**Code Smells**

**Long Method**(net.sf.freecol.server.ai.mission.Mission.travelToTarget(Location, CostDecider, LogBuilder))

protected MoveType travelToTarget(Location target, CostDecider costDecider,

LogBuilder lb) {

if (target == null) return MoveType.*MOVE\_ILLEGAL*;

final Tile targetTile = target.getTile();

if (!(target instanceof Europe) && targetTile == null) {

throw new RuntimeException("Target neither Europe nor Tile: "

+ target);

}

final Unit unit = getUnit();

AIUnit aiCarrier = aiUnit.getTransport();

final Map map = unit.getGame().getMap();

PathNode path = null;

boolean useTransport = false;

target = Location.*upLoc*(target);

// Consider where the unit is starting.

if (unit.isAtSea()) {

// Wait for carrier to arrive on the map or in Europe.

lb.add(", at sea");

return MoveType.*MOVE\_HIGH\_SEAS*;

} else if (unit.isOnCarrier()) {

// Transport mission will disembark the unit when it

// arrives at the drop point.

lb.add(", on carrier");

return MoveType.*MOVE\_NO\_ACCESS\_EMBARK*;

} else if (unit.isAtLocation(target)) {

// Arrived!

return MoveType.*MOVE*;

} else if (unit.isInEurope()) {

// Leave, or require transport.

if (!unit.getOwner().canMoveToEurope()) {

lb.add(", impossible move from Europe");

return MoveType.*MOVE\_ILLEGAL*;

}

if (unit.getType().canMoveToHighSeas()) {

unit.setDestination(target);

if (AIMessage.*askMoveTo*(aiUnit, map)) {

lb.add(", sailed for ", target);

return MoveType.*MOVE\_HIGH\_SEAS*;

} else {

lb.add(", failed to sail for ", target);

return MoveType.*MOVE\_ILLEGAL*;

}

}

useTransport = true;

} else if (!unit.hasTile()) {

// Fail!

return MoveType.*MOVE\_ILLEGAL*;

} else {

// On map. Either find a path or decide to use transport.

if (target instanceof Europe) {

// Going to Europe.

if (!unit.getOwner().canMoveToEurope()) {

lb.add(", impossible move to Europe");

return MoveType.*MOVE\_ILLEGAL*;

}

if (!unit.getType().canMoveToHighSeas()

|| aiCarrier != null) {

useTransport = true;

} else {

path = unit.findPath(unit.getLocation(), target,

null, costDecider, null);

}

} else if (aiCarrier != null) {

// Transport already allocated.

useTransport = true;

} else if (!unit.getType().canMoveToHighSeas()

&& !Map.*isSameContiguity*(target, unit.getLocation())) {

// Transport necessary.

useTransport = true;

} else {

// Should not need transport within the same contiguity.

path = unit.findPath(unit.getLocation(), target,

null, costDecider, null);

}

}

if (useTransport) {

if (aiCarrier != null) {

// A carrier has been assigned. Try to go to the

// collection point.

Location pick;

TransportMission tm;

boolean waiting = false;

PathNode ownPath;

int pathTurns, ownTurns;

if ((tm = aiCarrier.getMission(TransportMission.class)) == null) {

// Carrier has no transport mission?!? Bogus.

lb.add(", had bogus carrier ", aiCarrier.getUnit());

*logger*.warning(unit + " has transport " + aiCarrier

+ " without transport mission");

aiUnit.dropTransport();

aiCarrier = null;

} else if ((pick = tm.getTransportTarget(aiUnit)) == null) {

// No collection point for this unit? Bogus.

lb.add(", had bogus transport on ", aiCarrier.getUnit());

*logger*.warning(unit + " has transport " + aiCarrier

+ " with transport mission but null transport target\n"

+ tm.toFullString());

aiUnit.dropTransport();

aiCarrier = null;

} else if (Map.*isSameLocation*(pick, unit.getLocation())) {

// Waiting for the carrier at the collection point.

waiting = true;

} else if ((path = unit.findPath(unit.getLocation(), pick,

null, costDecider, null)) == null) {

// No path to the collection point.

lbAt(lb);

lb.add(", no path to meet ", aiCarrier.getUnit(),

" at ", pick);

path = unit.findPath(unit.getLocation(), target,

null, costDecider, null);

if (path == null) {

// Unable to fall back to going direct.

// Return failure in the hope that it is a

// transient blockage.

return MoveType.*MOVE\_NO\_TILE*;

}

// Fall back to going direct to the target.

lb.add(", dropped carrier");

aiUnit.dropTransport();

aiCarrier = null;

useTransport = false;

} else if ((ownPath = unit.findPath(unit.getLocation(),

target, null, costDecider, null)) == null

|| (ownTurns = ownPath.getTotalTurns())

> (pathTurns = path.getTotalTurns())) {

// Either there is no direct path to the target or

// a path exists but takes longer than using the

// carrier. This confirms that it is not only

// possible to travel to the collection point, it

// is also the best plan.

MoveType ret = followMapPath(path.next, lb);

if (ret != MoveType.*MOVE*) return ret;

waiting = true; // Arrived for collection.

