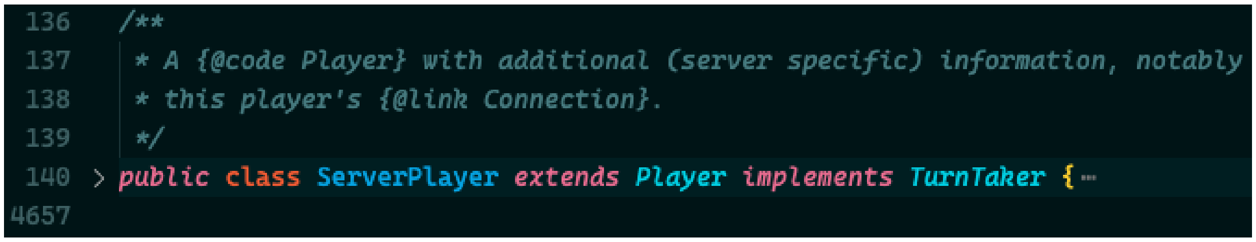
Code Smells

Rafael Tavares 60608

1 – Large Class

Snippet



```
/**
 * A {@code Player} with additional (server specific) information, notably
 * this player's {@link Connection}.
 */
public class ServerPlayer extends Player implements TurnTaker {⋯
```

Location

`src/net/sf/freecol/server/model/ServerPlayer.java`

Rationale

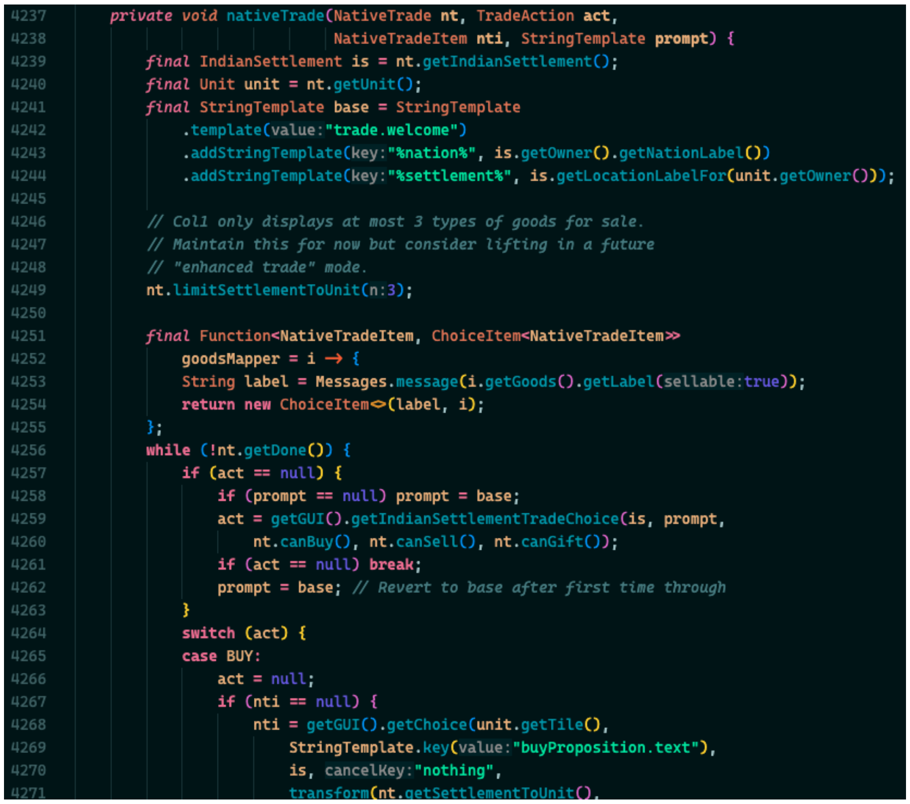
We can see this is a "Large Class" code smell because the class is excessively large, and it seems like it's trying to do too many things at once, this can lead to low cohesion, making the code harder to understand. It also makes testing challenging since it's difficult to isolate specific behaviors. Furthermore, such large classes are often less reusable, limiting their versatility.

Solution

To fix this, we can break down this class into smaller, more focused ones, each with a single clear responsibility, improving code organization and maintainability.

