Shreyas Bharadwaj

UID: 705 342 716

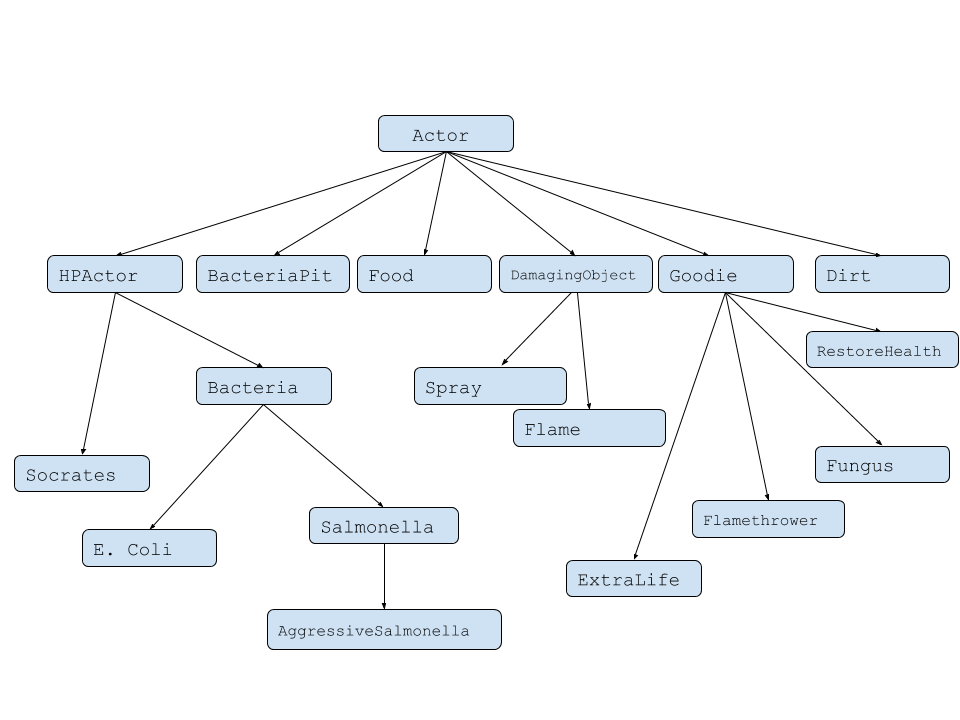
CS 32 Winter 2020

Kontagion Report

*Summary:*

The following report gives a synopsis and summary of the design and implementation decisions that went into the creation of *Kontagion*, a game involving a player, Socrates, trying to kill bacteria in a petri dish and progressing through levels after doing so. First, the general high level description of classes is given, followed by the specific member functions and the reason why they were implemented the way that they were, followed by testing examples.

*General Implementation Details*:

The highest level class in the hierarchy is Actor, a derivative of GraphObject. All objects in Kontagion are Actors, but some are derived from other subclasses of Actor. Subclasses of Actor provide some common functionality that Actor does not; for example, the DamagingObject class, a derivative of Actor, provides functionality to measure the distance travelled, in order to know whether it should disappear or not. The complete class hierarchy is shown in the diagram above, followed by descriptions and examples of each of the classes and their functionalities. 

*Actor*:

The Actor class is the base class of every kind of object in Kontagion. It provides a general functionality common to all actors, such as being able to die, returning the StudentWorld to which that specific instance of Actor belongs, and helper methods to determine whether a given Actor satisfies a certain kind of behavior.

This class was tested by trying to create objects of it, for example. Since the class is a pure virtual class, this should not be possible, and indeed, this results in a compiler error. Furthermore, the gameplay of the game itself shows in many ways that the Actor class provides the functionality it needs to

The methods from the Actor class are given below:

1. virtual void doSomething() = 0;

The doSomething()method provides an abstract definition for a common behavior across all Actors. Since the game proceeds in ticks, during which each Actor is given a chance to do something, all Actors need some kind of method to perform this. However, this method would be best declared pure virtual in Actor because there is no such thing as a general actor doing something, only a specific kind. Therefore, it is natural that any derived class would have to define and implement the doSomething() method on its own to define what its species of Actor would do during a tick. Furthermore, since there is little to no common functionality across classes in the doSomething()method, there is not much use in providing a common definition in the base class, because it would be overridden in every derived class anyway.

