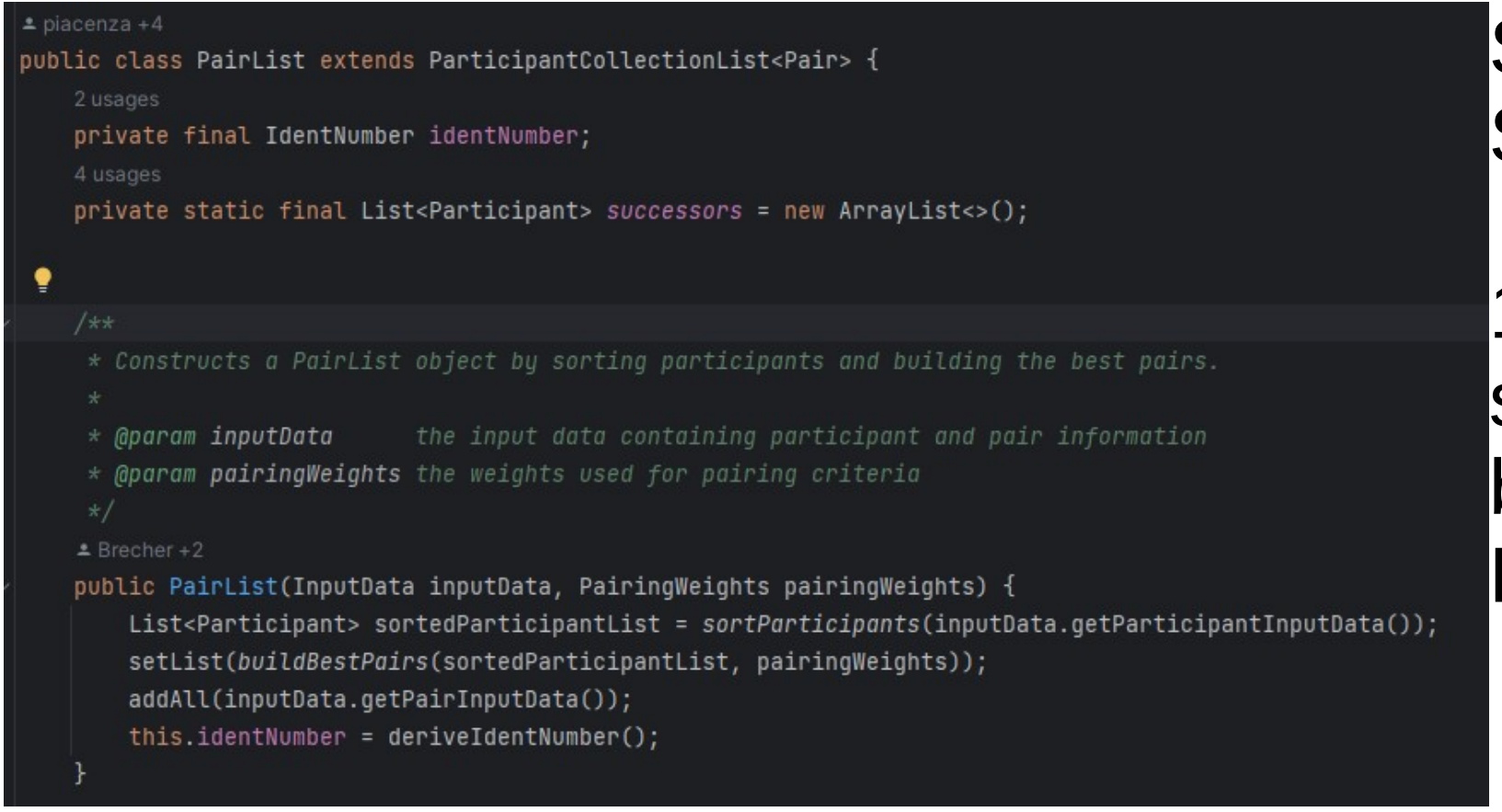
PairList



```
piacenza +4
public class PairList extends ParticipantCollectionList<Pair> {
    2 usages
    private final IdentNumber identNumber;
    4 usages
    private static final List<Participant> successors = new ArrayList<>();

    /**
     * Constructs a PairList object by sorting participants and building the best pairs.
     *
     * @param inputData      the input data containing participant and pair information
     * @param pairingWeights the weights used for pairing criteria
     */
    Brecher +2
    public PairList(InputData inputData, PairingWeights pairingWeights) {
        List<Participant> sortedParticipantList = sortParticipants(inputData.getParticipantInputData());
        setList(buildBestPairs(sortedParticipantList, pairingWeights));
        addAll(inputData.getPairInputData());
        this.identNumber = deriveIdentNumber();
    }
```

