* bwapi
  + PositionOrUnit
    - Position
      * 1x1 position pixel
      * holds an x and a y value
      * getApproxDistance(): distance between two positions
      * point<int, POSITION\_SCALE>
      * point<T, scale>
    - WalkPosition
      * 8x8
    - TilePosition
      * 32x32
    - Unit
      * pointer to a UnitInterface
      * get information about an individual unit and issue it orders
      * getLastCommand(): the command that was successfully issued to this unit
  + Unitset
    - set of pointers to unit objects
    - allows you to control multiple units at once
  + UnitType
    - lets you get information about a type of unit
  + TechType
    - represents a passive upgrade that can be researched with unitinterface::upgrade
  + UpgradeType
    - represents units’ abilities that can be researched and/or used by a UnitInterface
  + Order
  + Player
    - pointer to a PlayerInterface
    - each player in a game has an instance
    - neutral player owns things like mineral patches
    - allows you to get information and statistics about a player
  + UnitCommand
    - has a variety of static functions that issue commands to units
  + UnitCommandType
  + Broodwar
    - a Game
    - contains a wide variety of useful functions
    - mapWidth/Height: width and height of map in Tiles
    - canBuildHere(): pass a TilePosition and UnitType to determine if the unit can be built there
    - isBuildable(): determines if you can build at the given x and y coordinates of a tile
    - getStaticGeysers(): returns all vespene geysers on map
    - isExplored()
    - canMake(), canResearch(), canUpgrade()
    - isWalkable()
    - getUnitsOnTile()
  + Race
    - allows you to get information about a race
* bwta
  + global functions:
    - functions to get all regions, all chokepoints, all base locations, all base locations that are start positions, and all unwalkable polygons
    - function to get region a bwapi::tileposition is in
    - functions to get nearest chokepoint, base location, tile position, unwalkable polygon, and unwalkable position
    - function to get distances to a set of tile locations
    - function to determine if two locations are connected, distance between locations
    - function to get shortest path
  + BaseLocation
    - position on the base where it makes sense to put a base (has minerals)
    - getPosition(), getTilePosition(), getRegion(),
    - functions that get total mineral and gas count
    - functions that return a bwapi::unit\* set referencing mineral and gas deposits
    - functions for ground and air distance to other bases
    - functions that determine if a base is a starting location, has minerals only, and if it is an island
  + Region
    - partition of the map defined by a polygon boundary. connected to other regions by chokepoints
    - getPolygon(), getCenter(), getChokepoints(), getBaseLocations(), isReachable() from another region, getReachableRegions()
  + Polygon
    - vector of bwapi::positions
    - functions to get area, perimeter, centre
    - getNearestPoint(): returns the point on the edge of the polygon closest to a given position
    - isInside(): returns true if given position is inside the polygon
  + Chokepoint
    - connects exactly two regions
    - functions to get the two regions connected by the chokepoint, the sides of the chokepoint, the center, and the width
* GameCommander
  + in general, manages the use of the other managers
  + class UnitToAssign: holds a unit and bool isAssigned
  + private members
    - combatcommander, timermanager
    - unitsets validunits, combatunits, and scoutunits
    - bool initialscoutset
      * starts at false
  + private functions
    - assignunit: assign given unit to given unitset
    - isassigned: return true if given unit is in combatunits or scoutunits
  + public functions
    - update(): calls update() for InformationManager, MapGrid, BOSSManager, WorkerManager, ProductionManager, BuildingManager, \_combatCommander, and ScoutManager. MapTools is currently disabled
    - handleunitassignments: clear valid and combat units, then set valid, scout, and combat units
    - setvalidunits: adds all usable units to validunits
    - setscoutunits: if there is no scout unit, get a worker to scout by adding it to scoutunits (initialscoutset = false and scoutunits is empty)
    - setcombatunits: for all units in valid units, if it’s not assigned and it’s a combat unit, or it’s a worker, then add it to combatunits
    - getclosestunittotarget: given a unittype and a position, search every unit in validunits. returns a pointer to the unit that is the same type as the given type and is closest to the given position
    - getclosestworker to target: search all workers in validunits for the worker closest to the given position who is free. returns a pointer to the worker unit
    - assignunit: given a unit and a unitset, removes the unit from scoutunits and combatunits and assigns it to the given unitset
* combat
  + Micro
    - not a class; rather a collection of global functions accessed through Micro::
    - contains the essential unit movements that can be used by other classes
    - SmartAttackUnit(): given unit attacks another unit
    - SmartAttackMove(): given unit attack moves to the given position
    - SmartMove(): move given unit to given position
    - SmartRightClick(): unit right clicks target unit (ex. to gather)
    - SmartLaySpiderMine(): unit lays spider mine at target position
    - SmartRepair(): unit repairs target
    - SmartKiteTarget(): kites a target, after considering various situations where kiting won’t be useful. note: seems to have to manually consider range upgrades
    - MutaDanceTarget(): attacks a target, flees, and comes back when attack cooldown is done. flee direction is determined by GetKiteVector()
    - GetKiteVector(): used to help mutas kite. note: consider using this for normal kiting? or zergling kiting?