1. virtual void die() { aliveStatus = false; }

The die() method provides an implementation for a common functionality across all Actors; that is, dying. When dead, an Actor is removed from its corresponding StudentWorld by the move() method, and so each Actor would need a method to allow it to do so. Since there is not really a difference between one Actor dying versus another, I decided that it would be best if this class were not virtual and were simply defined as is, and overridden if there seemed to be a pressing need in any derived class.

1. virtual bool hasHP() { return false; }

This method in the Actor class exists to provide a way to determine if a given actor satisfies some property; in this case, the property that it may have hit points. The behavior of Actors that have hit points and those that do not have hit points differs in many respects, so a method that determined a common behavior to differentiate these two different types of Actors would be a good idea. It is defined to implicitly return false, so that it can be overridden if the Actor happens to have hitpoints, and the virtual nature of the function means that it will be replaced with the behavior of the subclass’s method should the method be overridden.

1. virtual bool canBeConsumed() { return false; }

This method in the Actor class exists to provide a way to determine if a given actor satisfies some property; in this case, the property that it can be consumed by other Actors; that is, food. When other Actors in the StudentWorld interact with a given Actor, it would be very useful to know whether this could be eaten and used to gain hit points or provide some other functionality or not, and having a function in the base class Actor is one way to ensure that this useful property is satisfied. The function is virtual in accordance with the nature of the function; that is, by default, it may be false, but it may be overridden to return true in the appropriate case.

1. virtual bool producesBacteria() { return false; }

This method in the Actor class exists to provide a way to determine if a given Actor produces bacteria. The behavior of an object that can do so is quite different than one that cannot, and having this function is a neat way of providing a functionality that would make interaction of objects easier.

1. virtual bool canHurtSocrates() { return false; }

This method in the Actor class exists to provide a way to determine if a given Actor can hurt the player; in other words, whether the object is a bacteria, flame, fungus, or any of the other objects in Kontagion that can cause Socrates to reduce his hit points. The behavior of an object that can do so is quite different than one that cannot, and having this function is a neat way of providing a functionality that would make interaction of objects easier. Providing this function could determine in a general case whether the overlap with Socrates is harmless, for example, or whether it would result in a decrease in hit points for him. The function is deemed virtual because that is the most natural setting; by default, all objects do not harm Socrates, but if a subclass of Actor were to provide this functionality, it could simply override and alter this method to inform the StudentWorld of its behavior.

1. virtual bool isSalmonella() { return false; }

This method in the Actor class, while not an ideal design choice because of its implementation solely to determine one kind of object, was necessary in my program in order to complete it on time, frankly speaking. Since salmonella is a more general kind of object, for which the sounds and images are separate compared to others, I hope that this is an excusable method but recognize that another should have been implemented to show more common behavior.

1. virtual bool isDamageableObject() { return false; }

This method in the Actor class exists to provide a way to determine if a given Actor can take damage from something ejected by Socrates; in other words, whether the object is a bacteria, flame, fungus, or any of the other objects in Kontagion that can be hurt by a projectile of Socrates. The behavior of an object that can do so is quite different than one that cannot, and having this function is a neat way of providing a functionality that would make interaction of objects easier. Providing this function could determine in a general case whether the overlap with a bacteria is harmless, for example, or whether it would result in a decrease in hit points for it. The function is deemed virtual because that is the most natural setting; by default, all objects cannot be damaged by Socrates’ projectiles, but if a subclass of Actor were to provide this functionality, it could simply override and alter this method to inform the StudentWorld of its behavior.

