Code Smells

# Duplicated Code

net.sf.freecol.common.model.Colony.getsTurnsToComplete(BuildableType buildable)

| /\*\* \* Returns how many turns it would take to build the given \* {@code BuildableType}. \* \* @param buildable The {@code BuildableType} to build. \* @return The number of turns to build the buildable, negative if \* some goods are not being built, UNDEFINED if none is. \*/ public int getTurnsToComplete(BuildableType buildable) {  return getTurnsToComplete(buildable, null); }  /\*\* \* Returns how many turns it would take to build the given \* {@code BuildableType}. \* \* @param buildable The {@code BuildableType} to build. \* @param needed The {@code AbstractGoods} needed to continue \* the build. \* @return The number of turns to build the buildable (which may \* be zero, UNDEFINED if no useful work is being done, negative \* if some requirement is or will block completion (value is \* the negation of (turns-to-blockage + 1), and if the needed \* argument is supplied it is set to the goods deficit). \*/ public int getTurnsToComplete(BuildableType buildable,  AbstractGoods needed) {  final List<AbstractGoods> required = buildable.getRequiredGoodsList();  int turns = 0, satisfied = 0, failing = 0, underway = 0;   ProductionInfo info = productionCache.getProductionInfo(buildQueue);  for (AbstractGoods ag : required) {  final GoodsType type = ag.getType();  final int amountNeeded = ag.getAmount();  final int amountAvailable = getGoodsCount(type);  if (amountAvailable >= amountNeeded) {  satisfied++;  continue;  }  int production = productionCache.getNetProductionOf(type);  if (info != null) {  AbstractGoods consumption = find(info.getConsumption(),  AbstractGoods.matches(type));  if (consumption != null) {  // add the amount the build queue itself will consume  production += consumption.getAmount();  }  }  if (production <= 0) {  failing++;  if (needed != null) {  needed.setType(type);  needed.setAmount(amountNeeded - amountAvailable);  }  continue;  }   underway++;  int amountRemaining = amountNeeded - amountAvailable;  int eta = amountRemaining / production;  if (amountRemaining % production != 0) eta++;  turns = Math.max(turns, eta);  }   return (satisfied + underway == required.size()) ? turns // Will finish  : (failing == required.size()) ? UNDEFINED // Not even trying  : -(turns + 1); // Blocked by something } |
| --- |

The above code from the class Colony qualifies as the Code Smell Duplicated Code as they consist of two methods that perform essentially the same functionality. The main difference between the methods resides in the amount of parameters, as the first instance of getTurnsToComplete() only has one parameter, whilst the second occurrence has 2. A way of fixing this code smell would be to converge these methods, resulting in a single method, as the first instance could be written as a circumstance covered by the second in which the object AbstractGoods equals null.

# 

# Message Chains

net.sf.freecol.client.control.ServerColony.csNewTurnWarnings(Random random, LogBuilder lb, ChangeSet cs)