    - Rotate(): takes an x y vector and an angle to rotate the vector by
    - Normalize(): normalizes an x y vector to distance 1
  + MicroManager
    - most of the other combat managers inherit from this
    - structs AirThreat and GroundThreat, which hold a unit and a weight
    - private members:
      * bwapi::unitset \_units
    - protected members:
      * SquadOrder order
    - protected methods:
      * unitNearEnemy(): return true if given unit is within 800 units of an enemy
      * checkPositionWalkable(): returns true if a given position is walkable according to bwapi::broodwar->isWalkable(), and the tile does not have a building or ground-attacking unit on it
      * trainSubunits(): for a protoss carrier or reaver, have it spawn subunits
      * unitNearChokepoint(): returns true if given unit is 80 units away from a bwta::chokepoint
      * executeMicro(): virtual function that must be overriden
    - public methods:
      * setUnits(): sets \_units to given unitset
      * calcCenter(): returns bwapi::position(0,0) if \_units is empty. otherwise returns a bwapi::position corresponding to the average center of the group of units in \_units
      * execute(): does nothing if \_units is empty. otherwise, create unitset of nearby enemies and populate it with enemies in the target area if Defend and enemies in the target area as well as on the way there if Attack. this is done with the help of MapGrid.getUnits(). then attack all units on the way to the order position???
      * getUnits(): return \_units
      * regroup(): moves all \_units towards given regroup bwapi::position
  + MeleeManager
    - inherits from MicroManager
    - \_units should be a unitset of only melee units
    - public methods:
      * executeMicro(): calls assignTargetsOld() on the given unitset of targets
      * assignTargetsOld(): given a set of targets, isolates the targets that can be attacked by melee units . then for every melee unit, 1. test if meleeUnitShouldRetreat(). if so, make it run towards main base. 2. use getTarget() to find a suitable target and have it attack it. 3. move to the order position
      * findClosestUnitPair(): given unitsets of attackers and targets, for every attacker, call getTarget(). then get the distance between the attacker and target (possible bug here??). keep doing this and return the attacker/target pair with the shortest distance
      * getTarget(): search given unitset of targets. for each target, call getAttackPriority() and getDistance() between the attacker and target. returns the target with firstly the highest priority then the shortest distance
      * getAttackPriority(): given an attacker and a target, return an int corresponding to attack priority. can have a variety of if statements for different attacker/target combinations
      * closestMeleeUnit(): given a set of melee units to assign and a target, returns the closest melee unit to the target
      * meleeUnitShouldRetreat(): 1. terran don’t retreat. 2. don’t retreat the unit unless its shields or hp are below a threshold defined in Config. 3. if there are ranged units in the given set of targets, and their range exceeds a certain threshold and the unit to retreat is within range, don’t bother retreating since it’s not worth taking the damage
      * assignTargetsnew(): buggy. tries not to assign more than 2 units to attack a zergling and not more than 10 units on a building
  + RangedManager
    - inherits from MicroManager
    - \_units should be a unitset of only ranged units
    - public methods:
      * executeMicro(): call assignTargetsOld on given set of targets
      * assignTargetsOld(): go through all ranged units. use getTarget() to find each unit a target. attack it by calling the appropriate method in MicroManager based on the situation. if there are no targets, move to the order position. currently option to trainSubUnits is off.
      * findClosestUnitPair(): identical to MeleeManager
      * getTarget(): finds a target from a set of targets for given ranged unit. uses priority and distance similar to MeleeManager. also has potential to employ LTD function in UnitUtil, which calculates DPS as a measure of a target’s threat
      * getAttackPriority(): similar to MeleeManager. useful to make certain units act as counters to others. also has some lines to guard against enemies building things 1200 units away from main base.
