Joseph Lewis

Project Proposal:

Make a SimAnt clone. Sim Ant was a game developed in 1991 by Maxis which was part simulator part game. There were two ant colonies in the game (black and red) your job as an ant in the black colony was to eventually beat out the red colony by either starving it or killing their queen; while keeping the same things from being done to you. You also have to keep your colony alive by feeding and protecting it.

This would be a somewhat simplified version (no scenarios, and limited kinds of ants). The interface will be written in Swing. The red ant colony will be entirely autonomous and run without a real central intelligence (similar in the way real ant colonies behave).

Update 2011-05-10:

As the code currently stands:

```
-------------------------------------------------------------------------------
Language               files          blank          comment       code
-------------------------------------------------------------------------------
Java                           31             607            1045                 2126
HTML                           1              2              0             22
-------------------------------------------------------------------------------
SUM:                           32             609            1045                 2148
-------------------------------------------------------------------------------
```

Meaning there are a total number of 3802 lines of code.

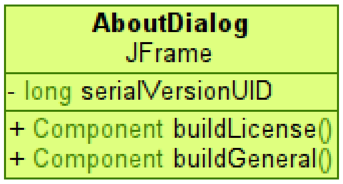
The finished product is at: http://cs.du.edu/~joslewis and will stay there for a while.

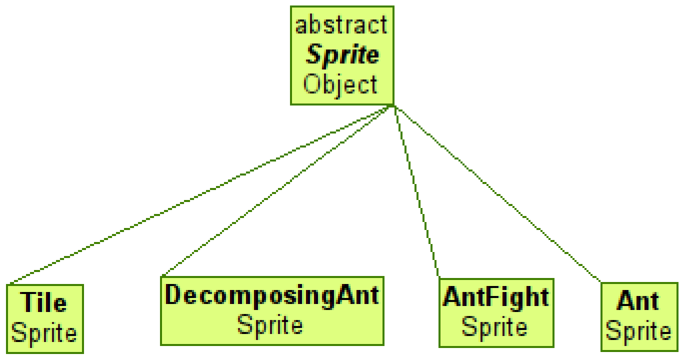
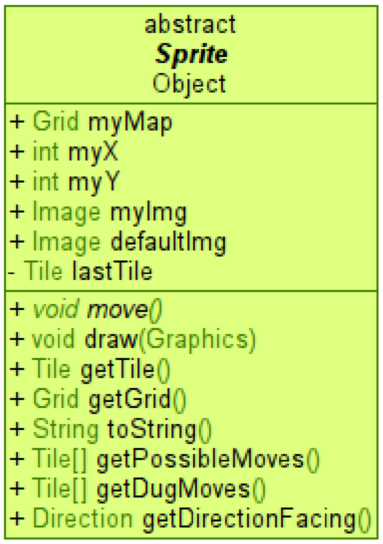
Class names, descriptions, and notable methods:

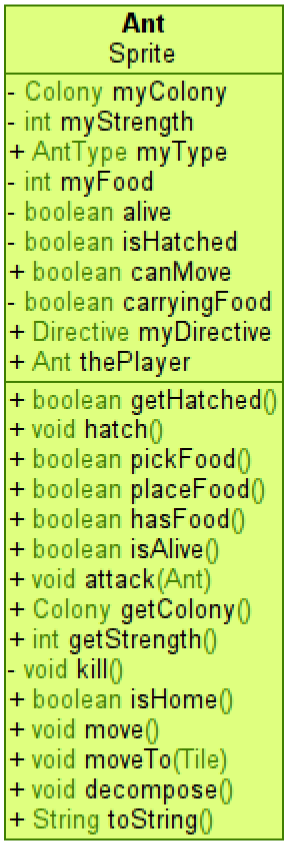
Enums	AntType - Kinds of ants (Queen, Worker, etc.) GridType - Kind of maps (Surface, Red Nest, Black Nest) TileType - Kinds of tiles (Underground, Tunnel, etc.) *not used* ImageType - Images used in the game. Direction - The directions on the compass.

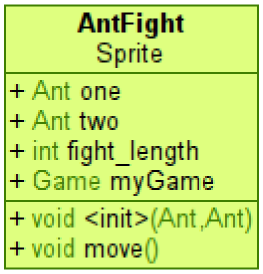
Applet	This class constructs a new Applet, with a new MainWindowContainer as the contents. This is the public static void main(String[] args) of the JApplet world.

