This software system includes 6 packages and additional main file. Each package has up to 10 classes. The first step is identifying connections of classes between packages. The second step – within the packages.

Step 1.

i and ii) If we were to draw all the connections between all the classes, things would get messy and not so informative. The optimal level of abstraction seems to be on the package level with showing some precise classes when it seems to be necessary (highlighting special features of the system). Reasons: first, we make use of the fact, that the designer of the software took time and effort to design software well, made it readable, organized classes in package, named the packages and classes meaningfully. Second, we get a big picture of what is happening in the software system without going too much in detail. At the same time, we highlight these special features (e.g., when one class heavily relies on the rest of the system)

The best way to proceed: run software system (using the main method in Launcher class) and see what it does. Once it’s clear, relate the packages to the app. Find an entry point, frequently it is a good idea to start with the class containing main method – “Launcher” in our case. To draw the diagram between packages we use import statements.

iii) what we understood:

1. In the package “Sprite” only class “PacManSpirits” relies on the external classes, hence we highlight this class. As regards other packages, they rely on PacManSpirits, Sprite and Animated Sprite only (mainly on the first two). We show this feature to underline that any changes made in these classes would lead to consequences in the rest of the system. The rest of the classes in this package are not imported anywhere. Hence, we do not show them in the diagram and abstract away from these details. Those classes are not coupled with the external classes.
2. All packages are depended on the package “Board”. All the classes of this package are imported in some other packages. Hence there is no need to show the exact dependencies and list all the classes, here we abstract on the package level. On the contrary – package “Board” depends only on two classes – “Sprite” and “PacManSpirits” – both members of the package “Sprite”. This feature is also shown. Conclusion: any changes made in package “Board” would have consequences in other parts of the system, whereas the package “Board” itself does not depend much on the external classes.

“

1. Only package “UI” and class “Launcher” rely on the package “Game”, specifically only on 2 classes: “Game” and “Game Factory”. We highlight this feature in the diagram as well. Similarly, only these two classes import stuff, hence we show them in the diagram.
2. As regards package “UI”, only class “Launcher” depends on it. Specifically, it imports interface “Action” and classes PacManUi and PacManUiBuilder. Similarly, mostly only these classes patriciate in importing stuff to UI. Therefore, they are shown in the diagram. Conclusion: this package not much coupled with the rest of the system.
3. As regards package “npc”, it contains subpackage – “ghost” and another class “Ghost”. Whenever something is imported from the package “ghost” it is either “GhostColor” or “GhostFactory”,
4. Finally, many packages rely on the classes from the package “Level”. Class “player” is imported in all the cases. Hence, we show it in the diagram, and abstract away from other classes.

Summing up, the diagram shows that 1. package on which everyone relies – “Board” – all the classes there are pretty frequently used outside of the package, 2. in other packages such classes as “PacManSpirits”, “Sprite”, “Player”, “GhostColour” “GhostFactory”, “Game” are also used by at least 2 external packages. 3. Class Launcher which includes the main method relies on each part of the system. 4. In the package Sprite everything except of “PacManSpirits” does not import anything.

Graphical user interface

Description automatically generated