Score-System:

1. Person sucht sich besten Partner

Sortierung

```
private static List<Participant> sortParticipants(List<Participant> participantList) {
    List<Participant> sortedNoKitchenList = new ArrayList<>();
    List<Participant> sortedMaybeKitchenList = new ArrayList<>();
    List<Participant> sortedYesKitchenList = new ArrayList<>();

    for (Participant participant : participantList) {
        switch (participant.isHasKitchen()) {
            case NO:
                sortedNoKitchenList.add(participant);
                break;
            case MAYBE:
                sortedMaybeKitchenList.add(participant);
                break;
            case YES:
                sortedYesKitchenList.add(participant);
                break;
        }
    }

    List<Participant> sortedParticipantList = new ArrayList<>();
    sortedParticipantList.addAll(sortByFoodType(sortedNoKitchenList));
    sortedParticipantList.addAll(sortByFoodType(sortedMaybeKitchenList));
    sortedParticipantList.addAll(sortByFoodType(sortedYesKitchenList));

    return sortedParticipantList;
}
```

- Sortierung nach Küche

  -garantiert, dass sich alle “No Kitchen” Teilnehmer einen Partner suchen

Untersortierung

```
3 usages  Krenzer +2
private static List<Participant> sortByFoodType(List<Participant> participantList) {
    List<Participant> sortedParticipantList = new ArrayList<>();

    for (Participant participant : participantList) {
        if (participant.getFoodType() == FoodType.VEGAN) {
            sortedParticipantList.add(participant);
        }
    }
    for (Participant participant : participantList) {
        if (participant.getFoodType() == FoodType.VEGGIE) {
            sortedParticipantList.add(participant);
        }
    }
    for (Participant participant : participantList) {
        if (participant.getFoodType() == FoodType.MEAT) {
            sortedParticipantList.add(participant);
        }
    }

    for (Participant participant : participantList) {
        if (participant.getFoodType() == FoodType.NONE) {
            sortedParticipantList.add(participant);
        }
    }

    return sortedParticipantList;
}
```

- Sortierung nach FoodType

  -NONE zum Schluss, damit Filler-Foodtype garantiert über bleibt

Pairingweights

```
2 usages  2 inheritors  piacenza
public abstract class Weights {
    3 usages
    protected double ageDifferenceWeight;
    3 usages
    protected double genderDifferenceWeight;
    3 usages
    protected double foodPreferenceWeight;
```