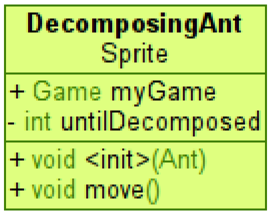
SwingSet	This class constructs a new JFrame, with a new MainWindowContainer as the contents. It has a public static void main(String[] args). This class should be set to be the one run in the manifest of any runnable Jar file.

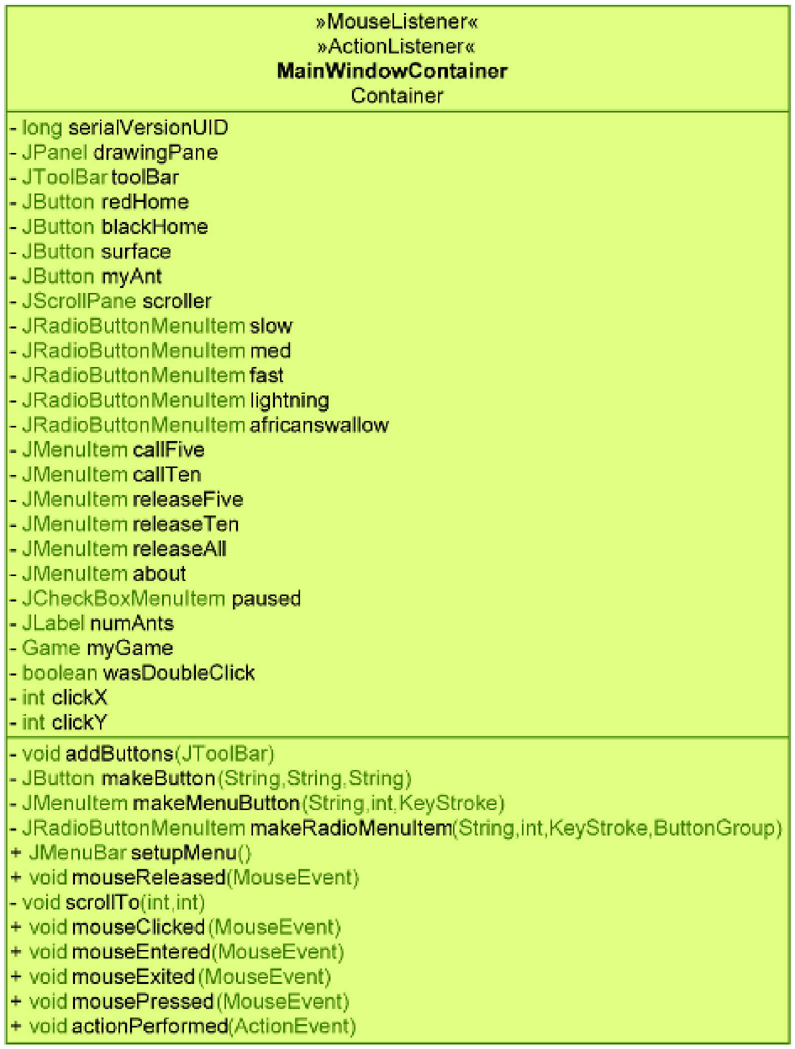
AboutDialog	This class is a JFrame that shows standard information about the program.  AboutDialog JFrame - long serialVersionUID + Component buildLicense() + Component buildGeneral() *Notable Methods:* buildLicense() reads the GPL from a text file in the Jar.