2 – Long Method

Snippet



```java
    private void nativeTrade(NativeTrade nt, TradeAction act,
                             NativeTradeItem nti, StringTemplate prompt) {
        final IndianSettlement is = nt.getIndianSettlement();
        final Unit unit = nt.getUnit();
        final StringTemplate base = StringTemplate
            .template(value:"trade.welcome")
            .addStringTemplate(key:"%nation%", is.getOwner().getNationLabel())
            .addStringTemplate(key:"%settlement%", is.getLocationLabelFor(unit.getOwner()));

        // Col1 only displays at most 3 types of goods for sale.
        // Maintain this for now but consider lifting in a future
        // "enhanced trade" mode.
        nt.limitSettlementToUnit(n:3);

        final Function<NativeTradeItem, ChoiceItem<NativeTradeItem>>
            goodsMapper = i -> {
            String label = Messages.message(i.getGoods().getLabel(sellable:true));
            return new ChoiceItem<>(label, i);
        };
        while (!nt.getDone()) {
            if (act == null) {
                if (prompt == null) prompt = base;
                act = getGUI().getIndianSettlementTradeChoice(is, prompt,
                    nt.canBuy(), nt.canSell(), nt.canGift());
                if (act == null) break;
                prompt = base; // Revert to base after first time through
            }
            switch (act) {
            case BUY:
                act = null;
                if (nti == null) {
                    nti = getGUI().getChoice(unit.getTile(),
                        StringTemplate.key(value:"buyProposition.text"),
                        is, cancelKey:"nothing",
                        transform(nt.getSettlementToUnit(),
```

It goes beyond, just did not fit.

Location

`src/net/sf/freecol/client/control/InGameController.java    Ln. 4229`

Rationale

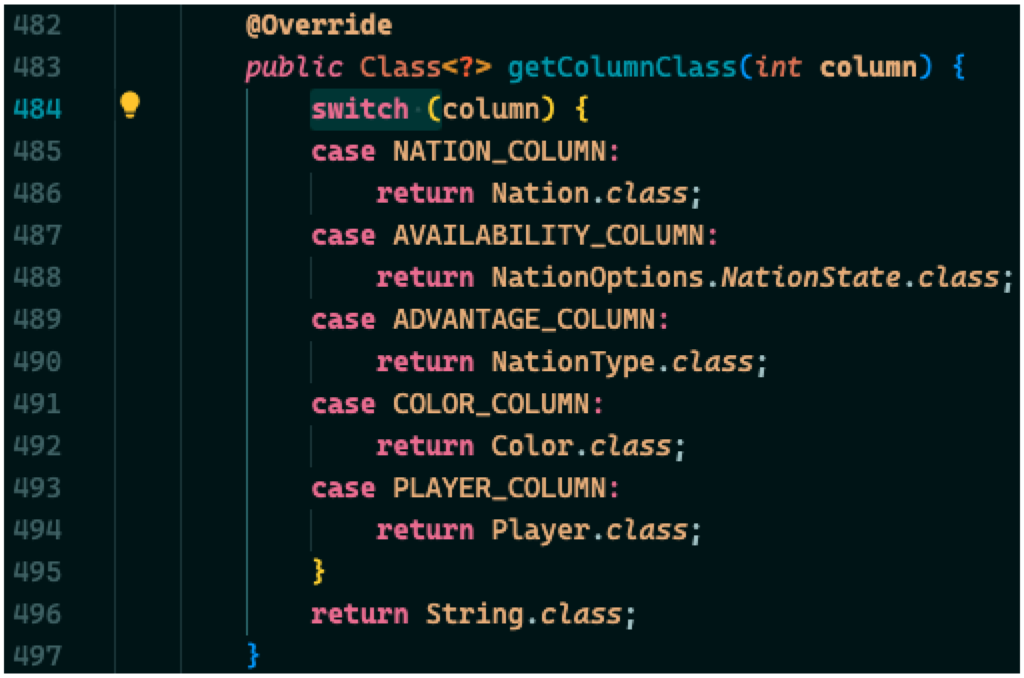
Here a "Long Method" code smell can be clearly identified as this method can't even fit on screen. After analyzing the method we can conclude it's a “Long Method” code smell which is composed by a method that has grown excessively long and complex. This can make the code harder to understand, maintain, and reuse.

Solution

To improve the situation, we can refactor the long method into smaller, more focused sub-methods, each with a specific role.