      * closestrangedUnit(): from a set of ranged units, returns the closest unit to a target
      * assignTargetsNew(): similar to MeleeManager, also buggy
  + DetectorManager
    - inherits from MicroManager
    - \_units should be a set of detector units
    - private members:
      * cloakedUnitMap: map of bools to bwapi::units. tracks enemy cloaked units and tries to assign a detector to each
      * bwapi::unit unitClosestToEnemy
    - private methods:
      * isAssigned()
    - public methods:
      * executeMicro(): look at all enemy units and make a unitset of enemy units that can cloak. for all detector units, 1. move to the position of the unitClosestToEnemy if there is a battle. 2. otherwise have it scout by moving it to MapGrid.getLeastExplored()
      * closestCloakedUnit(): returns the closest cloaked unit from a set to a given detector unit if that cloaked unit hasn’t been assigned a detector yet
      * setUnitClosestToEnemy(): sets unitClosestToEnemy to given unit
  + ScoutManager
    - only manages a worker scout and not a combat unit scout
    - private members:
      * bwapi::unit \_workerScout
      * some bools relating to gas steal. \_didGasSteal: prevents you from gas stealing more than once
    - private methods:
      * moveScouts(): 1. does nothing if you don’t have a \_workerScout. 2. peforms gasSteal() and checks to see if the gas steal is in progress or is completed. 3. if we know where the enemy base is, if the scout is not under attack by a non-worker, harass workers. 4. flee if getting attacked. 5. if we aren’t at the enemy base, go there. 6. if we don’t know where enemy base is, search for it. 7. update \_previousScoutHP
      * followPerimeter(): call getFleePosition() to find a position to move \_workerScout to
      * gasSteal(): can be turned off in Config. first checks preconditions for gas stealing (i.e. found enemy base and geyser). then calls ProductionManager.queueGasSteal(). sets \_didGasSteal flag to true
      * closestEnemyWorker(): returns the closest enemy worker. prioritizes a worker that is constructing. then finds the enemy worker closest to a geyser?
      * getEnemyGeyser(): searchers enemy units in their main base for a geyser and returns it
      * enemyWorkerInRadius(): searches enemy units for a worker that is within 300 units of \_workerScout
      * immediateThreat(): returns true if \_workerScout isUnderAttack(). also returns true if the \_workerScout is in attacking range of enemy combat units. also makes a unitset of enemy combat workers for no apparent reason
      * getClosestVertexIndex(): searches \_enemyRegionVertices for the closest vertex to the given unit
      * getFleePosition(): ??
      * calculateEnemyRegionVertices(): ??
    - public methods:
      * Instance()
      * update(): ScoutManager can be turned off in Config. 1. calls calculateEnemyRegionVertices() if it hasn’t been done yet. 2. moveScouts()
      * setWorkerScout(): releases the old \_workerScout and makes the given unit the new \_workerScout. calls WorkerManager.setScoutWorker()
  + MedicManager
    - inherits from MicroManager
    - executeMicro(): medics heal damaged units and move to squad order position
  + TankManager
    - inherits from MicroManager
    - enables use of siege mode
    - may have some play around chokepoints
  + Squad
    - private members:
      * various managers to control its units
      * SquadOrder \_order
    - private methods:
    - public methods:
  + SquadOrder
    - enum SquadOrderTypes: various squad orders such as Attack, Defend, etc
    - size\_t \_type, int \_radius, bwapi::position \_position, string \_status. radius and position correspond to a circle where the order is meant to be carried out
  + CombatSimulation
    - interacts with Sparcraft
    - private member: SparCraft::GameState state
    - public methods:
      * setCombatUnits(): set the starting states based on combat units inside a given radius at given position. the center is most likely the position of the forwardmost allied unit. adds ally and enemy units to state. has a few selection criteria: does not consider workers, seems to not consider flying units, and always checks that SparCraft supports this type of unit
      * getSparCraftUnit(): given a bwapi::unit or a UnitInfo, returns the corresponding sparcraft::unit. currently treats medics as marines
      * simulateCombat(): returns a SparCraft::ScoreType after running a combat simulation with state
      * getSparCraftState(): return state
      * getSparCraftPlayerID(): returns the sparcraft equivalent of given bwapi::player
* macro
  + BuildOrder
    - private members:
      * bwapi::race \_race
      * vector of MetaTypes \_buildOrder
    - public methods:
      * add(): add a MetaType to the \_buildOrder
      * getRace(), size()
  + BuildOrderQueue
    - struct BuildOrderItem
      * metaType (thing to build), priority (priority in queue), blocking (bool), isGasSteal (bool)
    - private members:
      * queue: deque of BuildOrderItems. highest priority is back of the queue
      * int lowestPriority, highestPriority (both initialized to 0)
      * int defaultPrioritySpacing (initialized to 10), numSkippedItems (initialized to 0)
    - public methods:
      * clearAll(): empties queue and resets priorities to 0
      * getHighestPriorityItem(): resets numSkippedItems to 0 and returns queue.back()
      * getNextHighestPriorityItem(): returns the Xth last item in queue, where X is numSkippedItems
      * skipItem(): after checking canSkipItem(), increment numSkippedItems
      * canSkipItem(): makes sure the queue is big enough to skip, and checks the next highest priority item’s blocking value: if blocking is turned on, you can’t skip
