# Class Description

## StudentWorld Class: GameWorld

* virtual int init();
  + It loads the current level’s map, puts everything needed on the map and initializes the player’s bonus points.
* virtual int move();
  + It updates the display text on the top of the screen, asks every actor that is alive to do something, checks whether the player is still alive or finishes the level to inform the program whether to restart the current level, end the game or load the next level, and removes every dead object after every tick.
* virtual void cleanUp();
  + After the current level ends (either because the player dies or finishes the level), it releases all allocated memory for the object on the map.
* Actor\* getContentOf(int x, int y, Actor\* self = nullptr);
  + It will **only** return the object that **blocks the actor** and will **never return the caller object itself.** This way we can know whether the player or robot can move to the next square according to the result of this method. If there are multiple objects on the same square, it will always return the kleptobot first, so that when the kleptobot is first created by the factory and on the same square as the factory, this method will always return the kleptobot. Thus the bullet can detect the kleptobot and do damage to it.
* Goodie\* getGoodie(int x, int y);
  + It will return a pointer to the goodie on the square(x, y), or nullptr if there is none. The kleptobot will use this method to determine whether there is a goodie on the square it is on and then decide whether it should pick up the goodie.
* bool playerIsOn(int x, int y);
  + If the player is on (x, y), it will return true; otherwise it will return false. The goodie class will use this method to determine whether it is picked up by the player. Also, the robot’s playerInSight() method will use it to determine whether the player is on the same row or column as the robot is.
* bool exitIsOn(int x, int y) { return (exitX == x && exitY == y); }
  + It will return whether the exit is on square (x, y). It will be called when the player tries to push a boulder onto the exit. If the exit is not revealed (i.e. m\_jewels > 0), the boulder can be pushed onto the square the exit is on. Otherwise, the boulder cannot be pushed onto the exit. (This has a slight difference with the exit behavior from the sample program the professor gave us. See detailed description in the next section).
* Player\* getPlayer() { return m\_player; }
  + The playerInSight() method will use this to get the coordinate of the player. According to different directions the player is facing, the coordinate can help to quickly determine if the player is in front of the robot (there may be obstructions in between). If the player is not in front of the player (which happens most of the case), the playerInSight method will immediately return false, thus greatly increasing the speed of checking.
* void addActor(Actor\* ptr)
  + It will add an actor to the studentworld. In this program, it will only be used to add bullet or regular/angry kleptobot. It will always add the kleptobot to the front of the list, so that when the kleptobot and factory are on the same square, the kleptobot can be detected and damaged by the bullet.
* int getNumOfJewels() { return m\_jewels; }
  + It will return the number of jewels on the map to help determine whether to show the exit or not.
* void decJewel() { m\_jewels--; }
  + It will decrease the number of jewels by 1. It will be used when a jewel is picked up by the player.
* void removeDeadGameObjects();
  + It will release the memory reserved by the dead object and remove its pointer from the list. It will be called after every tick.
* void setDisplayText();
  + It will set the display text on the top of the screen. It will be called to update the text every tick.
* void restorePlayerHealth();
  + It will restore the player’s health to full (20 hit points). It will be called when the player pick up the restore health goodie.
* void increasePlayerAmmo();
  + It will increase the player’s ammo by 20. It will be called when the player pick up an ammo goodie.
* GraphObject::Direction\* getDirections(int i) { return dirOrders[i]; }
  + It will return a set of 4 different directions in random order. All possible combinations of the 4 directions is generated when the StudentWorld class is constructed and are stored in the array called dirOrders, a 24\*4 array of directions. When the kleptobot needs to get a set of random directions, a random number i will be generated and passed into this method as the parameter. Then the method will return the ith combination of random directions. This way it can largely reduce the cost of generating a set of 4 different random directions to only O(1).

## Actor Class: GraphObject

* virtual void doSomething() = 0;
  + Every actor has to do something, and their ways of doing something differ.
* StudentWorld\* getWorld() { return m\_stdworld; }
  + Every actor needs to have access to the world they are in.
* bool checkAlive() const { return alive; }
  + Every actor needs to check if they are alive.
* void setDead(){ alive = false; }
  + Most actors can be dead.
* void move(int& x, int& y, Direction dir);
  + This will do plus or minus to the coordinate in the direction the actor is facing, but not move the actor to that square, so that we can get the square(s) in front of the actor and check what is on the square(s).
* virtual void doDamage(){}
  + Some actors can be damaged, others cannot. So the version in the base class does nothing.
* virtual bool blockActor() { return false; };
  + Some actors can block other actors, others cannot. So the version in the base class returns false.
* virtual bool moveable() { return false; }
  + This returns whether the actor can be moved by other actors. Only the boulder has such property, so the version in the base class returns false.
* virtual bool imAHole() { return false; }
  + We need to check if the actor is a hole so that a boulder can be filled in the hole. Only the hole class will return true, so in the base class it returns false.
* virtual bool imAGoodie() { return false; }
  + We need to check if the actor is a goodie so that it can be picked up by the player or stolen by the Kleptobot.
* virtual bool imARobot() { return false; }
  + We need to check if the actor is a robot so that the factory can count how many kleptobot is around it.
* virtual bool blockBullet() { return false; }
  + We need to check if the actor blocks the bullet so that when the bullet hits it, it will set itself to dead.

