. Note that these slides do not have to be presented, but we need the information.

Your presentation should not consist of slides filled with text, but of slides with clear design diagrams and keywords or a few short sentences. The goal of giving a presentation is to communicate a message, not to write a novel. All design diagrams should be *clearly readable* and use the correct UML notation. It is therefore typically a bad idea to create a single class diagram with all information. Instead, you can for example use an overview class diagram with only the most

important classes, and use more detailed class diagrams to document specific parts of the system. Similarly, use appropriate interaction diagrams to illustrate the working of the most important (or complex) parts of the system.

2.8 Peer/Self-assessment

In order for you to critically reflect upon the contribution of each team member, you are asked to perform a peer/self-assessment within your team. For each team member (including yourself) and for each of the criteria below, you must give a score on the following scale: *poor/lacking/adequate/good/excellent*. The criteria to be used are:

- Design skills (use of GRASP and DESIGN patterns, ...)
- Coding skills (correctness, defensive programming, documentation,...)
- Testing skills (approach, test suite, coverage, ...)
- Collaboration (teamwork, communication, commitment)

In addition to the scores themselves, we expect you to briefly explain for each of the criteria why you have given these particular scores to each of the team members. The total length of your evaluation should not exceed 1 page.

Please be fair and to the point. Your team members will not have access to your evaluation report. If the reports reveal significant problems, the project advisor may discuss these issues with you and/or your team. Please note that your score for this course will be based on the quality of the work that has been delivered, and not on how you are rated by your other team members.

Submit your peer/self-assessment by e-mail to both Prof. Bart Jacobs & Tom Holvoet and your project advisor, using the following subject: **[SWOP] peer-/self-assessment of group $groupnumber$ by $firstname$ $lastname$**.

2.9 Deadlines

- The deadline for handing in the ZIP-archive on Toledo is **18 March, 2022, 3:30pm**.
- The deadline for submitting your peer/self-assessment is **20 March, 2022, 6pm**, by e-mail to both your project advisor and Prof. Bart Jacobs & Tom Holvoet.

3 Problem Domain Analysis

3.1 Domain Model

Figure 1 shows the domain model of AssemAssist.

3.2 Glossary

This subsection gives an overview of the vocabulary used in this document. The different introduced concepts are listed in the following table.

Car Manufacturing Company	A manufacturing company produces cars and sells them to individual garage holders. Well known examples are General Motors (Opel, Saab, ...), Toyota (Lexus, Toyota, ...) and Volkswagen Group (Audi, Volkswagen, Bently, ...).
Car Model	A specific model of a certain car manufacturing company (e.g. Ford Mustang).
Car Model Specification	The specification of a car model consists of the set of car parts of which each car that complies to the model, should be composed. For each car part also a number of concrete, interchangeable instances is specified.
Car Order	A car order is a request made by a garage holder to a car manufacturing company to produce a car adhering to a specific car model. It gives a concrete selection of the different alternatives per car part specified by the specification of the car model.
Production Schedule	The production schedule is a plan that specifies the start and end time of each pending car order.
Car Assembly Process	A car assembly process describes the sequence of assembly tasks that have to be performed to build a car according to its order description.
Assembly Task	An assembly task is a clearly defined concrete step in the process of manufacturing a car. It consists of a list of indivisible actions to be performed in sequence at a single workpost. Examples of assembly tasks are : painting a car blue, installing a 2.5L engine, inserting a 5-speed automatic gearbox, etc.
Assembly Line	The assembly line is a sequence of workposts, connected by a conveyor belt. The conveyor belt can only move if all the current assembly tasks are fulfilled.

Work Station	A work station is a location at the assembly line that is specifically equipped to perform particular assembly tasks.
Garage Holder	The owner of a car garage where cars are being sold. He forms the link between the car manufacturer and the end consumer. He is the one who places the orders to a car manufacturing company.

