**Data Clumps:**

The methods:

net.sf.freecol.common.model.Map.search((final Unit unit, Location start,

final GoalDecider goalDecider,

final CostDecider costDecider,

final int maxTurns, final Unit carrier,

LogBuilder lb))

and

net.sf.freecol.common.model.Map.searchMap(final Unit unit, final Tile start,

final GoalDecider goalDecider,

final CostDecider costDecider,

final int maxTurns, final Unit carrier,

final SearchHeuristic searchHeuristic,

final LogBuilder lb)

have the code smell DataClumps since both of them share the fields unit, start, goalDecider, costDecider, maxTurns, carrier, lb, this could be fixed by adding a new class containing all of these fields. In it we could create a

method to get the current unit

( **Unit currentUnit = (start.isLand())  
 ? ((unit != null && unit.getLocation() == carrier  
 && start.hasSettlement()  
 && start.getSettlement().isConnectedPort())  
 ? carrier : unit)  
 : offMapUnit;**

)

, this will also decrease the complexity of the method searchMap.

|  |
| --- |
| **public PathNode search(final Unit unit, Location start,  final GoalDecider goalDecider,  final CostDecider costDecider,  final int maxTurns, final Unit carrier,  LogBuilder lb) {  if (traceSearch) lb = new LogBuilder(1024);   final Unit offMapUnit = (carrier != null) ? carrier : unit;    PathNode p, path;  if (start instanceof Europe) {  // Fail fast if Europe is unattainable.  if (offMapUnit == null  || !offMapUnit.getType().canMoveToHighSeas()) {  path = null;   // This is suboptimal. We do not know where to enter from  // Europe, so start with the standard entry location...  } else if ((p = searchMap(unit,  offMapUnit.getFullEntryLocation(),  goalDecider, costDecider, maxTurns, carrier,  null, lb)) == null) {  path = null;   // ...then if we find a path, try to optimize it. This  // will lose if the initial search fails due to a turn limit.  // FIXME: do something better.  } else {  path = this.findPath(unit, start, p.getLastNode().getTile(),  carrier, costDecider, lb);  }   } else {  path = searchMap(unit, start.getTile(), goalDecider,  costDecider, maxTurns, carrier, null, lb);  }   finishPath(path, unit, lb);  return path; }** |