## Player Class: Actor

* void checkMove(Direction dir);
  + Check if the player can move in a certain direction (including pushing the boulder). If the player can move, it will move the player to the square next to him in that direction.
* virtual void doDamage();
  + It will reduce the player’s hit points and play corresponding sound or set the player to dead if his hit points is less than or equal to zero.
* int getHealth() { return m\_hitpts; }
  + The display text needs to know how many hit points the player has.
* int getAmmo() { return m\_ammo; }
  + The display text needs to know how many ammo the player has.
* void restoreHealth() { m\_hitpts = 20; }
  + The RestoreHealth goodie needs to restore the player’s health.
* void increaseAmmo() { m\_ammo += 20; }
  + The Ammo goodie needs to increase the player’s ammo.
* virtual bool blockActor() { return true; }
  + Player can block actors.
* virtual bool blockBullet() { return true; }
  + Player can block bullets.

## Wall Class: Actor

* virtual void doSomething(){}
  + Wall does nothing.
* virtual bool blockActor() { return true; }
  + Wall can block actors.
* virtual bool blockBullet() { return true; }
  + Wall can block bullets.

## Boulder Class: Actor

* virtual void doSomething(){}
  + Boulder does nothing.
* virtual void doDamage(){ m\_hitpts -= 2; if (m\_hitpts <= 0) setDead(); }
  + It will do damage to the boulder and set it dead if its hit points is less than or equal to zero.
* virtual bool moveable() { return true; }
  + The boulder is the only actor that is moveable.
* virtual bool blockBullet() { return true; }
  + The boulder can block bullets.

## Bullet Class: Actor

* virtual void doSomething();
  + It will hit the actor in its square.
* void hit(int x, int y);
  + It will check the actor(s) in the square (x, y). If the actor can be damaged, it will do damage to it. If the actor blocks the bullet, it will set it dead. Note that any actor that can be hit will also blocks the bullet. So if the bullet does damage to something, it will always be dead.

## Hole Class: Actor

* virtual void doSomething();
  + It will check if there is a boulder on it. If there is, it will set both itself and the boulder to death.
* virtual bool blockActor() { return true; }
  + A hole can block actors.
* virtual bool imAHole() { return true; }
  + To make it able to be identified by the boulder so that the boulder can be moved onto it.

## Goodie Class: Actor

* virtual void doSomething();
  + If the player is on the same square as the goodie and the goodie is not stolen by a Kleptobot, it will set itself to death, play sound and do corresponding benefits to the player.
* virtual void increasePlayerPoints() = 0;
  + It is pure virtual because every kind of goodie will increase player points by different amounts, and goodie class is the base class for all goodies.
* virtual bool imAGoodie() { return true; }
  + This will let the Kleptobot determine whether there is a goodie to steal in its square.
* void setPickability(bool pickability) { pickable = pickability; }
  + When the goodie is stolen, it cannot be picked up. When the robot dies, it needs to be pickable up again.
* bool checkPickable() { return pickable; }
  + When the player or robot is on the same square as the goodie, it needs to check if the goodie is actually pickable.

## Jewel Class: Goodie

* Although it is not a goodie, it performs quite like a goodie. So it is derived from the goodie class and inherits the goodie class’s doSomething() method.
* virtual void increasePlayerPoints();
  + It will inform the StudentWorld class to increase the player’s points by 50 and decrease the number of jewels in the map, so that the exit knows when to set itself visible.
* virtual bool imAGoodie() { return false; }
  + A jewel is not a goodie and it cannot be picked up by a kleptobot.

## TheExit Class: Actor

* virtual void doSomething();
  + It will check if it needs to reveal itself and play the reveal-exit sound when the number of jewels on the map is 0. Otherwise, it will do nothing.

## ExtraLife Class: Goodie

* virtual void increasePlayerPoints();
  + It will inform the StudentWorld class to increase the player’s points by 1000 and life by 1.

## RestoreHealth Class: Goodie

* virtual void increasePlayerPoints();
  + It will inform the StudentWorld class to increase the player’s points by 500 and restore the player’s hit points to 20.

## Ammo Class: Goodie

* virtual void increasePlayerPoints();
  + It will inform the StudentWorld class to increase the player’s points by 100 and increase the player’s ammo by 20.

