# Summary of Packages

This project is divided into two main components: the “platform” sections deal primarily with essential display and movement features which do not require specific gameplay rules to function. These support overlapping objects, multiple types of interactions (other than collisions), storing movements of any sort (and reversing previously applied movements), and importantly, carrying out special actions after each movement on all game objects involved or affected by the movement. This allows each object to perform special actions not inherent to its presence on the map.

The “Sokoban” package uses this core platform to display an interface that allows the user to play Sokoban. The logic for interactions is divided between the Collision class and all the GameObject classes (such as crate/box, player, target, wall, etc.). The GameObject classes choose whether to allow themselves to be pushed/moved, and so to determine whether one can push an object, the player sends a request to push to the object in question. This object may choose to query other nearby objects for their decisions before replying back to the player with its own decision. During this chain of queries, a Movement is formed. If the end result of the queries is that the movement is to be approved, then the player applies this movement to the map, and the movement takes place.

# Description of Movement

Movement can be somewhat complex, so this is a breakdown of what takes place.

1. The GamePanel instance receives user keyboard input, and calls Map.move(Angle) with the Angle that represents the direction of movement in radians. For instance, 0 is movement to the right, π is movement to the left, etc.
2. The Map class receives this information and uses its own internal constant to transform the angle of movement into a vector of movement (by adding the number of pixels this movement should take up). The map class then calls Player.move(Vector) with this information.
3. The Player established as the main player for the map receives this vector and must decide whether or not to move based on its surroundings. To determine whether it should move, it must query 1) whether there are solid objects close by, and 2) those solid objects to see if they are willing to move.
   1. First off, the player uses its own private size/dimensions information to establish a rectangular area (measured in pixels and with a position on the map) which it would occupy should the movement take place. This is the movementArea.
   2. The player next creates an object of type Movement passing itself and its new position as arguments. This Movement object contains all the information relating to one net movement, but will not be executed unless the player decides to execute it explicitly.
   3. Then the player calls Collision.getSolidObjectsInArea(movementArea) to determine which solid objects (if any) currently occupy the area to which it wants to move. If none are present, the movement immediately is carried out.
   4. If solid objects are present, the player then asks these objects (by calling GameObject.movementQuery(Movement) on each object) whether they are willing to move. Each object has its own logic to determine whether movement should take place. In some cases (such as crates/boxes) the solid object will not be able to decide without looking at its own surroundings. This means that each solid object will in turn call movementQuery(Movement) on any other objects it deems must also participate in the movement. Each object called in this chain fashion will add itself to the Movement object.
   5. If the chain of movementQuery calls indicates (by returning true) that the movement may be carried out, the player calls Map.move(Movement) to have the movement be executed.
4. Once the map receives a movement, it calls Movement.execute(Map) to have the Movement object itself carry out the repositioning of objects.
5. The Movement will in turn call Map.moveObject(GameObject) as necessary to alter the registered locations of each object within the map.
6. As Map.moveObject(GameObject) is called, the map will make calls to internal helper functions to reposition the GameObject within its own internal position trees. These trees are used to optimize the searching for objects within specific areas used elsewhere. The details of the trees are not important, but it is important to call each helper function which will deregister each object from the position trees and reregister it at a different location.
7. When control returns to the map, the map then calls the object.postMovementAction() methods on all objects within the total area of movement. Each of these objects may react in its own special way, such as by altering its own graphic/image or by performing special operations on other objects or the map. Currently, only the Player class uses this function (to change its graphic’s direction).
8. Once this is done, control is passed back to the GamePanel instance. This instance first redraws the map, and then checks whether the map now indicates that it is solved and responds accordingly.

# Description of Drawing

1. The GamePanel instance receives a call to paintComponent(Graphics g). It then calls map.draw(Graphics g).
2. The map then receives this call and calls ScreenDrawer.draw(Graphics g).
3. ScreenDrawer uses internal information it received earlier as well as information present in the Graphics g object to determine which areas must be redrawn.
4. ScreenDrawer calls Map.getObjectsInArea(Rectangle) to determine which objects are in the areas that must be redrawn.
5. ScreenDrawer then sorts all the objects based on their object.drawPriority() values to determine which objects must be drawn on top of which other objects (crates and players on top of targets, for instance).
6. ScreenDrawer then calls object.draw(Graphics g) for each object and each object uses its own internal image to draw itself onto the Graphics object.