|  |
| --- |
| **private PathNode searchMap(final Unit unit, final Tile start,  final GoalDecider goalDecider,  final CostDecider costDecider,  final int maxTurns, final Unit carrier,  final SearchHeuristic searchHeuristic,  final LogBuilder lb) {  final HashMap<String, PathNode> openMap = new HashMap<>();  final HashMap<String, PathNode> closedMap = new HashMap<>();  final HashMap<String, Integer> f = new HashMap<>();  final PriorityQueue<PathNode> openMapQueue = new PriorityQueue<>(1024,  Comparator.comparingInt(p -> f.get(p.getLocation().getId())));  final SearchHeuristic sh = (searchHeuristic == null)  ? trivialSearchHeuristic : searchHeuristic;  final Unit offMapUnit = (carrier != null) ? carrier : unit;  Unit currentUnit = (start.isLand())  ? ((unit != null && unit.getLocation() == carrier  && start.hasSettlement()  && start.getSettlement().isConnectedPort())  ? carrier : unit)  : offMapUnit;  if (lb != null) lb.add("Search trace(unit=", unit,  ", from=", start,  ", max=", ((maxTurns == INFINITY)?"-":Integer.toString(maxTurns)),  ", carrier=", carrier, ")",  "\n", net.sf.freecol.common.debug.FreeColDebugger.stackTraceToString());   // Create the start node and put it on the open list.  final PathNode firstNode = new PathNode(start,  ((currentUnit != null) ? currentUnit.getMovesLeft() : -1),  0, carrier != null && currentUnit == carrier, null, null);  f.put(start.getId(), sh.getValue(start));  openMap.put(start.getId(), firstNode);  openMapQueue.offer(firstNode);   PathNode best = null;  int bestScore = INFINITY; ok: while (!openMap.isEmpty()) {  // Choose the node with the lowest f.  final PathNode currentNode = openMapQueue.poll();  final Location currentLocation = currentNode.getLocation();  openMap.remove(currentLocation.getId());  if (lb != null) lb.add("\n ", currentNode);   // Reset current unit to that of this node.  currentUnit = (currentNode.isOnCarrier()) ? carrier : unit;   // Check for success.  if (goalDecider.check(currentUnit, currentNode)) {  if (lb != null) lb.add(" \*\*\*goal(",  currentNode.getCost(), ")\*\*\*");  best = goalDecider.getGoal();  if (best == null || !goalDecider.hasSubGoals()) break ok;  bestScore = best.getCost();  continue;  }   // Valid candidate for the closed list.  closedMap.put(currentLocation.getId(), currentNode);  if (lb != null) lb.add(" closing");   // Skip nodes that can not beat the current best path.  if (bestScore < currentNode.getCost()) {  if (lb != null) lb.add("...goal cost wins(",  bestScore, " < ", currentNode.getCost(), ")...");  continue;  }   // Ignore nodes over the turn limit.  if (currentNode.getTurns() > maxTurns) {  if (lb != null) lb.add("...out-of-range");  continue;  }   // Collect the parameters for the current node.  final int currentMovesLeft = currentNode.getMovesLeft();  final int currentTurns = currentNode.getTurns();  final boolean currentOnCarrier = currentNode.isOnCarrier();   final Tile currentTile = currentNode.getTile();  if (currentTile == null) { // Must be in Europe.  // FIXME: Do not consider tiles "adjacent" to Europe, yet.  // There may indeed be cases where going to Europe and  // coming back on the other side of the map is faster.  if (lb != null) lb.add("...skip Europe");  continue;  }   // Try the tiles in each direction  PathNode closed;  for (Tile moveTile : currentTile.getSurroundingTiles(1)) {  // If the new tile is the tile we just visited, skip it.  if (lb != null) lb.add("\n ", moveTile);  if (currentNode.previous != null  && currentNode.previous.getTile() == moveTile) {  if (lb != null) lb.add(" !prev");  continue;  }   // Skip neighbouring tiles already too expensive.  closed = closedMap.get(moveTile.getId());  if (closed != null) {  int cc = closed.getCost();  if (cc <= currentNode.getCost()) {  if (lb != null) lb.add(" !worse ", cc);  continue;  }  }   // Prepare to consider move validity  Unit.MoveType umt = unit.getSimpleMoveType(currentTile,  moveTile);  boolean unitMove = umt.isProgress();  boolean carrierMove = carrier != null  && carrier.getSimpleMoveType(carrier.getTile(), moveTile).isProgress();  if (lb != null) lb.add(" ", ((unitMove) ? "U"  : ((carrierMove) ? "C" : "")));  MoveCandidate move;  String stepLog;    // Is this move to the goal? Use fake high cost so  // this does not become cached inside the goal decider  // as the preferred path.  boolean isGoal = goalDecider.check(unit,  new PathNode(moveTile, 0, INFINITY/2, false,  currentNode, null));  if (isGoal) {  if (lb != null) lb.add(" \*goal\*", umt);  if (unitMove) {  if (moveTile.hasSettlement() && currentOnCarrier && carrierMove) {  // If the goal has a settlement and the  // unit is travelling by carrier, dock the  // carrier.  move = new MoveCandidate(carrier, currentNode,  moveTile, currentMovesLeft, currentTurns,  true, CostDeciders.tileCost());  } else {  // Otherwise let the unit complete the path.  int left = (currentOnCarrier)  ? ((currentNode.embarkedThisTurn(currentTurns))  ? 