## Robot Class: Actor

* virtual bool checkMove(Direction dir) = 0;
  + Every kind of robots can move and has different ways of moving.
* virtual bool blockActor() { return true; }
  + Robots can block actors.
* virtual void doDamage();
  + It will reduce the robot’s hit points by 2. If the hit points is still larger than 0, it will inform the StudentWorld class to play robot-impact sound. Otherwise it will set the robot to death, increase the player’s points accordingly, play robot-die sound and check if the robot carries any goodie. If the robot carries a goodie, it will move the goodie to the square where the robot dies, set the goodie visible and pickable. All robots perform the same when damaged.
* virtual bool blockBullet() { return true; }
  + A robot can block bullets.
* int getTicks() { return ticks; }
  + Get the robot’s ticks to compare if it needs to do something in the current tick.
* int getCurrentTick() { return currentTick; }
  + Get the robot’s ticks to compare if it needs to do something in the current tick.
* bool playerInSight();
  + Check if the player is in the robot’s sight to determine whether the robot should fire its gun (if it has).
* virtual Goodie\* getGoodieCarrying() { return nullptr;}
  + Get the goodie the robot is carrying. By default the robot carries nothing. If the robot cannot steal a goodie (snarlbots), it will always return nullptr.
* virtual void increasePlayerPoints() = 0;
  + Every kind of robots increase different amounts of player points.
* void fireGun();
  + It creates a bullet in front of the robot. It will be used by the SnarlBot and the AngryKleptoBot if the player is in front of the robot.

## SnarlBot Class: Robot

* virtual void doSomething();
  + If it can do something in the current tick, it will check if the player is in sight. Otherwise, it will move.
* virtual void increasePlayerPoints();
  + When SnarlBot dies, it will increase the player’s points by 100.
* virtual bool checkMove(Direction dir);
  + It will check the square next to the robot in the direction it is facing. If the robot is blocked, it will reverse the robot’s direction. Otherwise, it will move the robot to that square. This method will be used in the doSomething() method to move the robot if it does not fire a bullet.

## KleptoBot Class: Robot

* void randomizeDistance()
  + randomize distance before turning and set distance moved to 0.
* virtual bool imAKleptobot() { return true; }
  + Identify itself as kleptobot so that the factory knows how many kleptobots is around it.
* virtual bool checkMove(Direction dir);
  + It will check if the kleptobot can move to the square next to it in the direction the robot is facing. If there is nothing blocking the actor, it will move to that square and return true to inform the doSomething() method that the robot moves successfully. Otherwise it will return false to indicate that it fails to move.
* virtual void doSomething();
  + It will increase the current tick by 1. If the robot can do something during the current tick, it will first check if there is a goodie available to pick up in the current square. If there is and the robot gets the chance, it will pick it up, set the goodie invisible and unpickable, have a pointer pointing to the goodie it is carrying, and inform the StudentWorld class to play robot-munch sound. Otherwise, if it has not reached the distance-before-turning in the current direction, it will try to move to the next square. If it fails to move, it will choose a new distance-before-turning randomly, get a set of random directions it will try and check one by one if it can move in that direction. If all the directions fail, it will set its direction to the first direction it tries.
* virtual void increasePlayerPoints();
  + It will increase the player’s points by 10 when it dies.
* virtual void setGoodieCarrying(Goodie\* goodie) { goodieCarrying = goodie; }
  + It sets the goodie it is carrying when it steals a goodie, so that it knows what goodie it needs to restore to the map when it dies.
* virtual Goodie\* getGoodieCarrying() { return goodieCarrying; }
  + It overrides the version which always return nullptr and returns the goodie it is carrying. Because the goodieCarrying data member is set to nullptr by default, If the kleptobot does not carry any goodie, the method will still return nullptr.

## AngryKleptoBot Class: KleptoBot

* virtual void doSomething();
  + It will first check if the player is in the angrykleptobot’s sight and choose whether to fire a bullet. If the player is not in sight, the angrykleptobot will do exactly the same thing as the kleptobot.
* virtual void increasePlayerPoints();
  + It will inform the StudentWorld Class to increase the player’s points by 10 if the robot is killed.

## Factory Class: KleptoBot

* bool canCreate();
  + It will count how many kleptobots (including regular and angry) are around the factory. If the number is less than 3, it will return true to inform the factory that it can create new kleptobots. Otherwise it will return false to inform the factory that it should not create new kleptobots.
* virtual bool blockActor() { return true; }
  + A factory can block actors.
* virtual bool blockBullet() { return true; }
  + A factory can block bullets.

## KleptoBot\_Factory Class: Factory

* virtual void doSomething();
  + If the condition allows the factory to create new regular kleptobots and it also gets the chance to do so, the method will inform the StudentWorld class to add a new regular kleptobot in its square and play robot-born sound. Otherwise it will do nothing.