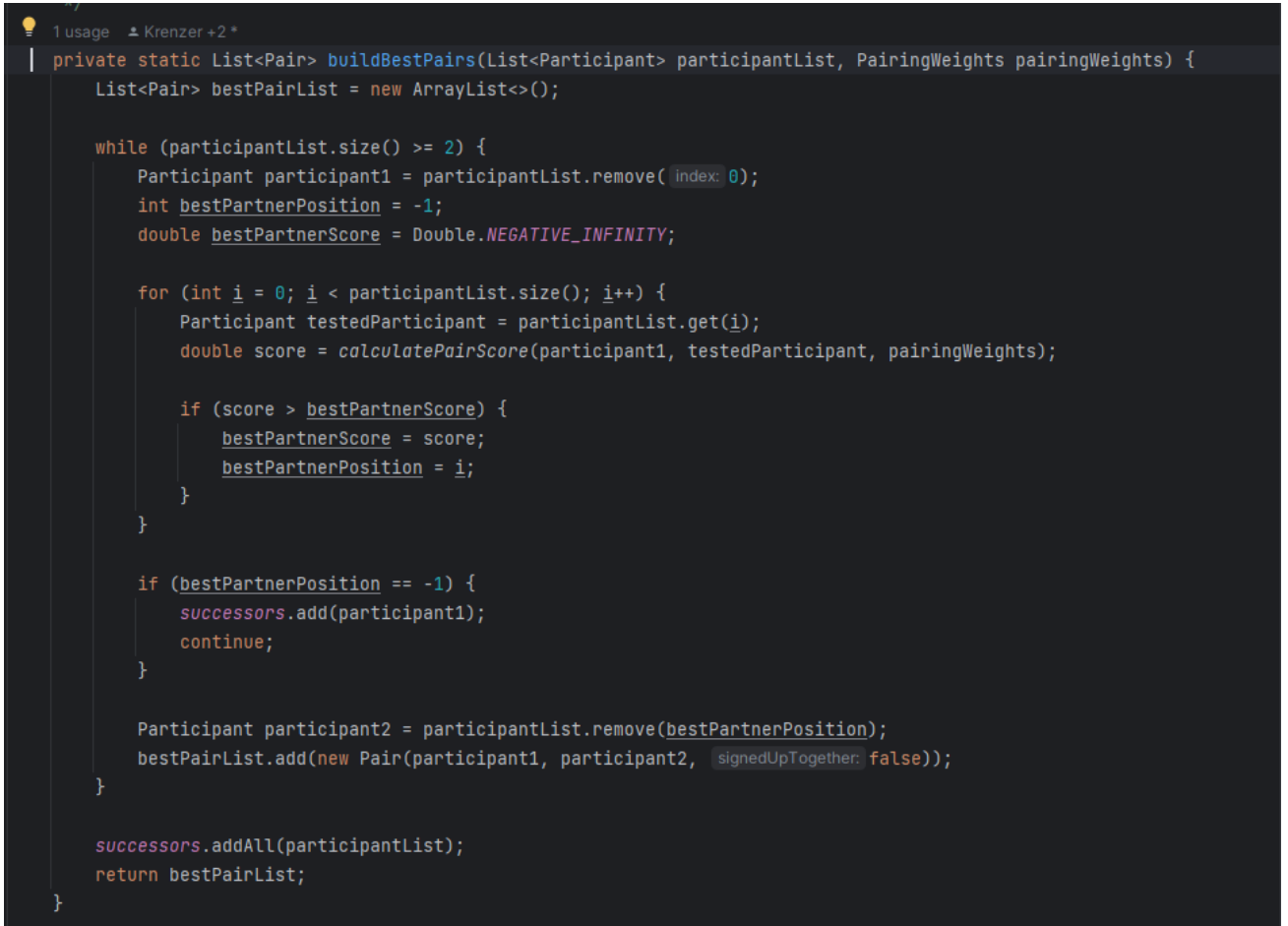
- Zuweisung von doubles als Gewichtung

```
31 usages  Krenzer +1
public class PairingWeights extends Weights {

    11 usages  Krenzer +1
    public PairingWeights(double pairAgeDifferenceWeight, double pairGenderDifferenceWeight, double pairFoodPreferenceWeight) {
        super(pairAgeDifferenceWeight, pairGenderDifferenceWeight, pairFoodPreferenceWeight);
    }

}
```