1. virtual bool isBlockingObject() { return false; }

This method in the Actor class exists to provide a way to determine if a given Actor can block movement from other actors. Although right now, this kind of object is only a Dirt, in the future, if another kind of object were to enter to game that could also block the movement of projectiles or bacteria or anything else in Kontagion, that class could simply override this method and return true as well, just like the Dirt class. For this reason, the class is implemented virtually; by default, and for most of the objects in the game, they do not block each other’s movement, so whichever kind of objects do may override this and provide the desired functionality.

1. virtual bool doesAdd() { return false; }

This method in the Actor class exists to provide a way to determine if a given Actor can be of use to Socrates’ status; that is, whether it can increase his hit points, number of sprays, and so on. Right now, only the Goodie class provides this functionality, so in that class this method is overridden and returns true instead. The purpose of making this function virtual

1. virtual void gainHP(int x) {}

This method provides a way of incrementing the hit points for an Actor. To make the implementation easier, this method was introduced in the Actor class itself, and overridden in subsequent classes where the hit points were of concern. Therefore, a common method could be called without having to worry whether the Actor had a hit points attribute or not. The function is therefore virtual, because any object that has hit points would want to override the method, while an object without the attribute has no need to change it and can inherit the method as is.

1. virtual void setHP(int x) {}

This method provides a way of setting the hit points for an Actor. To make the implementation easier, this method was introduced in the Actor class itself, and overridden in subsequent classes where the hit points were of concern. Therefore, a common method could be called without having to worry whether the Actor had a hit points attribute or not. The function is therefore virtual, because any object that has hit points would want to override the method, while an object without the attribute has no need to change it and can inherit the method as is.

1. virtual int getHP() { return 0 ;}

This method provides a way of getting the hit points for an Actor. To make the implementation easier, this method was introduced in the Actor class itself, and overridden in subsequent classes where the hit points were of concern. Therefore, a common method could be called without having to worry whether the Actor had a hit points attribute or not. The function is therefore virtual, because any object that has hit points would want to override the method, while an object without the attribute has no need to change it and can inherit the method as is.

1. StudentWorld\* getSW() const { return world; }

Since each Actor is placed in its corresponding StudentWorld, it is important that there is a way to get the StudentWorld associated with an Actor. By doing so, one could alter the StudentWorld, for example, by adding or removing objects based on an Actor’s behavior. The function is not virtual because there is no real difference in implementation of returning the associated StudentWorld between one Actor and another. The StudentWorld is also a member variable of only the Actor class and not all of its derived classes, and so keeping the function non-virtual is the most natural choice -- there would be no way to override the function in a derived class, anyway, because the class would not have access to the member StudentWorld to which it belongs.

1. bool isAlive() const { return aliveStatus; }

This method is somewhat related to the die() function from before, and merely provides a way for the StudentWorld object (and possibly other Actors) to know whether a given object is alive or not. Since all Actors share the attribute of being alive or not, it is natural to include this as a member function in the Actor class.

1. static void polarToRect(int r, Direction d, int& x, int& y);

Since the petri dish in which the game takes place is circular, I thought it would be useful to include a helper function to convert polar coordinates into rectangular coordinates. This proves useful in constructing the dirt for example, because there is no need to check whether the coordinates are within VIEW\_HEIGHT/2 and VIEW\_WIDTH/2, all that needs to be done is generate a random radius less than VIEW\_RADIUS, and any angle, and add that to the center of the petri dish’s coordinates using the polarToRect function.

1. virtual ~Actor() {}

Virtual destructor for the class, because it has classes that derive from it and their respective instances of Actor must be destroyed as well.