3.3 Domain Description

The automobile manufacturing company assembles cars from raw parts, according to the orders made by their garage holders throughout the country. The company offers one car model, using a common set of components, but with several options that allow customization. The garage holder can offer his customer the following options:

- *Body*: sedan, break
- *Color*: red, blue, black, white
- *Engine*: standard 2l 4 cilinders, performance 2.5l 6 cilinders
- *Gearbox*: 6 speed manual, 5 speed automatic
- *Seats*: leather black, leather white, vinyl grey
- *Airco*: manual, automatic climate control
- *Wheels*: comfort, sports (low profile)

The automobile manufacturing company is built around an assembly line that runs throughout the facility. The raw chassis starts at the beginning of the conveyor belt and passes several work posts, resulting in a completed car at the end of the conveyor belt (see Figure 2). The manufacturing schedule is setup in such a way that tasks on a car at a given work post can be completed in an hour, allowing the company to create a fairly accurate production schedule. In rare occasions, mechanics fail to finish their tasks in an hour, which can cause delays in the manufacturing process. The schedule is updated during the working day, so it is possible that an additional car can be manufactured (in case numerous tasks end earlier), or a car is shifted to the next day (in case of delays).

The three work posts and their respective tasks are:

- *Car Body Post*: this work post is responsible for mounting a body on the chassis and painting it. The tasks at this workstation are:
  - Assembly Car Body
  - Paint Car
- *Drivetrain Post*: this work post is responsible for installing the mechanical parts, taken care of by the following tasks:

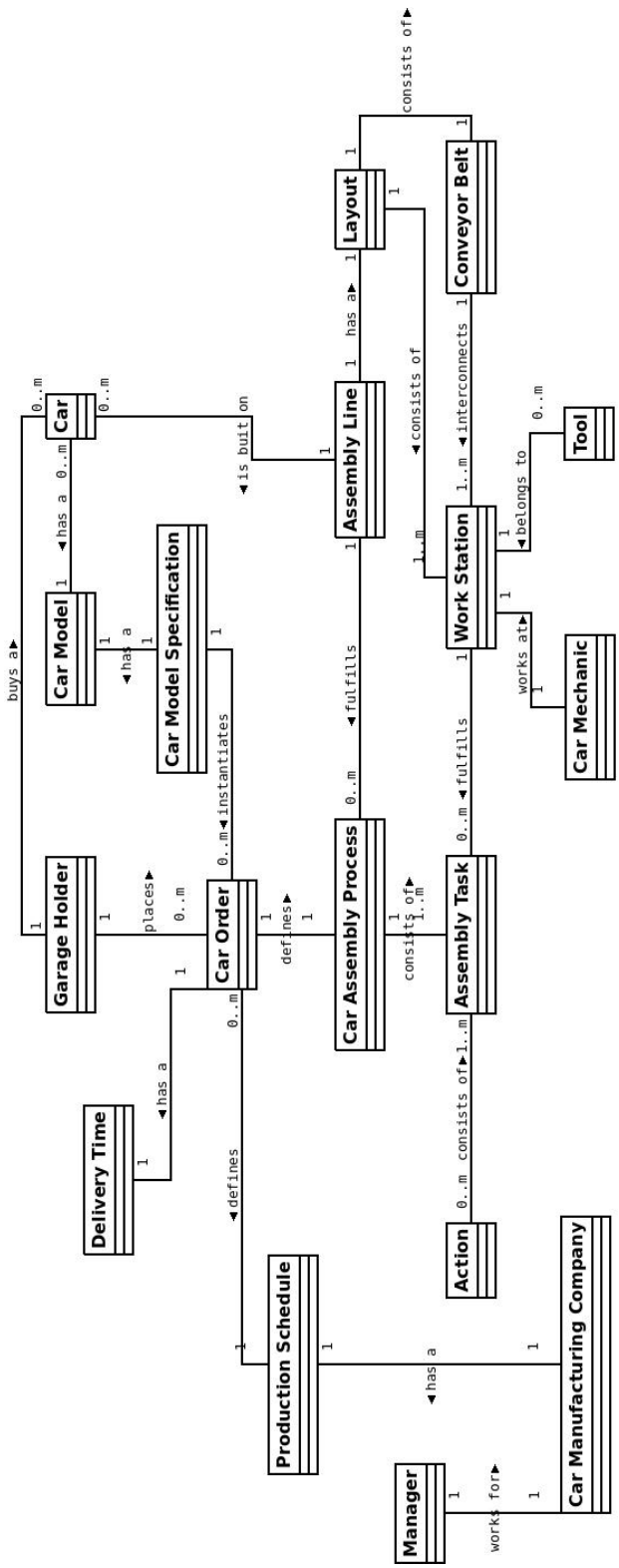


Figure 1: Domain model

  - Insert engine
  - Insert gearbox

- *Accessories Post*: this work post is responsible for the final parts, performed in the following tasks:
  - Install seats
  - Install airco
  - Mount wheels

