Code smells

Long Method code smell

The following Long Method code smell was identified:

| /\*\*  \* Tries to apply a colony plan given a list of workers.  \*  \* **@param** workers A list of {**@code** Unit}s to assign.  \* **@param** preferScout Prefer to make scouts rather than soldiers.  \* **@param** lb A {**@code** LogBuilder} to log to.  \* **@return** A scratch colony with the workers in place.  \*/  public Colony assignWorkers(List<Unit> workers, boolean preferScout,  LogBuilder lb) {  final GoodsType foodType = spec().getPrimaryFoodType();   // Collect the work location plans. Note that the plans are  // pre-sorted in order of desirability.  final List<GoodsType> produce = getPreferredProduction();  List<WorkLocationPlan> foodPlans = getFoodPlans();  List<WorkLocationPlan> workPlans = getWorkPlans();   // Make a scratch colony to work on.  Colony col = colony.copyColony();  Tile tile = col.getTile();   // Replace the given workers with those in the scratch colony.  List<Unit> otherWorkers = new ArrayList<>(workers);  workers.clear();  for (Unit u : otherWorkers) workers.add(col.getCorresponding(u));   // Move all workers to the tile.  // Also remove equipment, which is safe because no missionaries  // or active pioneers should be on the worker list.  for (Unit u : workers) {  u.setLocation(tile);  col.equipForRole(u, spec().getDefaultRole(), 0);  }   // Move outdoor experts outside if possible.  // Prefer scouts in early game if there are very few.  Role[] outdoorRoles = {  spec().getRoleWithAbility(Ability.IMPROVE\_TERRAIN, null),  null,  spec().getRoleWithAbility(Ability.SPEAK\_WITH\_CHIEF, null)  };  if (preferScout) {  Role tmp = outdoorRoles[1];  outdoorRoles[1] = outdoorRoles[2];  outdoorRoles[2] = tmp;  }  for (Role outdoorRole : outdoorRoles) {  for (Unit u : new ArrayList<>(workers)) {  if (workers.size() <= 1) break;  Role role = outdoorRole;  if (role == null) {  for (Role r : u.getSortedMilitaryRoles()) {  if (u.getType() == r.getExpertUnit() && fullEquipUnit(spec(), u, r, col)) {  workers.remove(u);  lb.add(u.getId(), "(", u.getType().getSuffix(),  ") -> ", r.getSuffix(), "\n");  break;  }  }  } else if (u.getType() == role.getExpertUnit() && fullEquipUnit(spec(), u, role, col)) {  workers.remove(u);  lb.add(u.getId(), "(", u.getType().getSuffix(),  ") -> ", role.getSuffix(), "\n");  }  }  }   // Consider the defence situation.  // **FIXME:** scan for neighbouring hostiles  // Favour low-skill units for defenders, then order experts  // in reverse order of their production on the produce-list,  // and finally by least experience.  final Comparator<Unit> soldierComparator  = Comparator.<Unit>comparingInt(Unit::getSkillLevel)  .thenComparingInt(u ->  (u.getType().getExpertProduction() == null) ? 1 : 0)  .thenComparingInt(u ->  produce.indexOf(u.getType().getExpertProduction()))  .reversed()  .thenComparingInt(Unit::getExperience);  /\*   \* Defence is now handled by the military coordinator.  \*   for (Unit u : sort(workers, soldierComparator)) {  if (workers.size() <= 1) break;  if (!col.isBadlyDefended()) break;  if (u.getSkillLevel() > 0) {  // Stops experts from being used as soldiers  continue;  }  for (Role role : u.getSortedMilitaryRoles()) {  if (role != null && fullEquipUnit(spec(), u, role, col)) {   workers.remove(u);  lb.add(u.getId(), "(", u.getType().getSuffix(), ") -> ",  u.getRoleSuffix(), "\n");  break;  }  }  }  \*/   // Greedy assignment of other workers to plans.  List<AbstractGoods> buildGoods = new ArrayList<>();  BuildableType build = col.getCurrentlyBuilding();  if (build != null) buildGoods.addAll(build.getRequiredGoodsList());  List<WorkLocationPlan> wlps;  WorkLocationPlan wlp;  boolean done = false;  while (!done && !workers.isEmpty()) {  // Decide what to produce: set the work location plan to  // try (wlp), and the list the plan came from so it can  // be recycled if successful (wlps).  