*HPActor:*

The HPActor class is a subclass of Actor. It represents a kind of Actor that has a hit points characteristic associated with it. The following classes are derivatives HPActors in the game: Socrates, Bacteria. By extension, the Salmonella, E. Coli, and AggressiveSalmonella classes are also derived from the HPActors and contain attributes of HPActor. The following methods are present in HPActor:

void loseHP(int x) { hp -= x; }

This method is the overridden version of the method by the same name present in the Actor class. The method is useful here because all HPActors have a member variable called hp, which can be altered. In Kontagion, only two things happen to the hit points of a character, usually: it loses some or gains some. To provide an easy way of decrementing the hit points a HPActor has, this method takes in one argument and decrements the hp variable. The method is not virtual because there is no need for it to be. Subclasses of HPActor do not have access to the hp member variable, so there is no use in overriding the method. The implementation is inline with the declaration because of the relative brevity. This doesn’t need to be a pure virtual or virtual function, because all HPActors have the same functionality and can inherit this behavior without any problems.

bool isHPActor() { return true; }

Provides a useful way of determining whether a given object is a HPActor. This is especially important because, for example, the DamagingObject class must know if the object it overlaps with has hit points to be decremented, or whether it should disappear from the game directly. Although it identifies a class directly and thus goes against ideal design principles, since this is a base class and there are never instances of HPActors themselves, I saw this as a reasonable design decision, because when adding new objects that have hit points, they can be subclasses of HPActor and inherit the functionality of this method; that is, there would be no extraneous or bloated code. This doesn’t need to be a pure virtual or virtual function, because all HPActors have the same functionality and can inherit this behavior without any problems.

void gainHP(int x) { hp += x; }

This method is the overridden version of the method by the same name present in the Actor class. The method is useful here because all HPActors have a member variable called hp, which can be altered. In Kontagion, only two things happen to the hit points of a character, usually: it loses some or gains some. To provide an easy way of incrementing the hit points a HPActor has, this method takes in one argument and increments the hp variable. The method is not virtual because there is no need for it to be. Subclasses of HPActor do not have access to the hp member variable, so there is no use in overriding the method. The implementation is inline with the declaration because of the relative brevity. This doesn’t need to be a pure virtual or virtual function, because all HPActors have the same functionality and can inherit this behavior without any problems.

void setHP(int x) { hp = x; }

This method is the overridden version of the method by the same name present in the Actor class. The method is useful here because all HPActors have a member variable called hp, which can be altered. In Kontagion, only two things happen to the hit points of a character, usually: it loses some or gains some. To provide an easy way of setting the hit points a HPActor has, this method takes in one argument and sets the hp variable. The method is not virtual because there is no need for it to be. Subclasses of HPActor do not have access to the hp member variable, so there is no use in overriding the method. The implementation is inline with the declaration because of the relative brevity. This doesn’t need to be a pure virtual or virtual function, because all HPActors have the same functionality and can inherit this behavior without any problems.

int getHP() { return hp; }

This method overrides a method by the same name in the Actor class, but applies it in a more relative context. Provides the functionality to get the hit points of a given HPActor. In Kontagion, this functionality is useful because of the many situations in which the number of hit points of an Actor is relevant. For example, if the hit points of a given HPActor go below 0, the actor must die and be removed from the game. In order to determine this, the StudentWorld instance would have to know the number of hit points the actor currently has, and so this function provides that information. This doesn’t need to be a pure virtual or virtual function, because all HPActors have the same functionality and can inherit this behavior without any problems.

int getMovementPlanDistance() { return movement\_plan\_distance; }

Similar to the getHP() function in the Actor class, this method provides a functionality to return the movement plan of an HPActor, an attribute that is present in only the Salmonella class. This is to make the implementation of the game easier, and to ensure that all HPActors can be called with a common method, in accordance with design principles regarding class hierarchy. The downside may be that, in some client function, an uninitialized movement plan distance might be called unintentionally, but this is not something that would happen for the foreseeable future. This doesn’t need to be a pure virtual or virtual function, because all HPActors have the same functionality and can inherit this behavior without any problems.