      * queueItem(): pushes a BuildOrderItem onto queue, then sorts it with the highest priority at the back, then updates lowestPriority and HighestPriority with the priority values of items in the front and back of queue
      * queueAsHighestPriority(): constructs a BuildOrderItem from given MetaType, blocking, and gasSteal and calls queueItem() with a priority of highestPriority + defaultPrioritySpacing
      * queueAsLowestPriority(): similar to above but gives the item lowest priority
      * removeHighestPriorityItem(): queue.pop\_back(). also readjusts highest and lowest priorities
      * removeCurrentHighestPriorityItem(): remove the back item adjusting for numSkippedItems
      * size(), isEmpty(): check these values for queue
  + BOSSManager
    - private members:
    - private methods:
    - public methods:
      * Instance()
  + BuildingManager
    - private members:
      * vector of Buildings \_buildings. represents buildings to be built
      * int \_reservedMinerals, \_reservedGas: reserved resources for planned buildings
    - private methods:
      * validateWorkersAndBuildings(): check if assigned workers have died . call removeBuildings() on all buildings that are obsolete
      * assignWorkersToUnassignedBuildings(): assign workers to unassigned buildings and label them planned using WorkerManager
      * constructAssignedBuildings(): check planned buildings and tell the worker to construct it. explore the build position if needed. if the worker has already been instructed to build several times but is failing, call finishedWithWorker(), freeTiles(), and set the building back to Unassigned
      * checkForStartedConstruction(): check if any buildings have started construction. for all assigned buildings currently under construction, unreserve their minerals and gas
      * checkForDeadTerranBuilders(): Terran only, but not implemented
      * checkForCompletedBuildings(): check building completion. for buildings that have completed, remove them from the under construction vector
      * isEvolvedBuilding(): given a UnitType, if it is a Zerg building that can evolve, return true
      * isBuildingPositionExplored(): for a given building to be built, returns false if any of the tiles where it will be built is unexplored (bwapi::broodwar->isExplored())
      * getBuildingWorkerCode():
    - public methods:
      * Instance()
      * update(): called in GameCommander. calls the first 6 private methods listed above
      * isBeingBuilt(): search \_buildings for the given UnitType and return true if it is in \_buildings
      * addBuildingTask(): given a building UnitType and target TilePosition, reserve resources for it, create a Building object, initialize its BuildingStatus to Unassigned, and add it to \_buildings. also an option for isGasSteal
      * getReservedMinerals/Gas()
      * buildingsQueued(): returns a vector of bwapi::UnitTypes corresponding to buildings in \_buildings that are either Assigned or Unassigned
      * getBuildingLocation(): returns a TilePosition for a building to be built. 1. for a gas steal, you must have first searched the enemy base and found a geyser. then it returns the tile position of a geyser. 2. if it’s a refinery, use BuildingPlacer to get a refinery position. 3. if it’s a depot, use MapTools to getNextExpansion(). 3. otherwise, use BuildingPlacer to getBuildLocationNear. building spacing can be manipulated in Config
      * removeBuildings(): given a vector of Buildings, remove them from \_buildings
  + BuildingPlacer
    - functions to find suitable building locations
    - private members:
      * \_reserveMap: vector of vector of bools. size = bwapi::broodwar->mapWidth() and height. initialized to false
      * int \_boxTop, \_boxBottom, \_boxRight, \_boxLeft. top/left initialized to max. bottom/right initialized to min. represents the resource box around a base that you can’t build in
    - private methods:
    - public methods:
      * Instance(): lets you access BuildingPlacer anywhere
      * computeResourceBox(): called during constructor. computes the resource box by getting mineral deposits near the start base and putting the results in \_box variables
      * isInResourceBox(): returns true if given x and y coordinates representing a Tile position are inside the resource box
      * canBuildHere(): uses bwapi::broodwar->canBuildHere(). also checks tiles on reserve map: if one returns true, the map is being reserved for something else. also checks if tileOverlapsBaseLocation()
      * tileBlocksAddon(): only applies to Terran
      * canBuildHereWithSpace(): 1. call canBuildHere() to make sure you can build here. 2. define a buffer of space around the building based on a given amount. this extra space can be made horizontal only. 3. see if this space stays within the map, does not overlap resource box, is not overlapping reserved space, and is buildable(). return true if conditions are satisfied
      * GetBuildLocation(): returns TilePosition(0,0)
      * getBuildLocationNear(): 1. use MapTools to get a sorted vector of tiles that are close to the desired location for the building. 2. returns None if not enough pylons for Protoss. 3. search the sorted vector of tiles for a tile that satisfies canBuildHereWithSpace(). this function can take a buffer space similar to canBuildHereWithSpace(). 4. return the tile, or none if none found
      * tileOverlapsBaseLocation(): checks to see if a given building to be built at given position will overlap a base location. checks all base locations. does not check for a depot
      * buildable(): checks if a building is buildable at given tile coordinates, taking into account units on the tile. uses bwapi::broodwar->isBuildable().