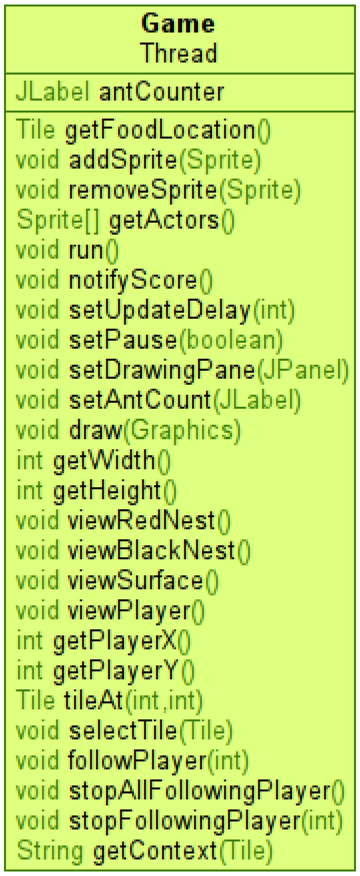
Sprite	This class represents anything on screen that has an X Y and a grid it belongs to. Multiple classes extend sprite:  abstract Sprite Object Tile Sprite DecomposingAnt Sprite AntFight Sprite Ant Sprite  abstract Sprite Object + Grid myMap + int myX + int myY + Image myImg + Image defaultImg - Tile lastTile + void move() + void draw(Graphics) + Tile getTile() + Grid getGrid() + String toString() + Tile[] getPossibleMoves() + Tile[] getDugMoves() + Direction getDirectionFacing() *Notable Methods:* draw() Every sprite has a draw method, which will place the Sprite's myImg in the proper location on the Graphics component given. move() This method must be overridden by the class extending Sprite, it is responsible for moving x and y. getDirectoionFacing() This method remembers where the sprite was last, returns the Direction its current location is compared to its last one.

Ant	This class represents a standard ant which make up most of the characters in the game; an ant is simply a sprite.  Ant Sprite - Colony myColony - int myStrength + AntType myType - int myFood - boolean alive - boolean isHatched + boolean canMove - boolean carryingFood + Directive myDirective + Ant thePlayer + boolean getHatched() + void hatch() + boolean pickFood() + boolean placeFood() + boolean hasFood() + boolean isAlive() + void attack(Ant) + Colony getColony() + int getStrength() - void kill() + boolean isHome() + void move() + void moveTo(Tile) + void decompose() + String toString() *Notable Methods:* move() If this ant has a directive, run it, if it has no directive get a new one through the DirectiveFactory. If there is an enemy on the tile this ant wants to move to, it sets up a new AntFight attack() Attacks another Ant, strength matters, although it only weighs the probability of winning. decompose() When the ant is dead, this method is called, causing the Ant to be replaced on the screen with a DecomposingAnt

AntFight	This class represents two ants that are fighting. It is constructed with two ants, it then removes the two from the move() sequence of the main class, and adds itself. It will wait for a predetermined number of rounds to complete, then fight the two ants returning the victor to the list of Sprites to move every turn, and finally remove itself. AntFight makes sure that Ants don't seemingly kill each other as they walk past one another, by providing a nice "fighting" image and a time delay.  AntFight Sprite + Ant one + Ant two + int fight_length + Game myGame + void <init>(Ant,Ant) + void move()

DecomposingAnt	This class represents an ant that is dying. It politely removes the given Ant from the list of Sprites to move in the game (keeping it from walking around zombified) and inserts itself in there where it will display an image of a dead ant, for a predetermined number of turns. After those turns are up, it removes itself from the list of sprites to be updated and is swept up by the great GarbageCollector.  DecomposingAnt Sprite + Game myGame - int untilDecomposed + void <init>(Ant) + void move()