void setMovementPlanDistance(int a){ movement\_plan\_distance = a;}

Similar to the setHP() function in the Actor class, this method provides a functionality to set the movement plan of an HPActor, an attribute that is present in only the Salmonella class. This is to make the implementation of the game easier, and to ensure that all HPActors can be called with a common method, in accordance with design principles regarding class hierarchy. The downside may be that, in some client function, an uninitialized movement plan distance might be set unintentionally, but this is not something that would happen for the foreseeable future. Furthermore, the damage this does to the overall gameplay is minimal and not something to worry about for our purposes. This doesn’t need to be a pure virtual or virtual function, because all HPActors have the same functionality and can inherit this behavior without any problems.

bool isDamageableObject() { return true; }

This method overrides a method of the same name in the Actor class to determine if an object can be hurt; that is, anything other than a dirt or a fungus, pretty much. The specifics of this function are in the Actor section, but it is overridden because all HPActors can be hurt, and this provides that common attribute, which can be checked for whenever needed. This doesn’t need to be a pure virtual or virtual function, because all HPActors have the same functionality and can inherit this behavior without any problems.

*BacteriaPit*

The BacteriaPit class represents a type of object that is different enough from every other type of object to warrant its own class that is a derivative of Actor. The BacteriaPit class, broadly speaking, produces bacteria based on the level. There are a predetermined number of these objects in each level of the game in random locations. The following methods belong to the BacteriaPit class and aid with its functionality:

void doSomething();

This method is the method called during each tick, when every actor in the StudentWorld is told to do something. In accordance with the specification document, this method, for the BacteriaPit class, produces a selection of bacteria based on random number generation. The function makes use of a member array of integers that keeps track of how many of each type of bacteria the pit has, and decrements the corresponding once a bacteria of a certain type is added to the game. Since there are no subclasses of the BacteriaPit class, there was no need to make this function virtual or pure virtual.

bool isPit() { return true; }

This function is an identifier function to identify a pit. Although this was not an ideal design choice, it made some aspects of the implementation much easier, including trying to determine when a given level had ended, which is true if and only if all the bacteria and the bacteria pits are destroyed. Since there are no subclasses of the BacteriaPit class, there was no need to make this function virtual or pure virtual.

int getRegSalmonella() { return bacteria[0]; }

Function that returns how many regular salmonella are in the bacteria pit at a given time. Since there are no subclasses of the BacteriaPit class, there was no need to make this function virtual or pure virtual.

int getAggSalmonella() { return bacteria[1]; }

Function that returns how many aggressive salmonella are in the bacteria pit at a given time. Since there are no subclasses of the BacteriaPit class, there was no need to make this function virtual or pure virtual.

int getEColi() { return bacteria[2]; }

Function that returns how many E. Coli are in the bacteria pit at a given time. Since there are no subclasses of the BacteriaPit class, there was no need to make this function virtual or pure virtual.

void decBacteria(int a);

Function that decrements the number of bacteria according to the code a; that is, which member in the array it is. If a is 0, the number of regular salmonella will be decremented, and so on. This is useful in the doSomething() method, in order to keep track of the remaining bacteria in the pit.

*Food*:

Food is a relatively simple class that does not bear much resemblance with any other type of Actor. It is represented by a pizza image in the game, and can be eaten by bacteria but not damaged by Socrates. The methods of the Food class are given below

void doSomething() { return; }

Since the Food class is the only class besides the Dirt class that doesn’t do anything when asked to doSomething(), this method’s relatively simple inline implementation simply returns, as it should. There is no need to add any kind of virtual or pure virtual functionality to this because the class doesn’t have any descendants.

bool isFood() { return true; }

Identifier function that allows bacteria to identify when they are overlapping with a food, so it can be eaten. There is no need to add any kind of virtual or pure virtual functionality to this because the class doesn’t have any descendants, and all food isFood().