      * reserveTiles(): reserves a rectangular space of tiles in \_reserveMap corresponding to the given TilePosition and width and height
      * freeTiles(): similar to above, but frees tiles instead
      * getRefineryPosition(): returns the TilePosition of the closest geyser to the main base
      * isReserved(): checks if given x y coordinates are reserved
  + ProductionManager
    - in general, works extensively with BuildOrderQueue and BOSS. Also has connections to StrategyManager
    - Action: unsigned char
    - private members:
      * BuildOrderQueue \_queue
      * bwapi::tileposition \_predictedTilePosition
      * bools \_enemyCloakedDetected, \_assignedWorkerForThisBuilding, \_haveLocationForThisBuilding (all initialized with false)
    - private methods:
      * setBuildOrder(): 1. clears \_queue. 2. for all items in given BuildOrder, queueAsLowestPriority() with blocking = true. called in constructor, and uses BuildOrder obtained from StrategyManager.getOpeningBookBuildOrder()
      * manageBuildOrderQueue(): does nothing if \_queue is empty. otherwise, getHighestPriorityItem(). for this item, 1. use getProducer() to find the unit that can produce the item. 2. use canMakeNow() to see if it can be made right now. 3. also seems to prevent the production of more than 3 refineries. 4. if the next item is a building and we can’t make it yet, construct a Building object and prepare a worker to construct it by using WorkerManager.getBuilder() and predictWorkerMovement() 5. if you can make the current item, call create() and removeCurrentHighestPriorityItem(). 6. if the above all fail, see if you can skip the current item and call getNextHighestPriorityItem(). if so, keep going through \_queue until it’s empty. if no, exit
      * getClosestUnitToPosition(): given a bwapi::unitset, returns the unit that is closest to the given bwapi::position. if position = none, i.e. you don’t care where, then returns the first unit.
      * create(): given a unit and a BuildOrderItem, has the unit build the BuildOrderItem. 1. if the item to create is a building, call BuildingManager.addBuildingTask(). 2. if the item to create is a unit, tell the producer unit to either train() or morph() the unit if it is Zerg. 3. if the item to create is a tech or upgrade, tell the producer to research() or upgrade()
      * canMakeNow(): checks to see if a given producer unit can make a given MetaType right now. 1. call meetsReservedResources(). 2. call bwapi::broodwar->canMake/Research/Upgrade() depending on the item to be made
      * detectBuildOrderDeadlock(): 1. see if supply providers are currently being built. 2. see if the current item being built (getHighestPriorityItem()) needs more supply. 3. if there isn’t enough supply and none is planned, there is supply block. 4. if zerg, and if a building is planned to be built, there is not supply block (???)
      * predictWorkerMovement(): called when the next item in \_queue is a building. 1. get a building location for the building with BuildingManager.getBuildingLocation(). 2. if certain conditions are met, WorkerManager.willHaveResources() and WorkerManager.setMoveWorker() are called
      * performCommand(): if the given bwapi::UnitCommandType is Cancel\_Construction, perform the extractor trick (cancel the construction of an extractor)
      * getFreeMinerals/Gas(): total resources – reserved resources from BuildingManager
      * meetsReservedResouces(): given a MetaType, see if you have the resources to produce it taking into account reserved resources for buildings
      * selectUnitOfType(): returns the unit of given UnitType closest to given bwapi::position, or nullptr if there are no units of that type. if the UnitType is a building, it will also consider how much training time the building has left
      * canPlanBuildOrderNow():
    - public methods:
      * Instance()
      * performBuildOrderSearch(): 1. does not do anything if build order search is turned off in Config, or if canPlanBuildOrderNow() returns false. 2. getBuildOrder() from BOSSManager and setBuildOrder() with this build order. 3. reset() BOSSManager. 4. if BOSSManager did not produce a build order, and it is not currently searching for one, start a new search based on StrategyManager.getBuildOrderGoal()
      * update(): called by GameCommander. 1. manageBuildOrderQueue(). 2. if nothing is currently building (i.e. \_queue.size() = 0), performBuildOrderSearch() to try and get something. 3. if supply blocked (detectBuildOrderDeadlock()), queueAsHighestPriority() a bwapi::...getSupplyProvider(). 4. if enemy has undetected cloaked units (\_enemyCloakedDetected = false and InformationManager.enemyHasCloakedUnits()), Zerg currently doesn’t do anything. set \_enemyCloakedDetected to true
      * onUnitDestroy(): if a given ally unit is destroyed, re-performBuildOrderSearch(), unless it is a scout or Zerg drone (is this problematic??)
      * getProducer(): returns a unit that can produce the given MetaType that is closest to the given bwapi::position. uses getClosestUnitToPosition() as a helper
      * queueGasSteal(): queues a gas steal at highest priority
  + MetaType
    - holds information about units, techs, upgrades, commands, etc.
    - \_type: a size\_t that refers to the enum
    - various Boolean functions about whether it is a unit, a tech, a building, a command, etc
    - functions to get cost
  + WorkerData
    - carries data about workers and their jobs
    - note that bwapi::unit parameters can get confusing. they can be mineral patches, bases, workers, etc.