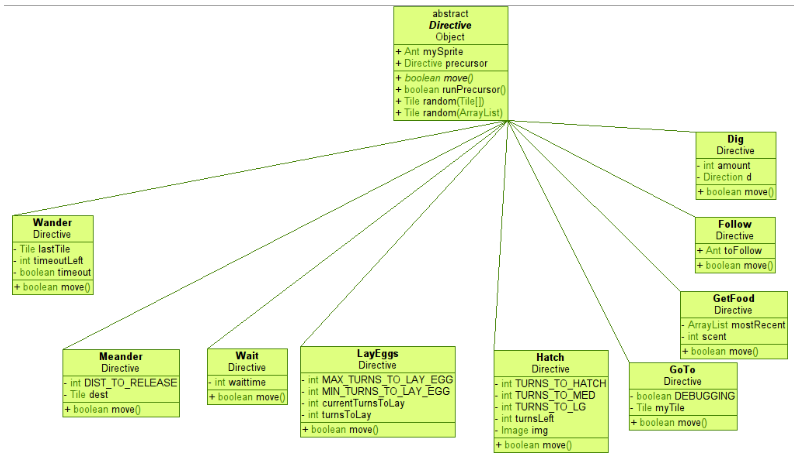
MainWindowContainer	This class is a standard Container, quite simply an object that fits in to JApplets and JFrames, it allows the game to be ported to either an Applet or desktop application very quickly and easily. Once constructed, the MainWindowContainer creates a new Game instance, as a daemon thread; that way the game logic and AWT don't mess with each other. »MouseListener« »ActionListener« **MainWindowContainer** Container - long serialVersionUID - JPanel drawingPane - JToolBar toolBar - JButton redHome - JButton blackHome - JButton surface - JButton myAnt - JScrollPane scroller - JRadioButtonMenuItem slow - JRadioButtonMenuItem med - JRadioButtonMenuItem fast - JRadioButtonMenuItem lightning - JRadioButtonMenuItem africanswallow - JMenuItem callFive - JMenuItem callTen - JMenuItem releaseFive - JMenuItem releaseTen - JMenuItem releaseAll - JMenuItem about - JCheckBoxMenuItem paused - JLabel numAnts - Game myGame - boolean wasDoubleClick - int clickX - int clickY - void **addButtons**(JToolBar) - JButton **makeButton**(String,String,String) - JMenuItem **makeMenuButton**(String,int,KeyStroke) - JRadioButtonMenuItem **makeRadioMenuItem**(String,int,KeyStroke,ButtonGroup) + JMenuBar **setupMenu**() + void **mouseReleased**(MouseEvent) - void **scrollTo**(int,int) + void **mouseClicked**(MouseEvent) + void **mouseEntered**(MouseEvent) + void **mouseExited**(MouseEvent) + void **mousePressed**(MouseEvent) + void **actionPerformed**(ActionEvent) *Notable Methods:* The constructor's sole pparameteris a RootWindow object, which can either be a JApplet or a JFrame, both of which support setting titles, and menubars.

Game	This class represents an entire CloneAnt game, save for the interface. It also provides a number of handy methods that the interface can call to change the state of the game (otherwise the game would just play itself). Once constructed, it sets up the two colonies, and the world. When started (it is a thread) and the screen becomes available a main loop executes, moving/updating all sprites and tiles, and drawing them to the screen. This continues to loop pausing for intervals in between so as to not play an entire game in a few seconds, until either the red or black nest says it is dead. Once that happens, the user is notified and the main loop is exited ending the game.  Game Thread JLabel antCounter Tile getFoodLocation() void addSprite(Sprite) void removeSprite(Sprite) Sprite[] getActors() void run() void notifyScore() void setUpdateDelay(int) void setPause(boolean) void setDrawingPane(JPanel) void setAntCount(JLabel) void draw(Graphics) int getWidth() int getHeight() void viewRedNest() void viewBlackNest() void viewSurface() void viewPlayer() int getPlayerX() int getPlayerY() Tile tileAt(int,int) void selectTile(Tile) void followPlayer(int) void stopAllFollowingPlayer() void stopFollowingPlayer(int) String getContext(Tile) *Notable Methods:* setDrawingPane() - Sets the JPanel that images are to be drawn to.

Directive

This abstract class defines an action that an Ant can perform. When an Ant is directed to move() it passes the buck to its directive.

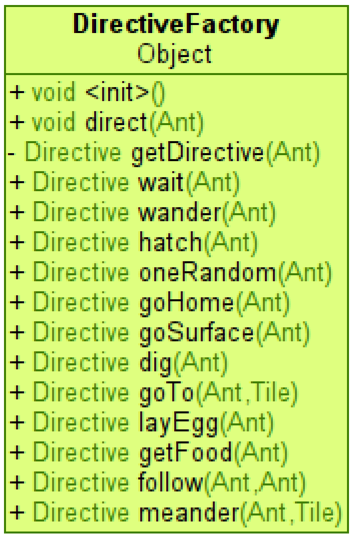
Directive has nine subclasses:

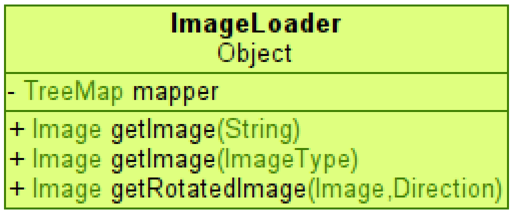


(better quality version at end of document)

