Einführung in die Softwaretechnik 2018
Sheet 03

Maximilian Frühauf

5th May 2018

1. Functional Requirements for the Bumpers Game:
   (a) Start Game
   (b) Stop Game
   (c) Play Music
   (d) Steer Car
   (e) Choose Car Type
   (f) Choose Collision Type
   (g) Notify Player
   (h) Crash Car
   (i) Play Crash Sound
2. Quality Requirements for the Bumpers Game:
   Usability:
   - The types of cars should be differentiable by the player.
   - The Music should not distract the player.
   - The Crash sound should be played when two cars collide. Also it has to be loud enough to be heard with the music in the background.

   Reliability:
   - The Game can be restarted at any time.
   - It should support multiple fast user inputs in succession, without crashing.

   Performance:
   - The game can support up to 20 cars simultaneously on the screen.
   - Button click should be executed immediately.

   Supportability:
   - The game has to be modifiable by the developer to correct defects.
   - It has to be runnable on multiple different Operating Systems.
3. As-is Scenario:

   The user starts the game and is presented with the game screen. To start the game he has to click the start game button in the upper left hand corner. Now the Non-Player-Cars are moving on the game board and the player can steer his own car with a mouse click. Also some music is played.

   If two of the cars on the screen intersect with each other a crash occurs. Therefore one of the two cars gets crunched, the other can continue driving. When the only car still driving, is the player car, he has won the game. If he gets crunched while other cars are still moving he loses the game.

   Whenever the player achieves a win or loss he is notified of the event. Then the game stops and he can restart again if he wishes to.

   Visionary Scenario:

The user starts the game and is presented with the game screen. To start the game he has to click the start game button in the upper left hand corner. Now the Non-Player-Cars are moving on the game board and the player can steer his own car with a mouse click. Also some music is played.

If two cars crash the larger one wins the crash. After the crash the winning car gets the size of the losing one added to itself and becomes larger. When the player has crashed all other cars and is therefore the largest car on the screen he wins the game.

Whenever the player achieves a win or loss he is notified of the event. Then the game stops and he can restart again if he wishes to.

4. Textual Description of the Use Case Car crash:

Use Case Name	Car Crash
Participating Actors	The player car and one non-player car
Entry Conditions	The game has started and the two cars intersect each other.
Flow of Events	(a) Depending on the speed of the two cars a winner of the collision is chosen. This is the faster of the two cars. (b) The loser car gets crunched. (c) The crash sound is played. (d) A notification is printed to the console if the player car was the loser in the collision. Otherwise the NPC (non-player car) is crunched and the player continues driving in the arena.
Exit Conditions	If a winner of the collision has been determined the Condition is exited.
Special Requirements	Because the Collision is based on the speed of the two cars involved the game has to be run for a least on time step. This allows the velocities of all cars to be calculated.

5. Difference between **«extends»** and **«includes»**:

   The «includes» stereotype in a UML diagram describes functionality that is common to multiple use cases and can therefore be factored out. This allows multiple different use cases to access the same functionality.

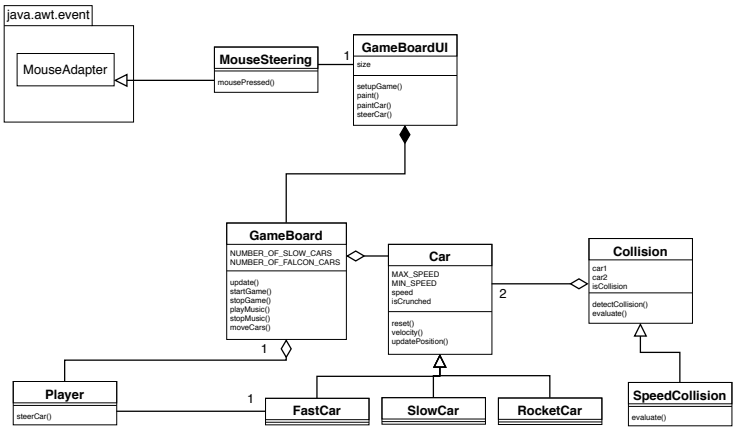
   The «extends» stereotype is used to separate functionality from a use case that is only rarely called but still has to be separate from the main use case.

   This difference can be seen in the *Start Game* use case. Here the use case *choose car type* is connected with an «extends» relationship because the player has the option to choose another car but is not forced to do so. The use case *play music* however is connected with an «includes» stereotype because the music is always played once the car starts, but it can also be used by other use cases.

6. Difference between **aggregation** and **composition**: Aggregation is used to convey that one object consists of the parts of another object.

   Composition however is a specialization of Aggregation and also imposes that the child object is dependant on the parent. Therefore if the parent is deleted, the child object has to get deleted as well.

   This concept can be seen in the UML class diagram in the GameBoard class. Here the GameBoard consists of one player, modeled by aggregation. The connection to the GameBoardUI class however is modeled as Composition, because the GameBoardUI cannot exist on it's own. The parent of GameBoard is needed to to use the GameBoardUI class.


java.awt.event
MouseAdapter
MouseSteering
mousePressed()
1
GameBoardUI
size
setupGame()
paint()
paintCar()
steerCar()
GameBoard
NUMBER_OF_SLOW_CARS
NUMBER_OF_FALCON_CARS
update()
startGame()
stopGame()
playMusic()
stopMusic()
moveCars()
1
Car
MAX_SPEED
MIN_SPEED
speed
isCrunched
reset()
velocity()
updatePosition()
2
Collision
car1
car2
isCollision
detectCollision()
evaluate()
Player
steerCar()
1
FastCar
SlowCar
RocketCar
SpeedCollision
evaluate()