| /\*\* \* Do the checks for user warnings that must wait for all player \* settlements, units and whatever to stabilize. Along the way, \* throw away excess goods. \* \* For example, a pioneer might clear a colony tile and change the \* lumber amount. \* \* @param random A {@code Random} number source. \* @param lb A {@code LogBuilder} to log to. \* @param cs A {@code ChangeSet} to update. \*/ public void csNewTurnWarnings(Random random, LogBuilder lb, ChangeSet cs) {  final Specification spec = getSpecification();  final BuildQueue<?>[] queues = new BuildQueue<?>[] {  this.buildQueue, this.populationQueue };  final GoodsContainer container = getGoodsContainer();   for (WorkLocation wl : getCurrentWorkLocationsList()) {  if (wl instanceof ServerBuilding) {  // **FIXME:** generalize to other WorkLocations?  ((ServerBuilding)wl).csCheckMissingInput(getProductionInfo(wl),  cs);  }  }   for (BuildQueue<?> queue : queues) {  ProductionInfo info = getProductionInfo(queue);  if (info == null) continue;  if (info.getConsumption().isEmpty()) {  BuildableType build = queue.getCurrentlyBuilding();  if (build != null) {  AbstractGoods needed = new AbstractGoods();  int complete = getTurnsToComplete(build, needed);  // Warn if about to fail, or if no useful progress  // towards completion is possible.  if (complete == -2 || complete == -1) {  cs.addMessage(owner,  new ModelMessage(MessageType.MISSING\_GOODS,  "model.colony.buildableNeedsGoods",  this, build)  .addName("%colony%", getName())  .addNamed("%buildable%", build)  .addAmount("%amount%", needed.getAmount())  .addNamed("%goodsType%", needed.getType()));  }  }  }  }   // Throw away goods there is no room for, and warn about  // levels that will be exceeded next turn  final int limit = getWarehouseCapacity();  final int adjustment = limit / GoodsContainer.CARGO\_SIZE;  for (Goods goods : transform(getCompactGoodsList(),  AbstractGoods::isStorable)) {  final GoodsType type = goods.getType();  final ExportData exportData = getExportData(type);  final int low = exportData.getLowLevel() \* adjustment;  final int high = exportData.getHighLevel() \* adjustment;  final int amount = goods.getAmount();  final int oldAmount = container.getOldGoodsCount(type);   if (amount < low && oldAmount >= low  && type != spec.getPrimaryFoodType()) {  cs.addMessage(owner,  new ModelMessage(MessageType.WAREHOUSE\_CAPACITY,  "model.colony.warehouseEmpty",  this, type)  .addNamed("%goods%", type)  .addAmount("%level%", low)  .addName("%colony%", getName()));  continue;  }  if (type.limitIgnored()) continue;  String messageId = null;  int waste = 0;  if (amount > limit) {  // limit has been exceeded  waste = amount - limit;  container.removeGoods(type, waste);  messageId = "model.colony.warehouseWaste";  } else if (amount == limit && oldAmount < limit) {  // limit has been reached during this turn  messageId = "model.colony.warehouseOverfull";  } else if (amount > high && oldAmount <= high) {  // high-water-mark has been reached this turn  messageId = "model.colony.warehouseFull";  }  if (messageId != null) {  cs.addMessage(owner,  new ModelMessage(MessageType.WAREHOUSE\_CAPACITY,  messageId, this, type)  .addNamed("%goods%", type)  .addAmount("%waste%", waste)  .addAmount("%level%", high)  .addName("%colony%", getName()));  }   // No problem this turn, but what about the next?  if (!(exportData.getExported()  && hasAbility(Ability.EXPORT)  && owner.canTrade(type, Market.Access.CUSTOM\_HOUSE))  && amount <= limit) {  int loss = amount + getNetProductionOf(type) - limit;  if (loss > 0) {  cs.addMessage(owner,  new ModelMessage(MessageType.WAREHOUSE\_CAPACITY,  "model.colony.warehouseSoonFull",  this, type)  .addNamed("%goods%", goods)  .addName("%colony%", getName())  .addAmount("%amount%", loss));  }  }  }   // If a build queue is empty, check that we are not producing  // any goods types useful for BuildableTypes, except if that  // type is the input to some other form of production. (Note:  // isBuildingMaterial is also true for goods used to produce  // role-equipment, hence neededForBuildableType). Such  // production probably means we forgot to reset the build  // queue. Thus, if hammers are being produced it is worth  // warning about, but not if producing tools.  if (any(queues, BuildQueue::isEmpty)  && any(spec.getGoodsTypeList(), g ->  (g.isBuildingMaterial()  && !g.isRawMaterial()  && !g.isBreedable()  && getAdjustedNetProductionOf(g) > 0  && neededForBuildableType(g)))) {  cs.addMessage(owner,  new ModelMessage(MessageType.BUILDING\_COMPLETED,  "model.colony.notBuildingAnything", this)  .addName("%colony%", getName()));  } } |
| --- |

The above code has various instances of the Code Smell Message Chains as there are many subsequent calls in order to get another object, for example:

| ModelMessage(MessageType.WAREHOUSE\_CAPACITY,  messageId, this, type)  .addNamed("%goods%", type)  .addAmount("%waste%", waste)  .addAmount("%level%", high)  .addName("%colony%", getName())) |
| --- |

A way of fixing this Code Smell is to allow direct access to the required object instead of chaining it through various method calls.

# 

# 

# Long Method

net.sf.freecol.server.model.ServerPlayer.csCombat(FreeColGameObject attacker, FreeColGameObject defender, List<CombatEffectType> crs, Random random, ChangeSet cs)