*DamagingObject:*

The DamagingObject class consists of objects that damage; namely, sprays and flames. Since other than in image and in moving behavior, these two classes behave very similarly, I thought it would be a good design choice to include a class solely for objects that can damage other ones, and extend flames and sprays from it. The following are methods included in this class:

int getTravelDistance() { return max\_travel\_distance; }

This method is an accessor method that provides access to the travel distance of the given instance of a DamagingObject. Both flames and sprays have a set distance after which they will cease to exist. Therefore, this common method provides the common functionality for objects of type DamagingObject to know their travel distance, which may change after they move in the game. There is no need to make this virtual, since derivative classes of the DamagingObject do not have access to their travel distance as a member variable, and can only access it through these methods.

void setTravelDistance(int x) { max\_travel\_distance = x; }

This method is an setting method that provides access to the travel distance of the given instance of a DamagingObject. Both flames and sprays have a set distance after which they will cease to exist, which must be dynamically set every time the object moves.. Therefore, this common method provides the common functionality for objects of type DamagingObject to set their travel distance, which may change after they move in the game.There is no need to make this virtual, since derivative classes of the DamagingObject do not have access to their travel distance as a member variable, and can only access it through these methods.

void doSomething();

The doSomething() method in the DamagingObject class provides a common functionality for all classes that derive from it, because they have such similar behavior. This avoids the need to implement separate doSomething() methods for each class. There is no need to make this virtual because it can simply be called in all its derivative classes’ and produce the correct behavior for the given class.

virtual bool isSpray() { return false; }

Identifier function that allows bacteria to identify a DamagingObject as a spray, in the case when the doSomething() method is applied. The function is virtual because in the derivative class Spray, it will be modified to return true, and the Flame class will continue to return false and behave appropriately.

virtual bool isFlame() { return false; }

Identifier function that allows bacteria to identify a DamagingObject as a flame, in the case when the doSomething() method is applied. The function is virtual because in the derivative class Flame, it will be modified to return true, and the Spray class will continue to return false and behave appropriately.

*Goodie*:

The Goodie is the parent class for every kind of Goodie: the ExtraLife, Flamethrower, Fungus, and RestoreHealth. It may be confusing that a Fungus is also a kind of Goodie, but in the implementation, both regular Goodies as well as Fungi behave very similarly and hence were included under the same parent class. The following are methods from the Goodie class:

virtual void doSomething();

The redefinition of the doSomething() within the Goodie class provides a common functionality for all Goodies as well as specialization based on each type of Goodie. Each goodie, for example, adds a given amount of points, whether positive or negative, to Socrates, as well as influencing his attributes. Thus, they are very similar, and the common aspects can be taken care of with the virtual function, while the specializations can be redefined in subclasses of Goodie.

bool isGoodie() { return true; }

Accessor function that allows other actors and StudentWorld to know when the given Actor is a goodie, due to various behavior that are unique to Goodies. This does not need to be virtual because all goodies are goodies and will return true regardless of what type they are.

int getLifetime(){ return lifetime; }

Accessor function that allows other actors and StudentWorld to know the lifetime of the Goodie. This does not need to be virtual because all goodies have lifetimes and moreover cannot return the member variable lifetime which they don’t have access to.

void setLifetime(int x) { lifetime = x; }

Setting function that allows other actors and StudentWorld to set the lifetime of a given goodie. This does not need to be virtual because all goodies are goodies and will return true regardless of what type they are.

virtual bool isRestoreHealth() { return false; }

Identifier function that allows other actors and StudentWorld to know whether a given goodie is a RestoreHealth goodie. This does need to be virtual because all goodies are not RestoreHealth goodies.

virtual bool isFlamethrower() { return false; }

Identifier function that allows other actors and StudentWorld to know whether a given goodie is a Flamethrower goodie. This does need to be virtual because all goodies are not Flamethrower goodies.

virtual bool isExtraLife() { return false; }

Identifier function that allows other actors and StudentWorld to know whether a given goodie is a ExtraLife goodie. This does need to be virtual because all goodies are not ExtraLife goodies.

virtual bool isFungus() { return false; }

Accessor function that allows other actors and StudentWorld to know whether a given goodie is a Fungus. This does need to be virtual because all goodies are not Fungi.

bool isDamageableObject() { return true; }

Full description of the method is given in the Actor class, but this is merely a redefinition because all Goodies are damageable. It does not need to be virtual, because every kind of Goodie is damageable.

virtual ~Goodie() {}

Virtual destructor because the class has children.