0  : unit.getInitialMovesLeft())  : currentMovesLeft;  move = new MoveCandidate(unit, currentNode,  moveTile, left, currentTurns, false,  CostDeciders.tileCost());  }  } else {  // Handle some special cases where the move may not  // necessarily progress, but still can do useful work.  switch (umt) {  case ATTACK\_UNIT:  case ATTACK\_SETTLEMENT:  case ENTER\_FOREIGN\_COLONY\_WITH\_SCOUT:  case ENTER\_INDIAN\_SETTLEMENT\_WITH\_SCOUT:  case ENTER\_INDIAN\_SETTLEMENT\_WITH\_FREE\_COLONIST:  case ENTER\_INDIAN\_SETTLEMENT\_WITH\_MISSIONARY:  case ENTER\_SETTLEMENT\_WITH\_CARRIER\_AND\_GOODS:  // Can not move to the tile, but there is  // a valid interaction with the unit or  // settlement that is there.  move = new MoveCandidate(unit, currentNode,  moveTile, currentMovesLeft, currentTurns,  false, CostDeciders.tileCost());  unitMove = true;  break;  case EMBARK:  move = new MoveCandidate(unit, currentNode,  moveTile, currentMovesLeft, currentTurns,  true, CostDeciders.tileCost());  unitMove = true;  break;  case MOVE\_NO\_ATTACK\_CIVILIAN:  // There is a settlement in the way, this  // path can never succeed.  if (moveTile.hasSettlement()) {  if (lb != null) lb.add(" !FAIL-SETTLEMENT");  if (!goalDecider.hasSubGoals()) break ok;  continue;  }  // There is a unit in the way. Unless this  // unit can arrive there this turn, assume the  // condition is transient as long as the tile  // is not in a constrained position such as a  // small island or river.  if (currentNode.getTurns() <= 0  || moveTile.getAvailableAdjacentCount() < 3) {  if (lb != null) lb.add(" !FAIL-ATTACK");  if (!goalDecider.hasSubGoals()) break ok;  continue;  }  if (lb != null) lb.add(" blocked");  unitMove = true;  move = new MoveCandidate(unit, currentNode,  moveTile, currentMovesLeft, currentTurns,  false, CostDeciders.tileCost());  break;  default:  // Several cases here, these are understood:  // MOVE\_NO\_ATTACK\_EMBARK:  // Land unit trying to use water, which  // can not work without a ship there  // MOVE\_NO\_ACCESS\_WATER:  // The unit can not disembark directly to  // the goal along this path  // MOVE\_NO\_ATTACK\_MARINE:  // Ampibious attack disallowed, disembark to  // reach the goal  // There will be more.  // We used to do---  // if (!goalDecider.hasSubGoals()) break ok;  // --- here, like in the transient failure case  // above but we should not truncate other  // surrounding tiles.  if (lb != null) lb.add(" !FAIL-", umt);  continue;  }  }  assert move != null;  stepLog = "@";  } else {  // Ordinary non-goal moves.  //  // Check for a carrier change at the new tile,  // creating a MoveCandidate for each case.  //  // Do \*not\* allow units to re-embark on the carrier.  // Note that embarking can actually increase the moves  // left because the carrier might be not have spent  // any moves yet that turn.  //  // Note that we always favour using the carrier if  // both carrier and non-carrier moves are possible,  // which can only be true moving into a settlement.  // Usually when moving into a settlement it will be  // useful to dock the carrier so it can collect new  // cargo. OTOH if the carrier is just passing through  // the right thing is to keep the passenger on board.  // However, see the goal settlement exception above.  MoveStep step = (currentOnCarrier)  ? ((carrierMove) ? MoveStep.BYWATER  : (unitMove) ? MoveStep.DISEMBARK  : MoveStep.FAIL)  : (carrierMove && !usedCarrier(currentNode))  ? MoveStep.EMBARK  : (unitMove) ? ((unit.isNaval())  ? MoveStep.BYWATER  : MoveStep.BYLAND)  : MoveStep.FAIL;  switch (step) {  case BYLAND:  move = new MoveCandidate(unit, currentNode, moveTile,  currentMovesLeft, currentTurns, false, costDecider);  break;  case BYWATER:  move = new MoveCandidate(offMapUnit, currentNode, moveTile,  currentMovesLeft, currentTurns, currentOnCarrier,  costDecider);  break;  case EMBARK:  move = new MoveCandidate(offMapUnit, currentNode, moveTile,  currentMovesLeft, currentTurns, true,  costDecider);  move.embarkUnit(carrier);  break;  case DISEMBARK:  move = new MoveCandidate(unit, currentNode, moveTile,  0, currentTurns, false, costDecider);  break;  case FAIL: default: // Loop on failure.  if (lb != null) lb.add("!");  continue;  }  stepLog = " " + step + "\_";  }  if (move.cost >= INFINITY) {  continue;  }  assert move.getCost() >= 0;  // Tighten the bounds on a previously seen case if possible  if (closed != null) {  if (move.canImprove(closed)) {  closedMap.remove(moveTile.getId());  move.improve(openMap, openMapQueue, f, sh);  stepLog += "^" + Integer.toString(move.getCost());  } else {  stepLog += "v";  }  } else {  if (move.canImprove(openMap.get(moveTile.getId()))){  move.improve(openMap, openMapQueue, f, sh);  stepLog += "+" + Integer.toString(move.getCost());  } else {  stepLog += "-";  }  }  if (lb != null) lb.add(stepLog);  }  }** |