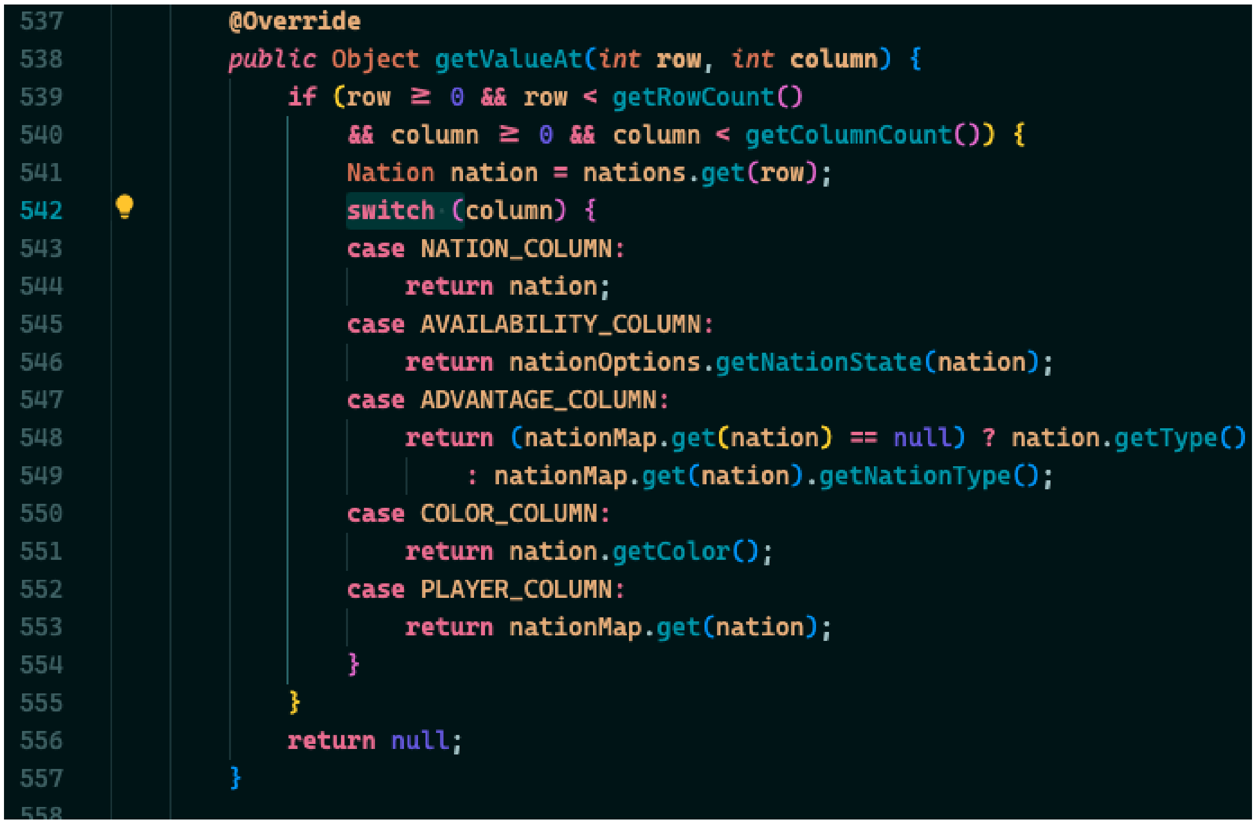
3 – Switch Statements

Snippet



```java
@Override
public Class<?> getColumnClass(int column) {
    switch (column) {
    case NATION_COLUMN:
        return Nation.class;
    case AVAILABILITY_COLUMN:
        return NationOptions.NationState.class;
    case ADVANTAGE_COLUMN:
        return NationType.class;
    case COLOR_COLUMN:
        return Color.class;
    case PLAYER_COLUMN:
        return Player.class;
    }
    return String.class;
}
```

and



```java
@Override
public Object getValueAt(int row, int column) {
    if (row >= 0 && row < getRowCount()
        && column >= 0 && column < getColumnCount()) {
        Nation nation = nations.get(row);
        switch (column) {
        case NATION_COLUMN:
            return nation;
        case AVAILABILITY_COLUMN:
            return nationOptions.getNationState(nation);
        case ADVANTAGE_COLUMN:
            return (nationMap.get(nation) == null) ? nation.getType()
                : nationMap.get(nation).getNationType();
        case COLOR_COLUMN:
            return nation.getColor();
        case PLAYER_COLUMN:
            return nationMap.get(nation);
        }
    }
    return null;
}
```

Location

`src/net/sf/freecol/client/gui/panel/PlayersTable.java   Ln. 482 and Ln. 537`

Rationale

In this class where both snippets come from we can notice multiple similar switch statements, indicating a "Switch Statement" code smell. This can lead to inflexible code and makes future modifications challenging and potentially causing code duplication.

Solution

To address this, we can employ design patterns like the Strategy Pattern. Instead of switches, we create separate classes or functions for each branch, improving extensibility and maintainability while adhering to object-oriented principles.