} else {

// It is quicker to cancel the transport and go to

// the target directly.

lb.add(", dropping carrier", aiCarrier.getUnit(),

" as it is faster (", ownTurns, "<", pathTurns,

" without it");

aiUnit.dropTransport();

aiCarrier = null;

path = ownPath;

useTransport = false;

}

if (waiting) {

// If waiting for the carrier, signal that this

// unit can be reexamined if the carrier is still

// moving.

lbAt(lb);

lb.add(", wait for ", aiCarrier.getUnit());

return (aiCarrier.getUnit().getMovesLeft() > 0)

? MoveType.*MOVE\_NO\_ACCESS\_EMBARK*

: MoveType.*MOVE\_NO\_MOVES*;

}

}

if (useTransport && aiCarrier == null) {

// Still interested in transport but no carrier.

lb.add(", needs transport to ", target);

return MoveType.*MOVE\_NO\_ACCESS\_EMBARK*;

}

}

// Follow the path to the target. If there is one.

if (path == null) {

lbAt(lb);

lb.add(", no path to ", target);

return MoveType.*MOVE\_NO\_TILE*;

}

if (path.next == null) {

// This should not happen, the isAtLocation() test above

// should have succeeded.

throw new IllegalStateException("Trivial path found "

+ path.fullPathToString()

+ " from " + unit.getLocation() + " to target " + target

+ " result=" + unit.isAtLocation(target));

}

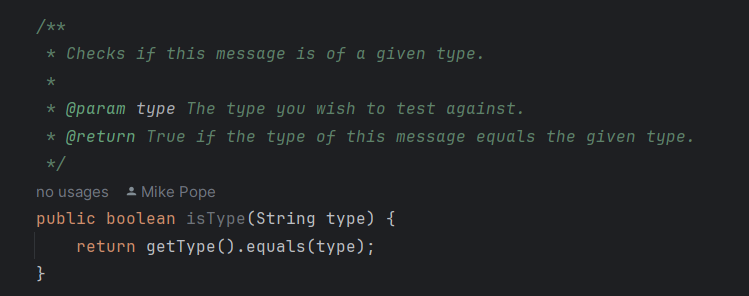
return followMapPath(path.next, lb);

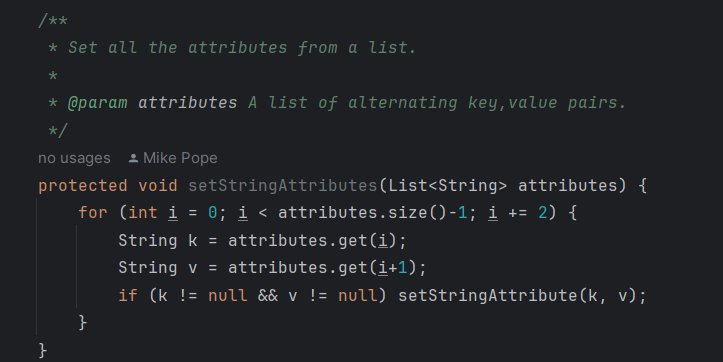
}

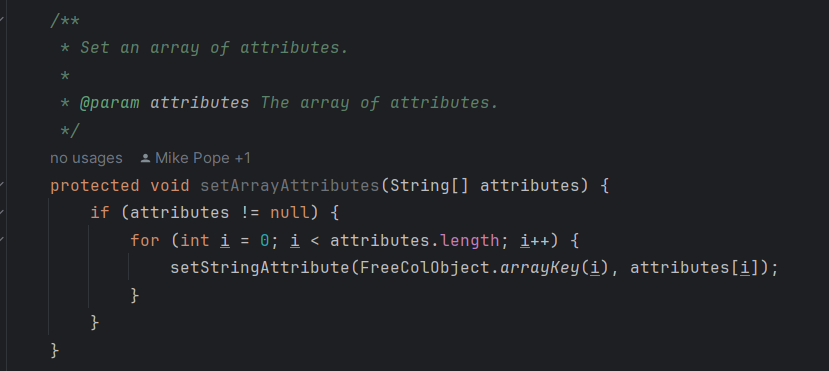
As mentioned above, this method is located in net.sf.freecol.server.ai.mission.Mission.travelToTarget(Location, CostDecider, LogBuilder).

This method is a bit too extensive and more complex than it needs to be. The various if and else statements make it difficult to understand with values changing throughout the method and being carried until the end (with almost 200 lines can be confusing to keep track), or even full functionalities, inside the if statement, that deserved their own method.

As said, a possible solution for this smell would be separating into their own methods the if statements functionalities and conditions, leaving a clean space and reorganized if and else statements with each method and condition being a lot more clear and easy to read.  
  
**Speculative Generality**

(3 methods from the class, scr.net.sf.freecol.common.networking.Message)





This code smell usually happens when programmers are preparing themselves for possible future functionalities or needs and preemptively code methods before they have any use. In this case, these 3 ended up not having any influence in the final project, therefore the possible and logical solution for this smell is simply removing the methods.

**Duplicate Code**

(3 more methods from the class, scr.net.sf.freecol.common.networking.Message)

*/\*\**

*\* Sets an attribute in this message with a boolean value.*

*\**

*\* @param key The attribute to set.*

*\* @param value The value of the attribute.*

*\*/*

protected void setBooleanAttribute(String key, Boolean value) {

if (value != null) setStringAttribute(key, Boolean.*toString*(value));

}

*/\*\**

*\* Sets an attribute in this message with an enum value.*

*\**

*\* @param key The attribute to set.*

*\* @param value The value of the attribute.*

*\*/*

protected void setEnumAttribute(String key, Enum<?> value) {

if (value != null) setStringAttribute(key, *downCase*(value.toString()));

}

*/\*\**

*\* Sets an attribute in this message with an integer value.*

*\**

*\* @param key The attribute to set.*

*\* @param value The value of the attribute.*

*\*/*

protected void setIntegerAttribute(String key, int value) {

setStringAttribute(key, Integer.*toString*(value));

}

These methods follow a similar pattern for different types of attributes (Boolean, Enum, Integer), but the core functionality of converting these values to strings and setting attributes is duplicated with minor variations. This qualifies as Duplicate code.

A possible solution for this particular case would be a single method for example “setAttribute”, that can handle various data types, using an Object argument as the value, and executes the setStringAttribute accordingly.

André Branco nº62482