wlps = null;  wlp = null;  if (col.getAdjustedNetProductionOf(foodType) > 0) {  // Try to produce something.  wlps = workPlans;  while (!produce.isEmpty()) {  if ((wlp = findPlan(produce.get(0), workPlans)) != null) {  break; // Found a plan to try.  }  produce.remove(0); // Can not produce this goods type  }  }   // See if a plan can be satisfied.  Unit best;  WorkLocation wl;  GoodsType goodsType;  for (;;) {  if (wlp == null) { // Time to use a food plan.  if (foodPlans.isEmpty()) {  lb.add(" Food plans exhausted\n");  done = true;  break;  }  wlps = foodPlans;  wlp = wlps.get(0);  }   String err = null;  goodsType = wlp.getGoodsType();  wl = col.getCorresponding(wlp.getWorkLocation());  best = null;  lb.add(" ", LogBuilder.wide(2, col.getUnitCount()),  ": ", LogBuilder.wide(-15, goodsType.getSuffix()),  "@", LogBuilder.wide(25, locationDescription(wl)),  " => ");   if (!wl.canBeWorked()) {  err = "can not be worked";  } else if (wl.isFull()) {  err = "full";  } else if ((best = ColonyPlan.getBestWorker(wl, goodsType,  workers)) == null) {  err = "no worker found";  }  if (err != null) {  wlps.remove(wlp); // The plan can not be worked, dump it.  lb.add(err, "\n");  break;  }   // Found a suitable worker, place it.  best.setLocation(wl);   // Did the placement break the production bonus?  if (col.getProductionBonus() < 0) {  best.setLocation(tile);  done = true;  lb.add(" broke production bonus\n");  break;  }   // Is the colony going to starve because of this placement?  if (col.getAdjustedNetProductionOf(foodType) < 0) {  int net = col.getAdjustedNetProductionOf(foodType);  int count = col.getGoodsCount(foodType);  if (count / -net < PRODUCTION\_TURNOVER\_TURNS) {  // Too close for comfort. Back out the  // placement and try a food plan, unless this  // was already a food plan.  best.setLocation(tile);  wlp = null;  if (goodsType.isFoodType()) {  lb.add(" starvation (", count, "/", net, ")\n");  done = true;  break;  }  lb.add(" would starve (", count, "/", net, ")\n");  continue;  }  // Otherwise tolerate the food stock running down.  // Rely on the warehouse-exhaustion code to fire  // another rearrangement before units starve.  }   // Check if placing the worker will soon exhaust the  // raw material. Do not reduce raw materials below  // what is needed for a building--- e.g. prevent  // musket production from hogging the tools.  GoodsType raw = goodsType.getInputType();  int rawNeeded = sum(buildGoods, ag -> ag.getType() == raw,  AbstractGoods::getAmount);  if (raw == null  || col.getAdjustedNetProductionOf(raw) >= 0  || (((col.getGoodsCount(raw) - rawNeeded)  / -col.getAdjustedNetProductionOf(raw))  >= PRODUCTION\_TURNOVER\_TURNS)) {  // No raw material problems, the placement  // succeeded. Set the work type, move the  // successful goods type to the end of the produce  // list for later reuse, remove the worker from  // the workers pool, but leave the successful plan  // on its list.  best.changeWorkType(goodsType);  workers.remove(best);  lb.add(" ", best.getId(), "(",  best.getType().getSuffix(),")\n");  if (!goodsType.isFoodType() && produce.remove(goodsType)) {  produce.add(goodsType);  }  break;  }   // Yes, we need more of the raw material. Pull the  // unit out again and see if we can make more.  best.setLocation(tile);   WorkLocationPlan rawWlp = findPlan(raw, workPlans);  if (rawWlp != null) {  // OK, we have an alternate plan. Put the raw  // material at the start of the produce list and  // loop trying to satisfy the alternate plan.  if (produce.remove(raw)) produce.add(0, raw);  wlp = rawWlp;  lb.add(" retry with ", raw.getSuffix(), "\n");  continue;  }   // No raw material available, so we have to give up on  // both the plan and the type of production.  // Hopefully the raw production is positive again and  // we will succeed next time.  wlps.remove(wlp);  produce.remove(goodsType);  lb.add(" needs more ", raw.getSuffix(), "\n");  break;  }  }   // Put the rest of the workers on the tile.  for (Unit u : workers) {  if (u.getLocation() != tile) u.setLocation(tile);  }   // Check for failure to assign any workers. This happens when:  // - there are no useful food plans  // - in which case look for a `harmless' place and add one worker  // - food is low, and perhaps partly eaten by horses, and no  // unit can \*improve\* production by being added.  // - find a place to produce food that at least avoids  // starvation and add one worker.  if (col.getUnitCount() == 0) {  if (getFoodPlans().isEmpty()) {  for (WorkLocation wl : col.getAvailableWorkLocationsList()) {  for (Unit u : new ArrayList<>(workers)) {  for (GoodsType type : libertyGoodsTypes) {  if (wl.canAdd(u)  && wl.getPotentialProduction(type,  u.getType()) > 0) {  u.setLocation(wl);  u.changeWorkType(type);  workers.remove(u);  break locations;  }  }  }  }  } else {  for (WorkLocationPlan w : getFoodPlans()) {  GoodsType goodsType = w.getGoodsType();  WorkLocation wl = col.getCorresponding(w.getWorkLocation());  for (Unit u : new ArrayList<>(workers)) {  GoodsType oldWork = u.getWorkType();  u.setLocation(wl);  u.changeWorkType(goodsType);  if (col.getAdjustedNetProductionOf(foodType) >= 0) {  lb.add(" Subsist with ", u, "\n");  workers.remove(u);  break plans;  }  u.setLocation(tile);  u.changeWorkType(oldWork);  }  }  }  }   // The greedy algorithm works reasonably well, but will  // misplace experts when they are more productive at the  // immediately required task than a lesser unit, not knowing  // that a requirement for their speciality will subsequently  // follow. Do a cleanup pass to sort these out.  List<Unit> experts = new ArrayList<>();  List<Unit> nonExperts = new ArrayList<>();  for (Unit u : col.getUnitList()) {  if (u.getType().getExpertProduction() != null) {  if (u.getType().getExpertProduction() != u.getWorkType()) {  experts.add(u);  }  } else {  nonExperts.add(u);  }  }  int expert = 0;  Iterator<Unit> expertIterator = experts.iterator();  while (expertIterator.hasNext()) {  Unit u1 = expertIterator.next();  Unit other = u1.trySwapExpert(experts);  if (other != null) {  lb.add(" Swapped ", u1.getId(), "(",  u1.getType().getSuffix(), ") for ", other, "\n");  expertIterator.remove();  } else if ((other = u1.trySwapExpert(nonExperts)) != null) {  lb.add(" Swapped ", u1.getId(), "(",  u1.getType().getSuffix(), ") for ", other, "\n");  expertIterator.remove();  }  }  for (Unit u : new ArrayList<>(workers)) {  GoodsType work = u.getType().getExpertProduction();  if (work != null) {  Unit other = u.trySwapExpert(col.getUnitList());  if (other != null) {  lb.add(" Swapped ", u.getId(), "(",  u.getType().getSuffix(), ") for ", other, "\n");  workers.remove(u);  workers.add(other);  }  }  }   // Rearm what remains as far as possible.  for (Unit u : sort(workers, soldierComparator)) {  if (u.getSkillLevel() > 0) continue;  for (Role role : u.getSortedMilitaryRoles()) {  if (fullEquipUnit(spec(), u, role, col)) {  lb.add(" ", u.getId(), "(", u.getType().getSuffix(),  ") -> ", u.getRoleSuffix(), "\n");  workers.remove(u);  break;  }  }  }  for (Unit u : transform(col.getUnits(), u -> !u.hasDefaultRole())) {  logger.warning("assignWorkers bogus role for " + u);  u.changeRole(spec().getDefaultRole(), 0);  }   // Log and return the scratch colony on success.  // Otherwise abandon this rearrangement, disposing of the  // scratch colony and returning null.  for (Unit u : workers) {  lb.add(" ", u.getId(), "(", u.getType().getSuffix(),  ") -> UNUSED\n");  }  if (col.getUnitCount() <= 0) col = null;  return col;  } |
| --- |