*Dirt*:

The Dirt class is a very simple class that does not do much besides block the movement of bacteria. In this respect, it is different from every other kind of Actor, and hence has its own subclass under Actor. The following methods describe the functionality of the Dirt class:

void doSomething() { return; }

Since the Dirt class is the only class besides the Food class that doesn’t do anything when asked to doSomething(), this method’s relatively simple inline implementation simply returns, as it should. There is no need to add any kind of virtual or pure virtual functionality to this because the class doesn’t have any descendants.

bool isDirt() { return true; }

Identifier method that returns true when a dirt is found. This is relevant because Dirts are the only kind of objects that are allowed to overlap when init() is called, and are the only objects that can block the movement of bacteria.

bool isDamageableObject() { return true; }

Full description of the method is given in the Actor class, but this is merely a redefinition because all Dirt is damageable. It does not need to be virtual, because there are no subclasses of Dirt.

*Bacteria*:

The bacteria class is a general class that was created to be a parent to all bacteria. The class is described by the following methods:

virtual void doSomething();

The redefinition of the doSomething() within the Bacteria class provides a common functionality for all Bacteria as well as specialization based on each type of Bacteria. Each goodie, for example, can hurt Socrates, as well as influencing his attributes. Thus, they are very similar, and the common aspects can be taken care of with the virtual function, while the specializations can be redefined in subclasses of Bacteria.

virtual bool isBacteria() { return true; }

Identifier function that serves the purpose of having the StudentWorld instance know when an object is a bacteria.

virtual bool isSalmonella() { return false; }

Identifier function that serves the purpose of having the StudentWorld instance know when an object is a Salmonella. This method is virtual so that the Salmonella class can override it and return true.

virtual bool isAggressiveSalmonella() { return false; }

Identifier function that serves the purpose of having the StudentWorld instance know when an object is an AggressiveSalmonella. This method is virtual so that the AggressiveSalmonella class can override it and return true.

virtual bool isEColi() { return false; }

Identifier function that serves the purpose of having the StudentWorld instance know when an object is an E. Coli. This method is virtual so that the E. Coli class can override it and return true.

bool isDamageableObject() { return true; }

Full description of the method is given in the Actor class, but this is merely a redefinition because all Bacteria is damageable. It does not need to be virtual, because there are no subclasses of Dirt.

void die();

Full description of the method is given in the Actor class, but this is merely a redefinition because Bacteria make a noise when they die. This does not need to be virtual because all the specialization is taken care of in the method itself, making the appropriate noise when each type of bacteria dies.

int getFoodEaten() { return foodEaten; }

Accessor function that allows other actors and StudentWorld to know the amount of food eaten by the Bacteria. This does not need to be virtual because all Bacteria eat food and also do not have access to the member variable foodEaten, which they can access only through methods of this class.

void setFoodEaten(int x) { foodEaten = x; }

Setting function that allows other actors and StudentWorld to know the amount of food eaten by the Bacteria. This does not need to be virtual because all Bacteria eat food and also do not have access to the member variable foodEaten, which they can access only through methods of this class.

*Socrates*:

Socrates is the main actor of the game. His behavior is controlled by the player of the game, and described by the functions below:

void doSomething();

Implements the doSomething() functionality of Socrates, allowing him to be controlled by the player and move around the petri dish, as well as allowing him to shoot sprays and flames at the bacteria. This function doesn’t need to be virtual because there are no subclasses of Socrates.

void changeSpray(const int& x) { spray += x; }

Mutator method that changes the number of sprays that Socrates has, when he shoots them or dies for example. This function doesn’t need to be virtual because there are no subclasses of Socrates.

void changeFlame(const int& x) { flame += x; }

Mutator method that changes the number of flames that Socrates has, when he shoots them or dies for example. This function doesn’t need to be virtual because there are no subclasses of Socrates.

bool isSocrates() const { return true; }

Identified method to know when a given Actor is Socrates, to inform him of damage or death. This function doesn’t need to be virtual because there are no subclasses of Socrates.

int getSpray() const { return spray; }

Accessor method that returns the number of sprays that Socrates has, when he shoots them or dies for example. This function doesn’t need to be virtual because there are no subclasses of Socrates.

int getFlame() const { return flame; }

Accessor method that returns the number of flames that Socrates has, when he shoots them or dies for example. This function doesn’t need to be virtual because there are no subclasses of Socrates.

