Ex. 1

There are 2 packages located in the src folder, one is called "main" and the other is called "test". The classes in the test folder are not useful for understanding how the system works, which is why it should be ignored in the depiction. The main folder contains a java and a resource package. Since the resource folder doesn't contain any java code it should be ignored as well.
Thus, the classes and packages contained in java.nl.tudelft.jpacman are the most important parts to understand how the game works. Those are "board", "game", "level", "npc", "sprite", "ui" and the classes "Launcher" and "PacManConfigurationException".
At this level of abstraction, one can already see that this is a game with a board and quite possibly provides some interaction with npc's. Looking at the individual packages doesn't reveal many details to how the game works yet.

Ex. 1.1

(ii)

We've chosen two levels of abstraction. With only one level, starting from the Launcher class, there are a lot of dependencies missing. Many classes don't get touched by the Launcher directly and the "sprites" package doesn't contain any class that gets touched at all. It seems like there is a lot of information missing.
However, with two levels we are not only able to find further dependencies, but also see clearly that there are many dependencies between different packages. This also has some kind of a structure where classes from one package may only depend on classes from one or two other packages, three at most. Also, very important to notice is that many different classes from different packages rely on a single one whereas we found up to six dependences on class.
Depicting a third level gets messy. Most of the classes got already touched, so going down a level mostly reveals dependencies that are already shown. Not a lot of new insight would be generated by doing so, thus we decided against it.
There are still classes that are untouched at only two levels but revealing only a few of them for each new level is not worth it. The overview will suffer to a point where we think that two levels are the sweet spot.

(iii)

The main understanding of this depiction is that some packages mainly work on their own, whereas others heavily rely on other classes and their dependencies. Very important to notice is that the main class, or in this case the "Launcher" cannot depend on every other class or package. It would simply become a very confusing class on its own and the readability would decrease immensely. Also, there is no strange cross dependence where the viewer could get confused. So, the task a package has to do is pretty clear since the naming of it and the contained classes speak for themselves.

Ex. 1.2

(ii)

We decided to not draw every branch of the call-graph with the same depth. Each branch has its reasons to be there but not each one is equal as important as others. Some methods called in the launcher simply make use of a constructor or add a few events, where we thought there is no need to show each addition of an event. This statement holds true for deeper levels as well, so we won't talk about each method that only adds basics. Other methods however go a lot deeper with the focus being on the UI. Thus, the branch that creates the UI deserves a third level of depth to show that it is composed by smaller elements.

(iii)

The call order, especially in the first level, is no coincidence. The first call builds the basis to create a game object, which will be present throughout the calls of launching Pacman. The second call produces a UI Builder to which we add the direction keys in the third call. Now the builder, using the game object, is used to create the UI itself in the fourth call. In the end, the UI provides a method that starts the thread, or better said starts the whole Pacman game. Important to understand here is that the creation of objects relies on previously created ones, meaning that changing the sequence of calls is not really possible unless it is a simple call chain of e.g., adding buttons. This dependence of objects is drawn in the call-graph with dotted arrows the avoid confusion. Not every possible one is shown though, only the more important ones.

Ex. 2.1

To start discussing we collected all ideas of CRC cards we had and added them to the field. All cards with black titles, besides the I/O Formatter, were there from the beginning. The TurnValidator was there from the beginning as well but had different tasks than what he has now. In the end, he got split into two classes: the validator and the handler. The red cards got removed from the pool with needed classes since we found better solutions for them. The green card got added to the pool to simplify other classes and take over some tasks. Classes with question marks were debatable from the beginning, no one was sure if we'll need them. Obviously, many classes stood with different responsibilities in the beginning than what they do now.

First, we establish a set of classes we feel is appropriate for the specifications given in the assignment. This leads to the classes "Piece" for the movable pieces, "Board" for the abstract projections of the game area and a "Game" class for metadata, as a gateway, for round-bounded information and as a logical state container. It will be the main entry point of the program as well as handle the user input and announce important events like winning and invalid input. It also takes care of printing the next board state right before accepting a new turn. For the sake of object orientation, we also try to formalize the aspect of turn validation and execution, this is constated as follows: First, we take away any initiative responsibility from the board class by relying simply on the result of the "TurnHandler" class. The handler will accept the turn and redirect it to a validator class first. The validator class will then check whether the turn can be executed using information about the involved pieces and their positions against other pieces on the board and report back the decision. The handler will then execute the turn and let the validator re-validate the post-turn state. If there is an action required, the game class will be notified and redirects the message to the user. The active turn will only complete once there is no conflict and the handler sets a "completed" status. We will not keep updating the CRC card set but instead reflect any further structural changes directly in the UML, Flow Chart and the source code itself.

Ex. 2.2

The main classes are "Board", "Game", "TurnHandler" and "TurnValidator" as they are sufficient to accommodate every task in an orchestrated manner. To follow the OOP pattern, the "Piece", "Turn" and "TilePosition" classes will store additional information for the other classes to use and decide on with their logic. The "Game" class in particular has the greatest interfacing capacity as it must act as a gateway for the user, the "TurnHandler" and the board state. The "TurnHandler" will also play a central role as it must check and redirect the input, handle turns based on the validation results and then preform the valid turns on the board. The "Board" class will serve as the static mutable source of truth. It will be read and instructed to perform moves on verified turns. Lastly, the "TurnValidator" houses all the specified game logic and is an essential part of the program as it defines the behavior of the game by judging on requested turn input.

Ex. 2.3

All the classes mentioned previously as being of assistive nature will in the end be a design choice as they could be merged or otherwise distributed among the main classes. For the sake of this exercise, we deem them to be purposeful abstractions as described in the following. The "Turn" class will be the carrying structure used across the turn processing pipeline for consistency and easy proxy for decisions at different parts of the application. The "Piece" abstraction will be useful for the turn validation process, the "IOFormatter" utility class aids with correct parsing and output formatting. The “Player” enum provides information about the active player which is used throughout the project but is considered a non-main component because of its relatively compact minor occurrence.