This code is located at *net.sf.freecol.server.ai.ColonyPlan.assignWorkers(List<Unit>, boolean, LogBuilder)*, and was selected based on size and complexity. As you can see, this method is extremely long and deals with multiple things that should not be addressed in the same method. For example, in the code we have a section where we try to assign outdoors experts to outdoors work, a section where we consider the defense of the colony, a huge section where we do a greedy assignment of the remaining workers (which in itself contains checks to verify if a plan can be satisfied and checks to verify if a placement breaks the production bonus, among many other verifications), a cleanup pass in the end of the assignments and code to rearm any remaining unit in the end.

A possible refactoring to remove this smell would be breaking up this method into several other smaller methods that each deal with a small issue. For example, all the verifications inside of the greedy assignment could be their own separate methods. There is also a part of the code inside the greedy assignment that deals with the exhaustion of raw materials. This code can be dealt with separately, inside their own methods.

Data Class code smell

The following Data Class code smell was identified:

| public static final class CombatResult {    private final List<CombatEffectType> effects;  private final int attackerHitpointsAfter;  private final int defenderHitpointsAfter;    public CombatResult(List<CombatEffectType> effects) {  this.effects = Objects.requireNonNull(effects);  this.attackerHitpointsAfter = -1;  this.defenderHitpointsAfter = -1;  }    public CombatResult(List<CombatEffectType> effects, int attackerHitpointsAfter, int defenderHitpointsAfter) {  this.effects = Objects.requireNonNull(effects);  this.attackerHitpointsAfter = attackerHitpointsAfter;  this.defenderHitpointsAfter = defenderHitpointsAfter;  }    public List<CombatEffectType> getEffects() {  return effects;  }    public boolean isAttackerHitpointsAffected() {  return attackerHitpointsAfter >= 0;  }    public boolean isDefenderHitpointsAffected() {  return defenderHitpointsAfter >= 0;  }    public int getAttackerHitpointsAfter() {  return attackerHitpointsAfter;  }    public int getDefenderHitpointsAfter() {  return defenderHitpointsAfter;  }  } |
| --- |

This class can be found in *net.sf.freecol.common.model.CombatModel.CombatResult*, and was selected because it only contains data and no functionality (only getter and setter methods). A possible refactoring to remove this smell would be to make this an abstract class and to add the combat logic code to the extensions of this class, one extension per combat model.

Switch Statement code smell

The following Switch Statement code smell was identified:

| (more code...)  for (CombatEffectType cr : crs) {  boolean ok;  switch (cr) {  case AUTOEQUIP\_UNIT:  ok = isAttack && settlement != null;  if (ok) {  csAutoequipUnit(defenderUnit, settlement, cs);  }  break;  case BURN\_MISSIONS:  ok = isAttack && result == CombatEffectType.WIN  && natives != null  && isEuropean() && defenderPlayer.isIndian();  if (ok) {  defenderTileDirty |= natives.hasMissionary(this);  csBurnMissions(attackerUnit, natives, cs);  }  break;  case CAPTURE\_AUTOEQUIP:  ok = isAttack && result == CombatEffectType.WIN  && settlement != null;  if (ok) {  csCaptureAutoEquip(attackerUnit, defenderUnit, cs);  attackerTileDirty = defenderTileDirty = true;  }  break;  case CAPTURE\_COLONY:  ok = isAttack && result == CombatEffectType.WIN  && colony != null  && (isEuropean() || isUndead()) && (defenderPlayer.isEuropean() || defenderPlayer.isUndead());  if (ok) {  csCaptureColony(attackerUnit, (ServerColony)colony,  random, cs);  attackerTileDirty = defenderTileDirty = false;  moveAttacker = true;  defenderTension += Tension.TENSION\_ADD\_MAJOR;  }  break;  case CAPTURE\_CONVERT:  ok = isAttack && result == CombatEffectType.WIN  && natives != null  && isEuropean() && defenderPlayer.isIndian();  if (ok) {  csCaptureConvert(attackerUnit, natives, random, cs);  attackerTileDirty = true;  }  break;  case CAPTURE\_EQUIP:  ok = isAttack && result != CombatEffectType.NO\_RESULT;  if (ok) {  if (result == CombatEffectType.WIN) {  csCaptureEquip(attackerUnit, defenderUnit, cs);  } else {  csCaptureEquip(defenderUnit, attackerUnit, cs);  }  attackerTileDirty = defenderTileDirty = true;  }  break;  case CAPTURE\_UNIT:  ok = isAttack && result != CombatEffectType.NO\_RESULT;  if (ok) {  if (result == CombatEffectType.WIN) {  csCaptureUnit(attackerUnit, defenderUnit, cs);  } else {  csCaptureUnit(defenderUnit, attackerUnit, cs);  }  attackerTileDirty = true;  defenderTileDirty = false; // Added in csCaptureUnit  }  break;  case DAMAGE\_COLONY\_SHIPS:  ok = isAttack && result == CombatEffectType.WIN  && colony != null;  if (ok) {  csDamageColonyShips(attackerUnit, colony, cs);  defenderTileDirty = true;  }  break;  case DAMAGE\_SHIP\_ATTACK:  ok = isAttack && result != CombatEffectType.NO\_RESULT  && ((result == CombatEffectType.WIN) ? defenderUnit  : attackerUnit).isNaval();  if (ok) {  if (result == CombatEffectType.WIN) {  csDamageShipAttack(attackerUnit, defenderUnit, cs);  defenderTileDirty = true;  } else {  csDamageShipAttack(defenderUnit, attackerUnit, cs);  attackerTileDirty = true;  }  }  break;  case DAMAGE\_SHIP\_BOMBARD:  ok = isBombard && result == CombatEffectType.WIN  && defenderUnit.isNaval();  if (ok) {  csDamageShipBombard(attackerSettlement, defenderUnit, cs);  defenderTileDirty = true;  }  break;  case DEMOTE\_UNIT:  ok = isAttack && result != CombatEffectType.NO\_RESULT;  if (ok) {  if (result == CombatEffectType.WIN) {  csDemoteUnit(attackerUnit, defenderUnit, cs);  defenderTileDirty = true;  } else {  csDemoteUnit(defenderUnit, attackerUnit, cs);  attackerTileDirty = true;  }  }  break;  case DESTROY\_COLONY:  ok = isAttack && result == CombatEffectType.WIN  && colony != null  && isIndian() && defenderPlayer.isEuropean();  if (ok) {  csDestroyColony(attackerUnit, colony, random, cs);  attackerTileDirty = defenderTileDirty = true;  moveAttacker = true;  attackerTension -= Tension.TENSION\_ADD\_NORMAL;  defenderTension += Tension.TENSION\_ADD\_MAJOR;  }  break;  case DESTROY\_SETTLEMENT:  ok = isAttack && result == CombatEffectType.WIN  && natives != null  && defenderPlayer.isIndian();  if (ok) {  burnedNativeCapital = settlement.isCapital();  csDestroySettlement(attackerUnit, natives, random, cs);  attackerTileDirty = defenderTileDirty = true;  moveAttacker = true;  attackerTension -= Tension.TENSION\_ADD\_NORMAL;  if (!burnedNativeCapital) {  defenderTension += Tension.TENSION\_ADD\_MAJOR;  }  }  