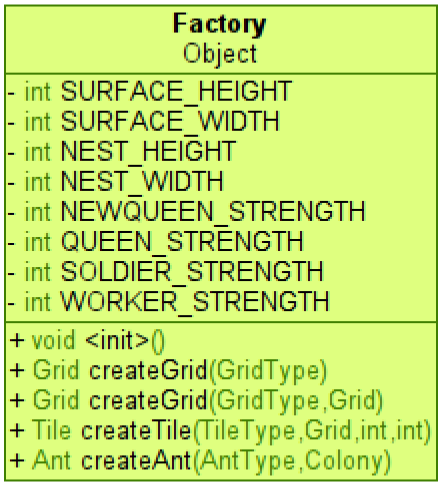
- Wander - Move randomly
- Meander - Move semi-randomly toward a target.
- Wait - Do nothing.
- LayEggs - Lay some eggs every so often (queens only)
- Hatch - Become an egg until you are ready to burst out.
- GoTo - Go to a specific tile.
- GetFood - Go find food on the surface.
- Follow - Stalk an Ant.
- Dig - Go home and make some room!

While these directives on their own would be great, larger chains of directives may be chained together to create “actions”. Directives may have “precursors” which are Directives that are run until they quit before a Directive itself is executed. An action may be something like “go kill the red queen”, to kill the red queen, you must follow it, but to get to following position you must go to the queen’s nest, but to get there you must leave your own. Therefore a chain of three simple directives can create a complex action.

DirectiveFactory	This class creates custom directives simply. Some of the directives in here don’t actually exist as real directives, but are rather constructed using precursors.  **DirectiveFactory** Object + void <init>() + void direct(Ant) - Directive getDirective(Ant) + Directive wait(Ant) + Directive wander(Ant) + Directive hatch(Ant) + Directive oneRandom(Ant) + Directive goHome(Ant) + Directive goSurface(Ant) + Directive dig(Ant) + Directive goTo(Ant,Tile) + Directive layEgg(Ant) + Directive getFood(Ant) + Directive follow(Ant,Ant) + Directive meander(Ant,Tile) *Notable Methods:* direct(Ant) - Gathers a bunch of information about an ant and its home, then decides what the Ant should do.

ImageLoader	This class loads images, and serves as a cache too, so images aren’t loaded more than once (a 100 by 50 grid means 5000 images of the same piece of dirt being loaded, at 2K a piece that is 100MB of data to be loaded, and possibly stored in RAM). The caching ability alone cut startup time from well over twenty seconds down to about three.  **ImageLoader** Object - TreeMap mapper + Image getImage(String) + Image getImage(ImageType) + Image getRotatedImage(Image,Direction) *Notable Methods:* getRotatedImage() rotates the given image so it is facing the given direction (don’t worry these are definitely cached); this means there need only be one sprite of each kind, and the rest can just be made.

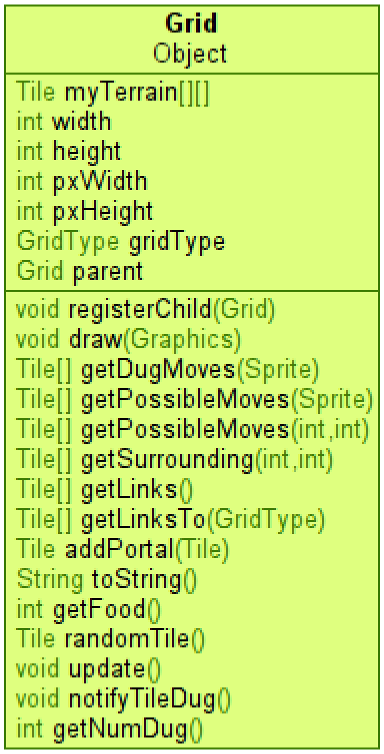
Factory	A simple multi-purpose factory. Has the ability to create Grids,

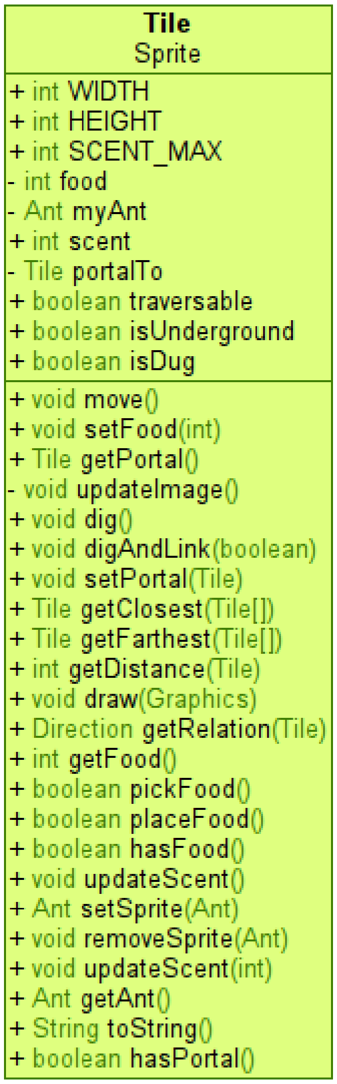
	Tiles, and Ants. 