## AngryKleptoBot\_Factory Class: Factory

* virtual void doSomething();
  + If the condition allows the factory to create new angry kleptobots and it also gets the chance to do so, the method will inform the StudentWorld class to add a new angry kleptobot in its square and play robot-born sound. Otherwise it will do nothing.

# Design decision

The spec does not say whether the Exit should block the boulder or not. In the sample program, it does block the boulder, even when it is not revealed. This seems unreasonable to me, since when it is not revealed, the square should look as if it were empty. However the player actually cannot push a boulder on it. So I made a small change in my program to allow the player push the boulder onto the square the exit is on when the exit is not revealed. When the exit is revealed, the boulder cannot be pushed onto it, which behaves the same as the original game.

# Testing classes

## StudentWorld

I first test its init() method to see if it correctly loads the level and puts everything on the map. Then after finishing writing player and wall classes, I wrote a simple version of move() without adding the updatetext() feature to see if it correctly asks every actor on map to do something. I then complete this method bit by bit as I finish more and more actor classes to fully implement this method.   
 After the whole game is completed, I tested whether the StudentWorld correctly deduct one bonus point during every tick, adds points after the player finishes one level and re-initializes the map, bonus points and player’s ammo after loading a level (new or current level). Also I checked whether it sends the player to the next level and plays the correct sound when the player steps on the exit.

## Actor

It is an abstract class.

## Player

First I tested the pushing-boulder behavior. I controlled it to push one boulder, two or more adjacent boulders and push one boulder into a hole. Then I tested its firing bullet function and made sure the bullet is just before the actor and the sound is correct. After that I made the player be hit by bullets and see if it dies correctly. When the player is hurt, it also makes the correct sound.

## Wall

I tested whether it blocks every actor correctly, including the player, all kinds of robots, boulders and bullets.

## Boulder

I tested whether it can be damaged by bullets and filled in the hole. I also made sure that it cannot be pushed onto the same square as any other actor except the bullet and the hole.

## Bullet

I tested whether the bullets correctly do damage to robots or the player, blocked by the wall, passes through goodies, jewels, holes and other bullets.

## Hole

I tested whether it can block robots and the player, and whether it can be “filled” by the boulder.

## Goodie

It is an abstract class.

## Jewel

I tested whether it is correctly picked up by the player when the player steps on it, plays the correct sound, and increases the correct amount of score. Also, it cannot be damaged, block the robots or bullets or stolen by kleptobots.

## TheExit

I tested whether it shows up immediately after the player collects all the jewels on the map. Also it should allow robots, players, boulders and bullets to pass through it when it is not revealed and block the boulder when it is revealed.

## ExtraLife

I tested whether it is correctly picked up by the player when the player steps on it, plays the correct sound, and increases the correct amount of score and player’s total lives. Also, it cannot be damaged, block the robots or bullets, but can be stolen by kleptobots.

## RestoreHealth

I tested whether it is correctly picked up by the player when the player steps on it, plays the correct sound, increases the correct amount of score and restores the player’s hit points. Also, it cannot be damaged, block the robots or bullets, but can be stolen by kleptobots.

## Ammo

I tested whether it is correctly picked up by the player when the player steps on it, plays the correct sound, increases the correct amount of score and ammo. Also, it cannot be damaged, block the robots or bullets, but can be stolen by kleptobots.

## Robot

It is an abstract class.

## SnarlBot

I first made it able to move correctly according to the spec. Then I added the feature of shooting. However, when the player is next to the robot, it chooses to reverse direction instead of shooting. So I checked my “playerInSight()” method and found that I started checking the square two units away from the robot instead of the square next to it. I then fixed the problem, but the robot shoots the gun very quickly (i.e. it shoots every tick). So I checked again and found that when the robot shoots the gun, I forgot to increase its ticks. After this problem is solved, everything works fine.

## KleptoBot

At first the kleptobot always moves in one direction until it encounters some obstruction. I checked the program for a long time and finally found that I made a mistake in getting its current direction when moving. After this problem is fixed, I tried to block the kleptobot in three directions in the game to see whether it can find the way out, block it in all four directions to see whether it changes direction when all four directions are blocked, and observed whether it will change its direction sometimes because it has reached its distanceBeforeTurning value.

## AngryKleptoBot

I tested whether it will fire its laser gun when the player is in its sight. The other tests are the same as the Kleptobot.

## KleptoBot\_Factory

I fire a bullet towards it to see whether it will block the bullet, or whether the bullet will hit the robot if the robot is on the same square as the factory. Then I checked whether it always creates new kletptobot only under the condition that there are less than three kleptobots of either type around it. Also I made sure it produces the regular kleptobots instead of the angry ones.

## AngryKleptoBot\_Factory

I made sure it produces the angry kleptobots instead of the regular ones. The rest of the tests are the same as the KleptoBot\_Factory.