break;  case EVADE\_ATTACK:  ok = isAttack && result == CombatEffectType.NO\_RESULT  && defenderUnit.isNaval();  if (ok) {  csEvadeAttack(attackerUnit, defenderUnit, cs);  }  break;  case EVADE\_BOMBARD:  ok = isBombard && result == CombatEffectType.NO\_RESULT  && defenderUnit.isNaval();  if (ok) {  csEvadeBombard(attackerSettlement, defenderUnit, cs);  }  break;  case LOOT\_SHIP:  ok = isAttack && result != CombatEffectType.NO\_RESULT  && attackerUnit.isNaval() && defenderUnit.isNaval();  if (ok) {  if (result == CombatEffectType.WIN) {  csLootShip(attackerUnit, defenderUnit, cs);  } else {  csLootShip(defenderUnit, attackerUnit, cs);  }  }  break;  case LOSE\_AUTOEQUIP:  ok = isAttack && result == CombatEffectType.WIN  && settlement != null;  if (ok) {  csLoseAutoEquip(attackerUnit, defenderUnit, cs);  defenderTileDirty = true;  }  break;  case LOSE\_EQUIP:  ok = isAttack && result != CombatEffectType.NO\_RESULT;  if (ok) {  if (result == CombatEffectType.WIN) {  csLoseEquip(attackerUnit, defenderUnit, cs);  defenderTileDirty = true;  } else {  csLoseEquip(defenderUnit, attackerUnit, cs);  attackerTileDirty = true;  }  }  break;  case PILLAGE\_COLONY:  ok = isAttack && result == CombatEffectType.WIN  && colony != null  && isIndian() && defenderPlayer.isEuropean();  if (ok) {  csPillageColony(attackerUnit, colony, random, cs);  defenderTileDirty = true;  attackerTension -= Tension.TENSION\_ADD\_NORMAL;  }  break;  case PROMOTE\_UNIT:  ok = isAttack && result != CombatEffectType.NO\_RESULT;  if (ok) {  if (result == CombatEffectType.WIN) {  csPromoteUnit(attackerUnit, cs);  attackerTileDirty = true;  } else {  csPromoteUnit(defenderUnit, cs);  defenderTileDirty = true;  }  }  break;  case SINK\_COLONY\_SHIPS:  ok = isAttack && result == CombatEffectType.WIN  && colony != null;  if (ok) {  csSinkColonyShips(attackerUnit, colony, cs);  defenderTileDirty = true;  }  break;  case SINK\_SHIP\_ATTACK:  ok = isAttack && result != CombatEffectType.NO\_RESULT  && ((result == CombatEffectType.WIN) ? defenderUnit  : attackerUnit).isNaval();  if (ok) {  if (result == CombatEffectType.WIN) {  csSinkShipAttack(attackerUnit, defenderUnit, cs);  defenderTileDirty = true;  } else {  csSinkShipAttack(defenderUnit, attackerUnit, cs);  attackerTileDirty = true;  }  }  break;  case SINK\_SHIP\_BOMBARD:  ok = isBombard && result == CombatEffectType.WIN  && defenderUnit.isNaval();  if (ok) {  csSinkShipBombard(attackerSettlement, defenderUnit, cs);  defenderTileDirty = true;  }  break;  case SLAUGHTER\_UNIT:  ok = isAttack && result != CombatEffectType.NO\_RESULT;  if (ok) {  if (result == CombatEffectType.WIN) {  csSlaughterUnit(attackerUnit, defenderUnit, cs);  defenderTileDirty = true;  attackerTension -= Tension.TENSION\_ADD\_NORMAL;  defenderTension += getSlaughterTension(defenderUnit);  } else {  csSlaughterUnit(defenderUnit, attackerUnit, cs);  attackerTileDirty = true;  attackerTension += getSlaughterTension(attackerUnit);  defenderTension -= Tension.TENSION\_ADD\_NORMAL;  }  }  break;  default:  ok = false;  break;  }  (more code...) |
| --- |

This code is a snippet of the method csCombat, located at *net.sf.freecol.server.model.ServerPlayer* (which can also serve as a long method code smell, but we will focus on the switch statement). This switch statement is a code smell because it will lead to issues when we want to make changes or add new combat effects. We can also observe code repetition in some cases. A possible refactoring would be to use polymorphism to take care of combat effects. For example, we could have an abstract combat effect class that in turn would be extended by each combat effect (and some effects could extend other effects), and the csCombat method would simply call a method of the abstract class. This would fix both the repeated code issues and the issues that arise due to the switch case.

Tiago Sousa, 63324