Paarbildung



```
1 usage   Krenzer +2 *
private static List<Pair> buildBestPairs(List<Participant> participantList, PairingWeights pairingWeights) {
    List<Pair> bestPairList = new ArrayList<>();

    while (participantList.size() >= 2) {
        Participant participant1 = participantList.remove( index: 0);
        int bestPartnerPosition = -1;
        double bestPartnerScore = Double.NEGATIVE_INFINITY;

        for (int i = 0; i < participantList.size(); i++) {
            Participant testedParticipant = participantList.get(i);
            double score = calculatePairScore(participant1, testedParticipant, pairingWeights);

            if (score > bestPartnerScore) {
                bestPartnerScore = score;
                bestPartnerPosition = i;
            }
        }

        if (bestPartnerPosition == -1) {
            successors.add(participant1);
            continue;
        }

        Participant participant2 = participantList.remove(bestPartnerPosition);
        bestPairList.add(new Pair(participant1, participant2,  signedUpTogether: false));
    }

    successors.addAll(participantList);
    return bestPairList;
}
```

- 1. Teilnehmer sucht sich seinen besten Partner
- Iterieren über alle anderen Teilnehmer
- Score muss geschlagen werden um bester Partner zu sein
- Paare werden progressiv schlechter, weil Pool kleiner wird

Score-Methode

- Aktuell bestimmt nur kitchen über illegale Paare
- Illegale Küchen direkt return -unendlich
- Beurteilung nach allen Score Methoden für Gesamtscore

```
1 usage  Dhtrx +2
private static double calculatePairScore(Participant participant1, Participant testedParticipant, PairingWeights pairingWeights) {
    double score = 0;
    double kitchenScore = compareKitchen(participant1, testedParticipant);
    if (kitchenScore == Double.NEGATIVE_INFINITY) {
        return Double.NEGATIVE_INFINITY;
    }
    score += kitchenScore;
    score += compareGender(participant1, testedParticipant, pairingWeights);
    score += compareFoodPreference(participant1, testedParticipant, pairingWeights);
    score += compareAge(participant1, testedParticipant, pairingWeights);
    return score;
}
```

Küchentest

- Gleiche Küche und Paarung No zu No illegal
- No zu maybe und maybe zu maybe -50, also erst möglich wenn es keine yes-Küchen mehr gibt

```
1 usage   piacenza +3
private static double compareKitchen(Participant participant1, Participant testedParticipant) {
    switch (participant1.isHasKitchen()) {
        case YES:
            return (testedParticipant.isHasKitchen() == KitchenAvailability.YES &&
                    participant1.getKitchen().equals(testedParticipant.getKitchen())) ? Double.NEGATIVE_INFINITY : 0;
        case NO:
            if (testedParticipant.isHasKitchen() == KitchenAvailability.YES) return 0;
            return (testedParticipant.isHasKitchen() == KitchenAvailability.MAYBE) ? -50 : Double.NEGATIVE_INFINITY;
        case MAYBE:
            return (testedParticipant.isHasKitchen() == KitchenAvailability.YES) ? 0 : -50;
        default:
            return 0;
    }
}
```

Age und Gender score

- Verschiedene Geschlechter geben 0.5 mal genderWeight

```
1 usage   Brecher +1
private static double compareGender(Participant participant1, Participant testedParticipant, PairingWeights pairingWeights) {
    return participant1.getGender().equals(testedParticipant.getGender()) ? 0 : 0.5 * pairingWeights.getGenderDifferenceWeight();
}
```

- 1 mal ageWeight als Standard, jedes Altersstufe zieht 0.1 mal ageWeight ab

```
1 usage   Brecher +2
private static double compareAge(Participant participant1, Participant testedParticipant, PairingWeights pairingWeights) {
    double ageDifference = participant1.getAge().getAgeDifference(testedParticipant.getAge());
    return pairingWeights.getAgeDifferenceWeight() * (1 - 0.1 * ageDifference);
}
```