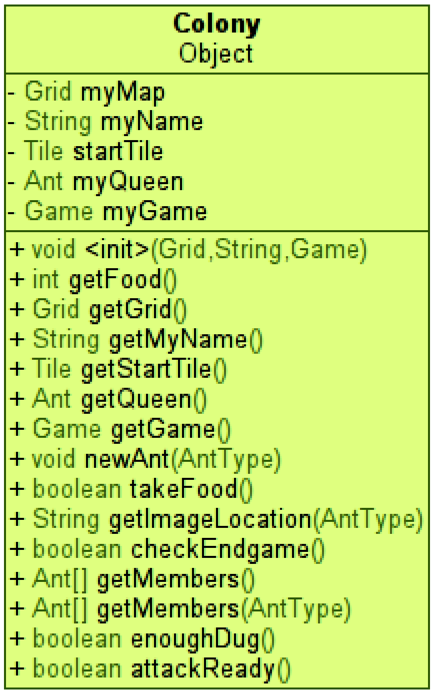
Factory
Object

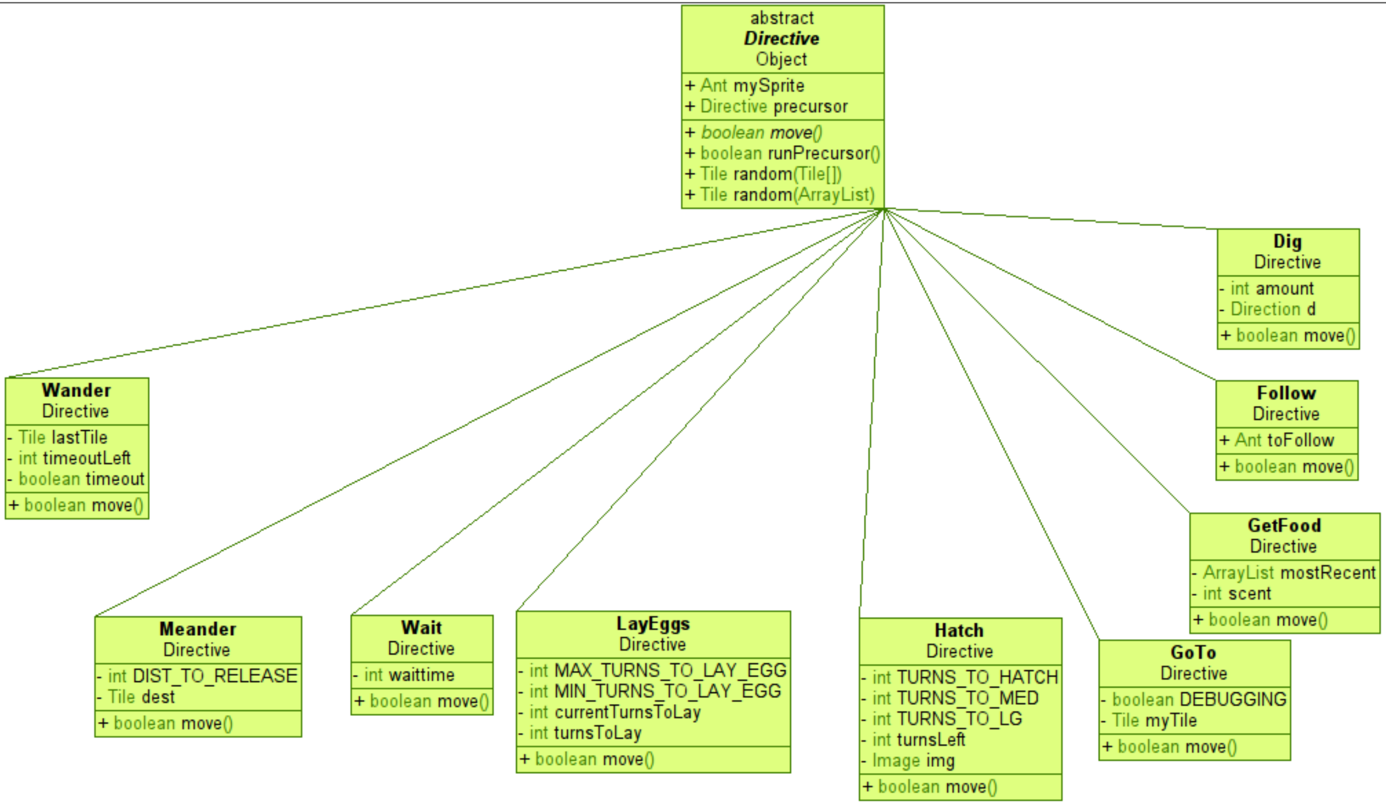
- int SURFACE_HEIGHT
- int SURFACE_WIDTH
- int NEST_HEIGHT
- int NEST_WIDTH
- int NEWQUEEN_STRENGTH
- int QUEEN_STRENGTH
- int SOLDIER_STRENGTH
- int WORKER_STRENGTH