~Socrates() {}

Empty destructor because there are no dynamically allocated variables. This destructor doesn’t need to be virtual because there are no subclasses of Socrates.

*Salmonella*:

Salmonella is a specific type of bacteria that has subclasses including AggressiveSalmonella. The following methods describe the class:

bool isSalmonella() { return true; }

Identifier method that returns true when a given Bacteria is a Salmonella. Does not need to be virtual because all Salmonella are Salmonella.

virtual void doSomething();

Redefinition of the doSomething() method to include common behaviours of Salmonella. This method needs to be virtual so that AggressiveSalmonella can include its own separate behaviors in addition to the ones provided by the regular Salmonella.

*E. Coli*:  
 E. Coli are a specific kind of bacteria. Its behavior is described by the following methods.

void doSomething();

The doSomething() behavior of the E. Coli bacteria is mostly implemented in the StudentWorld class, because it needs access to the other Actors in the StudentWorld. However, it is called from here. This does not need to be virtual because there are no subclasses of E. Coli.

bool isEColi() { return true; }

Identifier method that returns true when a given Bacteria is a E. Coli. Does not need to be virtual because there are no subclasses of E. Coli.

*Spray*:

The spray is a type of damaging object. It is described by the following methods:

bool isSpray() { return true; }

Returns whether a given object is a spray or not, useful in the context to determine damage to a given object. Does not need to be virtual because there are no derived classes from the Spray class.

*Flame*:

The flame is a type of damaging object. It is described by the following methods:

bool isFlame() { return true; }

Returns whether a given object is a flame or not, useful in the context to determine damage to a given object. Does not need to be virtual because there are no derived classes from the Flame class.

*RestoreHealth*:

The RestoreHealth restores Socrates to his full 100 hit points. It is described by the following methods:

bool isRestoreHealth() { return true; }

Returns whether a given object is a RestoreHealth or not, useful in the context to determine damage to a given object. Does not need to be virtual because there are no derived classes from the RestoreHealth class.

*Flamethrower*:

The RestoreHealth restores Socrates to his full flame amount as well as adding some amount of points. It is described by the following methods:

bool isFlamethrower() { return true; }

Returns whether a given object is a RestoreHealth or not, useful in the context to determine damage to a given object. Does not need to be virtual because there are no derived classes from the RestoreHealth class.

*Extra Life*:

The RestoreHealth Goodie gives Socrates an extra life. It is described by the following methods:

bool isExtraLife() { return true; }

Returns whether a given object is a RestoreHealth or not, useful in the context to determine damage to a given object. Does not need to be virtual because there are no derived classes from the RestoreHealth class.

*Fungus*:

The Fungus decreases Socrates’ hit points as well as losing points. It is described by the following methods:

bool isFungus() { return true; }

Returns whether a given object is a RestoreHealth or not, useful in the context to determine damage to a given object. Does not need to be virtual because there are no derived classes from the RestoreHealth class.

Known bugs:  
 While playing the game, I could not find any bugs.

Assumptions:

One assumption I made was letting my Dirt locations be put randomly radially around the dish. That is, they were computed to be random in polar coordinates. Since it was not specified that this should not be the case, and the method for doing so was much easier, I decided to do this due to the simplicity. This causes clustering toward the center of the dish, because the smaller radii will look much more centered. Other than that, I did not make any assumptions for the game and followed the specifications. Some of the methods in my program do have redundant code, and if I had more time on this project, I would choose to cut out these redundant parts.