**Message chains:**

The method net.sf.freecol.server.ai.EuropeanAIPlayer.startWorking() has two very long and convoluted message chains:

final List<AIUnit> normalAiUnits = etAIUnits().stream().filter(MilitaryCoordinator.isUnitHandledByMilitaryCoordinator().negate()).collect(Collectors.toList());

final Set<AIUnit> militaryUnits = getAIUnits().stream().filter(MilitaryCoordinator.isUnitHandledByMilitaryCoordinator()).collect(Collectors.toSet());

which add to the complexity of the code making it hard to read and making the code rigid. One way of solving this smell would be to explain each variable by separating them or by creating methods that solve them separately.

|  |
| --- |
| public void startWorking() {  final Player player = getPlayer();  final Turn turn = getGame().getTurn();  final Specification spec = getSpecification();  initializeFromSpecification(spec);   // This is happening, very rarely. Hopefully now fixed by  // synchronizing access to AIMain.aiObjects.  if (getAIMain().getAIPlayer(player) != this) {  throw new RuntimeException("EuropeanAIPlayer integrity fail: " + player);  }  clearAIUnits();  player.clearNationCache();  badlyDefended.clear();   // Note call to getAIUnits(). This triggers  // AIPlayer.createAIUnits which we want to do early, certainly  // before cheat() or other operations that might make new units  // happen.  LogBuilder lb = new LogBuilder(1024);  int colonyCount = getAIColonies().size();  lb.add(player.getDebugName(),  " in ", turn, "/", turn.getNumber(),  " units=", getAIUnits().size(),  " colonies=", colonyCount,  " declare=", (player.checkDeclareIndependence() == null),  " v-land-REF=", player.getRebelStrengthRatio(false),  " v-naval-REF=", player.getRebelStrengthRatio(true));  if (turn.isFirstTurn()) initializeMissions(lb);    if (isLikesAttackingNatives() && getGame().getTurn().getNumber() > 100) {  for (Player p : getGame().getLivePlayerList(player)) {  if (!p.isIndian()) {  continue;  }  player.getTension(p).setValue(Tension.TENSION\_MAX);  }  }    determineStances(lb);   if (colonyCount > 0) {  lb.add("\n Badly defended:"); // **FIXME:** prioritize defence  for (AIColony aic : getAIColonies()) {  if (aic.isBadlyDefended()) {  badlyDefended.add(aic);  lb.add(" ", aic.getColony());  }  }   lb.add("\n Update colonies:");  for (AIColony aic : getAIColonies()) aic.update(lb);   buildTipMap(lb);  buildWishMaps(lb);  }  cheat(lb);  buyUnitsInEurope(lb);   // Note order of operations below. We allow rearrange et al to run  // even when there are no movable units left because this expedites  // mission assignment.  List<AIUnit> aiUnits = getAIUnits();  final Set<AIUnit> militaryUnits = getAIUnits().stream()  .filter(MilitaryCoordinator.isUnitHandledByMilitaryCoordinator())  .collect(Collectors.toSet());    final MilitaryCoordinator militaryCoordinator = new MilitaryCoordinator(this, militaryUnits);  militaryCoordinator.determineMissions();    buildTransportMaps(lb);    final List<AIUnit> normalAiUnits = getAIUnits().stream()  .filter(MilitaryCoordinator.isUnitHandledByMilitaryCoordinator().negate())  .collect(Collectors.toList());  for (int i = 0; i < 3; i++) {  rearrangeColonies(lb);  giveNormalMissions(lb, normalAiUnits);  bringGifts(lb);  demandTribute(lb);  if (aiUnits.isEmpty()) break;  aiUnits = doMissions(aiUnits, lb);  }      lb.log(logger, Level.FINE);   clearAIUnits();  tipMap.clear();  transportDemand.clear();  transportSupply.clear();  wagonsNeeded.clear();  goodsWishes.clear();  workerWishes.clear(); } |

**Divergent Class**

The class net.sf.freecol.common.model.Map is a divergent class since it does many more things than just defining the map, as it is also used to make searches directly in it (like in the method net.sf.freecol.common.model.Map.searchMap(final Unit unit, final Tile start,

final GoalDecider goalDecider,

final CostDecider costDecider,

final int maxTurns, final Unit carrier,

final SearchHeuristic searchHeuristic,

final LogBuilder lb),

and net.sf.freecol.common.model.Map.search((final Unit unit, Location start,

final GoalDecider goalDecider,

final CostDecider costDecider,

final int maxTurns, final Unit carrier,

LogBuilder lb))

).

This makes the class more complex than it needs to be.

One way of solving this code smell would be to create other classes that would take care of making such searches using the Map.

(\*The code for the methods mentioned has been already shown in the beginning of this doc\*)