- Effektivität der weights so, dass bei Veränderung der 1,1,1 weight ähnliche Veränderung bewirkt

FoodPreference score

- Weight allgemein 0.5 mal FoodPreferenceWeight

  meat/none → meat/none = weight, meat → veggie/vegan = -1000weight

  vegan/veggie → none = 0.25/0.33 weight, vegan → veggie = 0.5 weight

```
1 usage   piacenza +2
private static double compareFoodPreference(Participant participant1, Participant testedParticipant, PairingWeights pairingWeights) {
    double weight = 0.5 * pairingWeights.getFoodPreferenceWeight();
    switch (participant1.getFoodType()) {
        case MEAT:
            if (testedParticipant.getFoodType() == FoodType.MEAT) return weight;
            return (testedParticipant.getFoodType() == FoodType.NONE) ? weight : - 1000 * weight;
        case VEGGIE:
            if (testedParticipant.getFoodType() == FoodType.VEGGIE) return weight;
            if (testedParticipant.getFoodType() == FoodType.VEGAN) return 0.5 * weight;
            return (testedParticipant.getFoodType() == FoodType.NONE) ? 0.33 * weight : -1000 * weight;
        case VEGAN:
            if (testedParticipant.getFoodType() == FoodType.VEGAN) return weight;
            if (testedParticipant.getFoodType() == FoodType.VEGGIE) return 0.5 * weight;
            return (testedParticipant.getFoodType() == FoodType.NONE) ? 0.25 * weight : -1000 * weight;
        case NONE:
            return (testedParticipant.getFoodType() == FoodType.NONE || testedParticipant.getFoodType() == FoodType.MEAT) ? weight : 0.25 * weight;
        default:
            return 0;
    }
}
```

Ende Paarzuweisung

```
1 usage   Krenzer +2 *
private static List<Pair> buildBestPairs(List<Participant> participantList, PairingWeights pairingWeights) {
    List<Pair> bestPairList = new ArrayList<>();

    while (participantList.size() >= 2) {
        Participant participant1 = participantList.remove( index: 0);
        int bestPartnerPosition = -1;
        double bestPartnerScore = Double.NEGATIVE_INFINITY;

        for (int i = 0; i < participantList.size(); i++) {
            Participant testedParticipant = participantList.get(i);
            double score = calculatePairScore(participant1, testedParticipant, pairingWeights);

            if (score > bestPartnerScore) {
                bestPartnerScore = score;
                bestPartnerPosition = i;
            }
        }

        if (bestPartnerPosition == -1) {
            successors.add(participant1);
            continue;
        }

        Participant participant2 = participantList.remove(bestPartnerPosition);
        bestPairList.add(new Pair(participant1, participant2, signedUpTogether: false));
    }

    successors.addAll(participantList);
    return bestPairList;
}
```

- 1. Teilnehmer wird mit bestem Partner zu Paar
- Falls kein legaler Partner gefunden wurde, wird Person zu Successors hinzugefügt
- Übrige Personen werden auch zu Succerssors hinzugefügt

Ende PairList

```java
piacenza +4
public class PairList extends ParticipantCollectionList<Pair> {
    2 usages
    private final IdentNumber identNumber;
    4 usages
    private static final List<Participant> successors = new ArrayList<>();


    /**
     * Constructs a PairList object by sorting participants and building the best pairs.
     *
     * @param inputData      the input data containing participant and pair information
     * @param pairingWeights the weights used for pairing criteria
     */
    Brecher +2
    public PairList(InputData inputData, PairingWeights pairingWeights) {
        List<Participant> sortedParticipantList = sortParticipants(inputData.getParticipantInputData());
        setList(buildBestPairs(sortedParticipantList, pairingWeights));
        addAll(inputData.getPairInputData());
        this.identNumber = deriveIdentNumber();
    }
```

- Nach der Paarzuteilung fügen wir die Input-Paare dazu
- Ident-numbers werden generiert