    - class WorkerMoveData
    - enum WorkerJob: various enums for worker tasks
    - private members:
      * bwapi::unitset workers
      * bwapi::unitset depots. i.e. bases. workers get assigned to bases
      * various maps to map activities to a unit: workerJobMap, workermineralmap, workerdepotmap, workerrefinerymap, workerrepairmap, workermovemap, workerbuildingtypemap
        + workerbuildingtypemap: maps a building to a worker building it
        + workerMoveMap: maps a WorkerMoveData to a worker
      * maps for depotworkercount, workersonmineralpatch, and refineryworkercount
      * map for workermineralassignment: maps mineral patches to workers
    - private functions:
      * clearPreviousJob()
    - public functions:
      * getWorkers(): returns workers
      * workerDestroyed(): for a destroyed worker, clearPreviousJob() and remove from workers
      * addWorker(): add given worker to workers and set workerJobMap[unit] to Default. can pass additional parameters and call setWorkerJob() to initialize it with a non-default job
      * addDepot(): add given unit to depots. initialize depotWorkerCount[unit] to 0
      * removeDepot(): remove given unit from depots and depotWorkerCount. then rebalance workers: any workers that were working at this depot are now Idle
      * addToMineralPatch(): adjusts the number of workersOnMineralPatch for given mineral patch
      * setWorkerJob(): gives a worker a new job. depending on the job, data members are adjusted
      * clearPreviousJob(): erases the unit from workerJobMap. adjusts data members depending on the previous job. this method needs to be called before giving a worker a new job
      * getNumWorkers(): return workers.size()
      * getNumMineralWorkers(): returns the total number of workers gathering minerals
      * getNumGasWorkers(): returns the total number of workers gathering gas
      * getNumIdleWorkers(): returns the number of workers whose job is Idle
      * getWorkerJob(): returns the WorkerJob enum for a unit. returns Default as a default
      * depotIsFull(): returns true if there are more than 3 workers per mineral patch at a given base
      * getMineralPatchesNearDepot(): returns a unitset of minerals 300 units away from given base. if there are no minerals this close, just use the whole map
      * getMineralsNearDepot(): returns the number of mineral patches 200 units away from given base. if none, return 0
      * getWorkerResource(): given a unit, returns a bwapi::unit corresponding to the mineral or gas patch it is mining. does this by searching either workerMineralMap or workerRefineryMap. returns nullptr as default
      * getMineralToMine(): returns a bwapi::unit associated with a nearby mineral patch for a given worker to mine
      * getWorkerRepairUnit(): searches workerRepairMap to return a bwapi::unit corresponding to the building under repair. returns nullptr as a default
      * getWorkerDepot(): search workerDepotMap to return the base a worker is assigned to
      * getWorkerBuildingType(): search workerBuildingTypeMap for a given worker and return a bwapi::UnitType
      * getWorkerMoveData(): search workerMoveMap for a unit and return a WorkerMoveData
      * getNumAssignedWorkers(): search depotWorkerCount or refineryWorkerCount for a given base or refinery to return the number of assigned workers
      * getJobCode(): for a unit, returns its job as a single char
  + WorkerManager
    - manipulates and interacts with WorkerData to control workers
    - private members:
      * WorkerData workerData
      * bwapi::unit previousClosestWorker. initially is nullptr
    - private functions:
      * isGasStealRefinery(): returns true if given worker is performing a gas steal in the enemy base
      * handleGasWorkers(): search all units for all refineries. if a refinery has fewer workers than desired (according to Config) use getGasWorker() to add more workers
      * handleIdleWorkers(): for all Idle workers, make them mine minerals with setMineralWorker(). i.e. minerals is the default worker job
      * handleRepairWorkers(): only called for Terran. search all units for damaged buildings. for every damaged building, getClosestMineralWorker() to the building and setRepairWorker()
      * handleCombatWorkers(): search all workers for Combat workers. have them each attack the closest enemy unit. note: the micro here is quite poor
      * handleMoveWorkers(): search all workers for Move workers. moves each worker to a position defined in each worker’s WorkerMoveData
    - public functions:
      * Instance(): WorkerManager::Instance() lets you access WorkerManager anywhere in UAlbertaBot
      * update(): calls various functions: updateWorkerStatus(), handle gas/idle/move/repair/combat workers
      * updateWorkerStatus(): update worker status for every worker in WorkerData. if a worker is working on gas but the refinery is destroyed, move the worker to minerals
      * setRepairWorker(): uses workerData.setWorkerJob() to give a worker a unit to repair
      * stopRepairing(): make a unit stop repairing
      * getClosestEnemyUnit(): search all enemy units for the unit closest to given worker
      * finishedWithCombatWorkers(): calls setMineralWorker on all Combat workers
      * getClosestMineralWorkerTo(): returns closest mineral worker to an enemy unit, and also sets previousClosestWorker to this unit
      * getWorkerScout(): searches all workers for one whose job is Scout and returns it. otherwise return nullptr