+ void <init>()
+ Grid createGrid(GridType)
+ Grid createGrid(GridType,Grid)
+ Tile createTile(TileType,Grid,int,int)
+ Ant createAnt(AntType,Colony)

Grid	This class represents a piece of land in the world. It is essentially a holder for a 2D array of Tiles, and has many methods that allow the querying of those tiles.  Grid Object Tile myTerrain[][] int width int height int pxWidth int pxHeight GridType gridType Grid parent void registerChild(Grid) void draw(Graphics) Tile[] getDugMoves(Sprite) Tile[] getPossibleMoves(Sprite) Tile[] getPossibleMoves(int,int) Tile[] getSurrounding(int,int) Tile[] getLinks() Tile[] getLinksTo(GridType) Tile addPortal(Tile) String toString() int getFood() Tile randomTile() void update() void notifyTileDug() int getNumDug() *Notable Methods:* notifyTileDug() - When a tile in this grid is dug, it needs to notify the grid, so the grid doesn't need to use two for loops and a bunch of iterations to count the number of tiles dug. Again, if a grid is 50x100, that means there are 5000 tiles; too slow to do two for loops.

Tile	This class represents the smallest division of land available. It can be dug (if underground) hold food, hold scent (which an Ant leaves while carrying food to alert its companions that food is the way from whence it came), and can be a portal to another map (surface to a nest). It also knows about the existence of an Ant on top of it, therefore when two ants of different colonies try to step there, all they need to do is check if another exists, and if it does, fight it. A slew of convenience methods are built in, which can tell the closest and farthest tiles from this one in a list of tiles (which helps Ants navigate).  Tile Sprite + int WIDTH + int HEIGHT + int SCENT_MAX - int food - Ant myAnt + int scent - Tile portalTo + boolean traversable + boolean isUnderground + boolean isDug + void move() + void setFood(int) + Tile getPortal() - void updateImage() + void dig() + void digAndLink(boolean) + void setPortal(Tile) + Tile getClosest(Tile[]) + Tile getFarthest(Tile[]) + int getDistance(Tile) + void draw(Graphics) + Direction getRelation(Tile) + int getFood() + boolean pickFood() + boolean placeFood() + boolean hasFood() + void updateScent() + Ant setSprite(Ant) + void removeSprite(Ant) + void updateScent(int) + Ant getAnt() + String toString() + boolean hasPortal()

Colony	This class does all the magic for Ants. Every ant belongs to a colony, the colony overall has needs (food and space) a distinctive color (black or red) a queen and Grid it calls home. The colony takes care of choosing the color of ants, checking what ants should do what next, etc.  Colony Object - Grid myMap - String myName - Tile startTile - Ant myQueen - Game myGame + void <init>(Grid,String,Game) + int getFood() + Grid getGrid() + String getMyName() + Tile getStartTile() + Ant getQueen() + Game getGame() + void newAnt(AntType) + boolean takeFood() + String getImageLocation(AntType) + boolean checkEndgame() + Ant[] getMembers() + Ant[] getMembers(AntType) + boolean enoughDug() + boolean attackReady() *Notable Methods:* The constructor creates a new Queen and two workers when it is made.



UML Diagrams For Directives