The production schedule maintained by the company adheres to the following rules:

- Orders are processed on FCFS basis (first come, first served).
- Car mechanics work in two shifts, meaning that the work posts at the assembly line are operated from 06:00 till 22:00. All mechanics of a shift start and stop at the same time (so some mechanics will be idle in the beginning/end of a day).
- The order of work posts is fixed (see above) and in normal circumstances, a car spends one hour at every work post.
- Scheduling must ensure that cars are produced in a single day, and cars can not be left on the conveyor belt at the end of the working day. In case of delays, mechanics do over time to finish the cars on the belt, but the scheduling algorithm must minimize this overtime. If a shift performs hours in overtime, it is deducted from the shift's next working day, meaning that the next working day ends earlier. The schedule is updated dynamically during the day, to account for time gains or losses, aiming to produce as many cars as possible within the working day.
- In ideal circumstances, assembly of a car takes three hours (one hour in each work post).

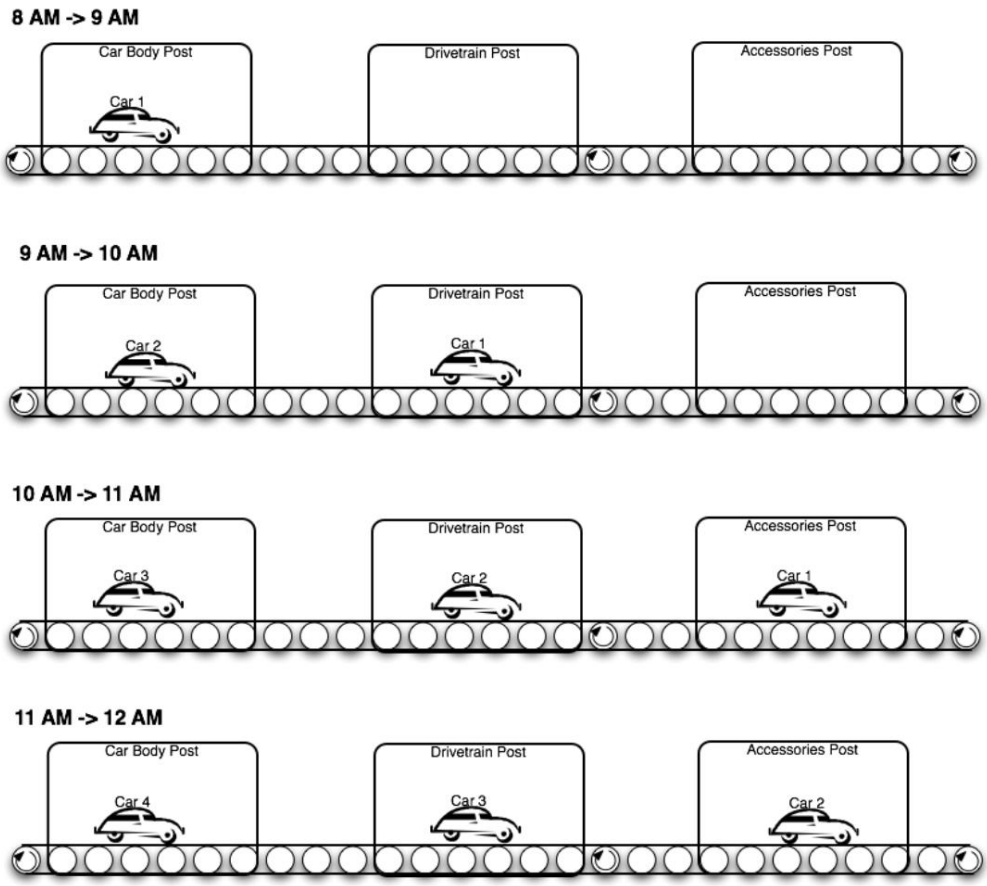


Figure 2: Production scenario on the assembly line

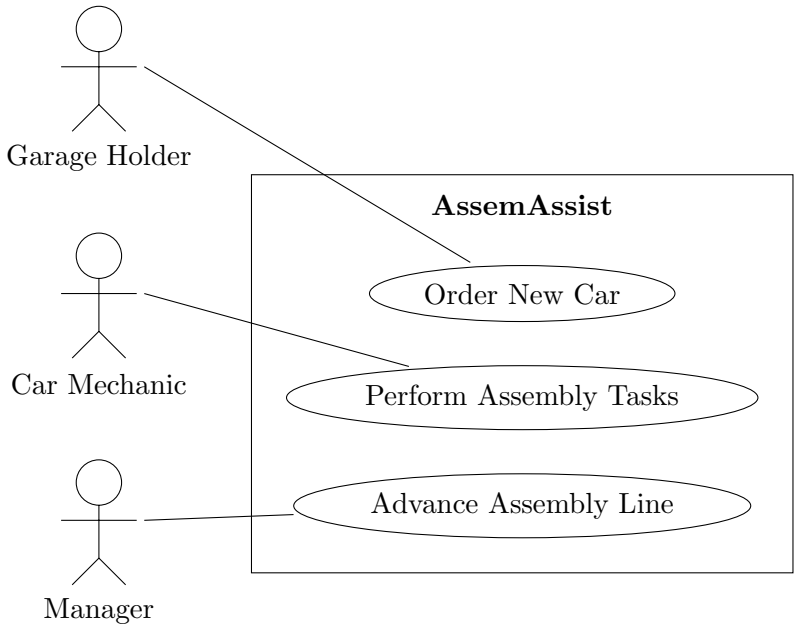


Figure 3: Use case diagram for AssemAssist.

4 Use Cases

Figure 3 shows the use case diagram for AssemAssist. The following sections describe the various use cases in detail.

4.1 Use Case: Order New Car

Primary Actor: Garage Holder.

Precondition: The garage holder is successfully logged into the system.

Success End Condition: An order for a new car has been placed.

Main Success Scenario:

1. The system presents an overview of the orders placed by the user, divided into two parts. The first part shows a list of pending orders, with estimated completion times, and the second part shows a history of completed orders, sorted most recent first.
2. The user indicates he wants to place a new car order.
3. The system shows a list of available car models.
4. The user indicates the car model he wishes to order.
5. The system displays the ordering form.
6. The user completes the ordering form.
7. The system stores the new order and updates the production schedule.
8. The system presents an estimated completion date for the new order.

Alternate Flow:

1. (a) The user indicates he wants to leave the overview.
2. The use case ends here.

Alternate Flow:

6. (a) The user indicates he wants to cancel placing the order.
7. The use case returns to step 1.

4.2 Use Case: Perform Assembly Tasks

Primary Actor: Car Mechanic.

Precondition: The car mechanic is successfully logged into the system.

Success End Condition: One or several tasks have been completed by the car mechanic.

Main Success Scenario:

1. The system asks the user what work post he is currently residing at.
2. The user selects the corresponding work post.
3. The system presents an overview of the pending assembly tasks for the car at the current work post.
4. The user selects one of the assembly tasks.
5. The system shows the assembly task information, including the sequence of actions to perform.
6. The user performs the assembly tasks and indicates when the assembly task is finished.
7. The system stores the changes and presents an updated overview of pending assembly tasks for the car at the current work post.
8. *The use case continues in step 4.*

Alternate Flow:

8. (a) The user indicates he wants to stop performing assembly tasks.
9. The use case ends here.

4.3 Use Case: Advance Assembly Line

Primary Actor: Manager.

Precondition: The manager is successfully logged into the system.

Success End Condition: The assembly line has been moved forward one work post.

Main Success Scenario:

1. The user indicates he wants to advance the assembly line.

2. The system presents an overview of the current assembly line status, as well as a view of the future assembly line status (as it would be after completing this use case), including pending and finished tasks at each work post.
3. The user confirms the decision to move the assembly line forward, and enters the time that was spent during the current phase (e.g. 45 minutes instead of the scheduled hour).
4. The system moves the assembly line forward one work post according to the scheduling rules.
5. The system presents an overview of the new assembly line status.
6. The user indicates he is done viewing the status.

Alternate Flow:

4. (a) The assembly line can not be moved forward due to a work post with unfinished tasks.
5. The system shows a message to the user, indicating which work post(s) are preventing the assembly line from moving forward.
6. *The use case continues in step 6.*

Good luck!
The SWOP Team members