      * setMineralWorker(): calls getClosestDepot to see if there are minerals near the given worker, and sets the worker to mine minerals there
      * getClosestDepot(): returns the closest depot to the given worker
      * finishedWithWorker(): sets the given worker to Idle unless it is a Scout. other managers that need workers call this when they are done with the worker
      * getGasWorker(): search all workers for Mineral workers. returns the closest Mineral worker to the given refinery
      * setBuildingWorker(): sets a worker’s job to Build the given Building
      * getBuilder(): gets a worker for BuildingManager to use. setJobAsBuilder is a Boolean parameter. if true, it will set the selected worker to Build. if false, the function will only return the selected worker. 1. if the building isGasSteal, choose a scout worker to build it. 2. otherwise, look for the closest Mineral worker and the closest Move worker. 3. return the closest worker to build the building. prioritize Move workers
      * setScoutWorker(): make given worker a Scout
      * getMoveWorker(): returns the closest Mineral worker to target position
      * setMoveWorker(): similar to getMoveWorker, but actually sets the worker to move to the position rather than returning it
      * willHaveResources(): determines if we will have the target amount of minerals and gas in the time it takes for a worker to travel the given distance
      * setCombatWorker(): makes a worker a Combat worker
      * onUnitMorph(): adds given worker to workerData, or removes it from workerData if it morphed into a building
      * onUnitShow(): add given unit to workerData if it is a depot or worker
      * rebalanceWorkers(): for every Mineral worker, check the depot it’s working at. if depotIsFull() or the depot doesn’t exist anymore, make the worker Idle
      * onUnitDestroy(): removes given unit from workerData if it’s a depot or worker. if it’s a mineral patch, rebalanceWorkers()
      * isFree(): returns true if given worker’s job is Idle or Minerals
      * isWorkerScout(): return true if given worker is Scout
      * isBuilder(): return true if given worker is Build
      * getNumMineralWorkers(), getNumIdleWorkers(), getNumGasWorkers(): call associated function in workerData
* util
  + DistanceMap
    - private members:
      * int rows, cols. initialized to bwapi::broodwar->mapheight() and width.
      * int startRow, startCol. initialized to -1.
      * dist: vector of ints. size = width x height. values initialized to -1.
      * moveTo: vector of chars. size = width x height. values initialized to ‘X’
      * sorted: vector of bwapi::TilePositions
    - private methods:
      * getIndex(): returns the index for either given row and col, or bwapi::position
    - public methods:
      * [] operator: given index or bwapi::position, returns the associated value in dist
      * setMoveTo(): give an index and value to set moveTo
      * setDistance: give an index and value to set dist
      * setStartPosition(): give a start row and start column to set startRow and startCol
      * reset(): resets the distance map. can pass it new cols and rows if you like
      * getSortedTiles(): return sorted
      * isConnected: given a bwapi::position, determines if it is connected. does this by getting the index of the position and searching dist to see if it is -1 or not
      * addSorted(): adds a bwapi::TilePosition to sorted
      * getMoveTo(): given a position, gives the bwapi::position we should move to to minimize distance. takes an int lookAhead which can be increased to make several steps forward
  + MapGrid
    - in general, holds information about the map, in particular location of natural expansion and unit locations
    - class GridCell
      * int timeLastVisited and timeLastOpponentSeen
      * bwapi::unitset ourUnits and oppUnits
      * bwapi::position center
    - private members:
      * int cellsize, mapwidth, mapheight, rows, cols
      * cells: vector of GridCells
      * bwapi::position naturalExpansion
    - private methods:
      * getCellCenter(): given int row and col, get bwapi::position of the center of that cell
      * clearGrid(): clears cells
    - public methods:
      * Instance(): use MapGrid::Instance() to access from anywhere in ualbertabot
      * getNaturalExpansion()
      * getLeastExplored(): returns the bwapi::position that has been least explored
      * update(): clear cells. then for all ally and enemy units, find out what cell they are standing on and add them to it
      * getCell(): returns the GridCell associated with the cell a unit is standing on or the cell at a position
      * getCellByIndex(): given int row and col, returns the GridCell in cells associated with row and col
      * getUnits(): given a position and a radius, adds units in the radius to given bwapi::unitset. can consider only ally units, only enemy units, or both, depending on given parameters.
  + MapTools
    - in general, holds information about the map, in particular distances and connectivity between a variety of locations
  + private members:
    - \_allMaps: maps DistanceMaps to bwapi:Positions. cache of computed distance maps
    - \_map: vector of bools. stores map at TilePosition resolution. 0/1 values for walkable or not walkable. size = width\*height. initialized to false.
    - \_units: vector of bools. stores whether a unit is on the position or not. size = width\*height. initialized to false.
    - \_fringe: vector of ints. size = width\*height. initialized to 0.