| /\*\* \* Combat. \* \* **@param** attacker The {**@code** FreeColGameObject} that is attacking. \* **@param** defender The {**@code** FreeColGameObject} that is defending. \* **@param** crs A list of {**@code** CombatResult}s defining the result. \* **@param** random A pseudo-random number source. \* **@param** cs A {**@code** ChangeSet} to update. \*/ public void csCombat(FreeColGameObject attacker,  FreeColGameObject defender,  List<CombatEffectType> crs,  Random random,  ChangeSet cs) {  CombatModel combatModel = getGame().getCombatModel();  boolean isAttack = combatModel.combatIsAttack(attacker, defender);  boolean isBombard = combatModel.combatIsBombard(attacker, defender);  Unit attackerUnit = null;  Settlement attackerSettlement = null;  Tile attackerTile = null;  Unit defenderUnit = null;  Player defenderPlayer = null;  Tile defenderTile = null;  if (isAttack) {  attackerUnit = (Unit)attacker;  //attackerPlayer = attackerUnit.getOwner();  attackerTile = attackerUnit.getTile();  defenderUnit = (Unit)defender;  defenderPlayer = defenderUnit.getOwner();  defenderTile = defenderUnit.getTile();  boolean bombard = attackerUnit.hasAbility(Ability.BOMBARD);  cs.addAttribute(See.only(this), "sound",  (attackerUnit.isNaval()) ? "sound.attack.naval"  : (bombard) ? "sound.attack.artillery"  : (attackerUnit.isMounted()) ? "sound.attack.mounted"  : "sound.attack.foot");  if (attackerUnit.getOwner().isIndian()  && defenderPlayer.isEuropean()  && defenderUnit.getLocation().getColony() != null  && !defenderPlayer.atWarWith(attackerUnit.getOwner())) {  StringTemplate attackerNation  = attackerUnit.getApparentOwnerName();  Colony colony = defenderUnit.getLocation().getColony();  cs.addMessage(defenderPlayer,  new ModelMessage(ModelMessage.MessageType.COMBAT\_RESULT,  "combat.raid.ours", colony)  .addName("%colony%", colony.getName())  .addStringTemplate("%nation%", attackerNation));  }  } else if (isBombard) {  attackerSettlement = (Settlement)attacker;  attackerTile = attackerSettlement.getTile();  defenderUnit = (Unit)defender;  defenderPlayer = defenderUnit.getOwner();  defenderTile = defenderUnit.getTile();  cs.addAttribute(See.only(this), "sound", "sound.attack.bombard");  } else {  throw new RuntimeException("Bogus combat: " + attacker  + " v " + defender);  }  assert defenderTile != null;   // If the combat results were not specified (usually the case),  // query the combat model.  CombatResult combatResult = null;  if (crs == null) {  combatResult = combatModel.generateAttackResult(random, attacker, defender);  crs = combatResult.getEffects();  }  if (crs.isEmpty()) {  throw new RuntimeException("empty attack result: " + this);  }    // Extract main result, insisting it is one of the fundamental cases,  // and add the animation.  // Set vis so that loser always sees things.  // **FIXME:** Bombard animations  See vis; // Visibility that insists on the loser seeing the result.  CombatEffectType result = crs.remove(0);  switch (result) {  case NO\_RESULT:  vis = See.perhaps();  break; // Do not animate if there is no result.  case WIN:  vis = See.perhaps().always(defenderPlayer);  if (isAttack) {  if (attackerTile == null  || attackerTile == defenderTile  || !attackerTile.isAdjacent(defenderTile)) {  logger.warning("Bogus attack from " + attackerTile  + " to " + defenderTile);  } else {  cs.addAttack(vis, attackerUnit, defenderUnit, true);  }  }  break;  case LOSE:  vis = See.perhaps().always(this);  if (isAttack) {  if (attackerTile == null  || attackerTile == defenderTile  || !attackerTile.isAdjacent(defenderTile)) {  logger.warning("Bogus attack from " + attackerTile  + " to " + defenderTile);  } else {  cs.addAttack(vis, attackerUnit, defenderUnit, false);  }  }  break;  default:  throw new IllegalStateException("generateAttackResult returned: "  + result);  }    // Now process the details.  boolean attackerTileDirty = false;  boolean defenderTileDirty = false;  boolean moveAttacker = false;  boolean burnedNativeCapital = false;  Settlement settlement = defenderTile.getSettlement();  Colony colony = defenderTile.getColony();  IndianSettlement natives = (settlement instanceof IndianSettlement)  ? (IndianSettlement) settlement  : null;  int attackerTension = 0;  int defenderTension = 0;    if (combatResult != null && combatResult.isAttackerHitpointsAffected()) {  attackerUnit.setHitPoints(combatResult.getAttackerHitpointsAfter());  attackerTileDirty = true;  }    if (combatResult != null && combatResult.isDefenderHitpointsAffected()) {  defenderUnit.setHitPoints(combatResult.getDefenderHitpointsAfter());  defenderTileDirty = true;  }    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;  }  if (!ok) {  throw new IllegalStateException("Attack (result=" + result  + ") has bogus subresult: "  + cr);  }  }   // Handle stance and tension.  // - Privateers do not provoke stance changes but can set the  // attackedByPrivateers flag  // - Attacks among Europeans imply war  // - Burning of a native capital results in surrender  // - Other attacks involving natives do not imply war, but  // changes in Tension can drive Stance, however this is  // decided by the native AI in their turn so just adjust tension.  if (attacker.hasAbility(Ability.PIRACY)) {  if (!defenderPlayer.getAttackedByPrivateers()) {  defenderPlayer.setAttackedByPrivateers(true);  cs.addPartial(See.only(defenderPlayer), defenderPlayer,  "attackedByPrivateers", Boolean.TRUE.toString());  }  } else if (defender.hasAbility(Ability.PIRACY)) {  ; // do nothing  } else if (burnedNativeCapital) {  defenderPlayer.getTension(this).setValue(Tension.SURRENDERED);  // **FIXME:** just the tension  cs.add(See.perhaps().always(this), defenderPlayer);  csChangeStance(Stance.PEACE, defenderPlayer, true, cs);  for (IndianSettlement is : transform(defenderPlayer.getIndianSettlements(),  is -> is.hasContacted(this))) {  is.getAlarm(this).setValue(Tension.SURRENDERED);  // Only update attacker with settlements that have  // been seen, as contact can occur with its members.  if (hasExplored(is.getTile())) {  cs.add(See.perhaps().always(this), is);  } else {  cs.add(See.only(defenderPlayer), is);  }  }  } else if (isEuropean() && defenderPlayer.isEuropean()) {  csChangeStance(Stance.WAR, defenderPlayer, true, cs);  } else { // At least one player is non-European  if (isEuropean()) {  csChangeStance(Stance.WAR, defenderPlayer, true, cs);  } else if (isIndian()) {  if (result == CombatEffectType.WIN) {  attackerTension -= Tension.TENSION\_ADD\_MINOR;  } else if (result == CombatEffectType.LOSE) {  attackerTension += Tension.TENSION\_ADD\_MINOR;  }  }  if (defenderPlayer.isEuropean()) {  ((ServerPlayer)defenderPlayer).csChangeStance(Stance.WAR, this, true, cs);  } else if (defenderPlayer.isIndian()) {  if (result == CombatEffectType.WIN) {  defenderTension += Tension.TENSION\_ADD\_MINOR;  } else if (result == CombatEffectType.LOSE) {  defenderTension -= Tension.TENSION\_ADD\_MINOR;  }  }  if (attackerTension != 0) {  this.csModifyTension(defenderPlayer,  attackerTension, cs);//+til  }  if (defenderTension != 0) {  ((ServerPlayer)defenderPlayer).csModifyTension(this,  defenderTension, cs);//+til  }  }   // Move the attacker if required.  if (moveAttacker) {  attackerUnit.setMovesLeft(attackerUnit.getInitialMovesLeft());  ((ServerUnit) attackerUnit).csMove(defenderTile, random, cs);  attackerUnit.setMovesLeft(0);  // Move adds in updates for the tiles, but...  attackerTileDirty = defenderTileDirty = false;  // ...with visibility of perhaps().  // Thus the defender might see the change,  // but because its settlement is gone it also might not.  // So add in another defender-specific update.  // The worst that can happen is a duplicate update.  cs.add(See.only(defenderPlayer), defenderTile);  } else if (isAttack) {  // The Revenger unit can attack multiple times, so spend  // at least the eventual cost of moving to the tile.  // Other units consume the entire move.  if (attacker.hasAbility(Ability.MULTIPLE\_ATTACKS)) {  int movecost = attackerUnit.getMoveCost(defenderTile);  attackerUnit.setMovesLeft(attackerUnit.getMovesLeft()  - movecost);  } else {  attackerUnit.setMovesLeft(0);  }  if (!attackerTileDirty) {  cs.addPartial(See.only(this), attacker,  "movesLeft", String.valueOf(attackerUnit.getMovesLeft()));  }  }   // Make sure we always update the attacker and defender tile  // if it is not already done yet.  if (attackerTileDirty) {  if (attackerSettlement != null) cs.remove(attackerSettlement);  cs.add(vis, attackerTile);  }  if (defenderTileDirty) {  if (settlement != null) cs.remove(settlement);  cs.add(vis, defenderTile);  } } |
| --- |

This method qualifies as the Code Smell Long Method as it undertakes too much and requires many lines of comments in order to understand what is being done. A way to fix this Code Smell would be to subdivide this method into smaller methods with their own comments that would then be called by the main method in order to achieve the necessary outcome, resulting in more comprehensible code.

Beatriz de Caiola Machado, 62551