    - int \_rows, \_cols. initialized to bwapi::broodwar()->mapHeight() and width
  + private methods:
    - getIndex(): given row and col, returns index for that position
    - setBWAPIMapData(): reads in map data from BWAPI for use in this format. called during constructor
    - unexplored(): given an index and a distancemap, returns true if the index is not -1, it is walkable according to \_map, and the distancemap is -1 at that index
    - reset(): resets fringe. call before searching
  + public methods:
    - getGroundDistance(): 1. if there are too many maps (\_allMaps.size()>20), clear them to free memory. 2. if the distance hasn’t been computed, use computeDistance() and add it to \_allMaps. 3. return the distance in \_allMaps.
    - computeDistance(): computes walk distance from given bwapi::position to all other points on the map by calling search() and storing it in a distancemap
    - getClosestTilesTo(): returns vector of bwapi::TilePositions that are closest to given position
    - getTilePosition(): returns bwapi::TilePosition for an index
    - getNextExpansion(): for a player, returns the closest base to the main base that has gas. uses self as default player.
  + InformationManager
    - in general, contains all data on the current visible state of the game
    - struct BaseInfo
    - BaseInfoVector: vector of baseinfo structs
    - private members:
      * self, enemy: bwapi::players
      * unitdata, mainbaselocations, occupiedregions: sets that map players to unitdata objects, bwta::baselocation pointers, and a set of bwta::region pointers
    - public functions:
      * Instance(): use InformationManager::Instance() to access the information manager from anywhere in ualbertabot
      * update(): calls updateunitinfo() and updatebaselocationinfo(). called repeatedly in update() loop in gamecommander
      * updateUnitInfo(): updates the state of ally and enemy units
      * updateBaseLocationInfo(): 1. clear occupiedregions for self and enemy. 2. try to identify the location of the main base. 3. call updateOccupiedRegions() on all regions with an enemy building. 4. call updateOccupiedRegions() on all regions with allied buildings
      * updateOccupiedRegions(): given a region and player, adds the region to \_occupiedRegions[player]
      * isEnemyBuildingInRegion(): return true if given region contains enemy building
      * getUnitInfo(): gets unit info for a player
      * getOccupiedRegions(): return regions occupied by a player
      * getMainBaseLocation(): get main base location of a player
      * updateUnit(): updates a unit for unitdata
      * isValidUnit(): determine if a unit is valid
      * onUnitDestroy(): updates a destroyed unit for unitdata
      * isCombatUnit(): determines if a bwapi::unittype is a combat unit
      * getNearbyForce(): examines all known units of a given player. if the unit can attack into a given radius at a given position, add it to the given vector<UnitInfo>
      * getNumUnits(): gets the number of units of a certain type for given player
      * getUnitData(): get unit data for a player
      * enemyHasCloakedUnits(): returns true if the enemy has cloaked units or a building to produce cloaked units
  + StrategyManager
    - in general, determines what you should build. unclear whether it should have a hand in managing combat strategy
    - MetaPair: maps size\_ts to MetaTypes
    - MetaPairVector: vector of meta pairs
    - struct Strategy: associated with a name, race, and BuildOrder
    - private members:
      * bwapi::race \_selfRace and \_enemyRace
      * \_strategies: maps Strategies to strings
      * BuildOrder \_emptyBuildOrder
    - private methods:
      * getProtossBuildOrderGoal(): 1. counts numbers of various owned units. 2. search a bunch of if statements for the strategy name given in Config. 3. add to a MetaPairVector various unit types and how many you want. 4. add various other things to build depending on conditions that aren’t related to build order strategy. 5. return the MetaPairVector
      * getTerranBuildOrderGoal(): similar to above
      * getZergBuildOrderGoal(): similar to above
      * shouldExpandNow(): 1. return false if there are no expansion locations. 2. expand if: more than 10 idle workers, more than 3000 minerals, or a certain amount of time (hardcoded) has passed
    - public methods:
      * Instance():use StrategyManager::Instance() to access the strategy manager from anywhere in ualbertabot
      * getOpeningBookBuildOrder(): finds the strategy given in Config and returns the BuildOrder associated with it
      * addStrategy(): given a Strategy and its name, add it to \_strategies
      * getBuildOrderGoal(): returns the build order goal, a MetaPairVector, according to race
      * also contains a bunch of methods for implementing UCB. their use is toggled with UsingStrategyIO in Config
  + TimerManager
    - private members:
      * \_timers: vector of timers
      * \_timerNames: vector of strings
  + UnitData
    - struct UnitInfo
      * stores data about a unit
      * main purpose is to keep track of units so they can still be accessed even if they leave vision range
    - UnitInfoVector: vector of infounits
    - UIMap: maps a unit to its UnitInfo
    - private members:
      * UIMap unitMap
    - public functions:
      * updateUnit(): updates a unit in unitMap
      * removeUnit(): removes a unit from unitMap
      * badUnitInfo(): determines if a unit is bad
      * removeBadUnits(): remove units from unitMap if badUnitInfo() returns true for them
      * getUnits(): return unitMap
* UnitUtil
* BuildingData
  + class Building
    - a building that will be built
* Config
  + variety of changeable parameters