Rozpočítávání peněz pomocí mobilní aplikace

Semestrální projekt pro A4M35KO

David Vávra
vavradav@fel.cvut.cz

13. května 2011

1 Zadání

Parta kamarádů jede na výlet. Náklady hradí společně, ale každý platí za něco jiného. Např. někdo platí za benzín, jiný za jídlo v restauraci, jiný za vstupenky atd. Po příjezdu domů se chtějí finančně vyrovnat. Mohou si posílat peníze převodem, ale chtějí to udělat tak, aby převodů peněz bylo co nejméně. Cílem práce je tedy vytvořit program, jehož výstupem bude kdo má komu kolik poslat. Sám bych takovou aplikaci využil, zvlášť pokud by byla ve formě mobilní aplikace a pomáhala by mi už v průběhu výletu.

1.1 Kategorizace

Na vstupu bude zadáno, kolik kdo platil a za koho. Na výstupu budou transakce, jimiž dojde k vyrovnání dluhů. Minimalizační kritérium bude počet transakcí, sekundární kritérium bude minimální objem peněz, který se musí poslat. Problém by šel řešit pomocí ILP. Dá se ale vymyslet lineární algoritmus, který problém vyřeší vždy s n-1 transakcemi, kde n je počet účastníků - více v podsekci 3.1. Mým cílem je vymyslet algoritmus s menším počtem transakcí než n-1 (pokud to jde). Program má být reálná mobilní aplikace, z toho vyplývají další omezení popsané v podsekci 1.2.

1.2 Další omezení a specifikace

Cílem není jenom samotný algoritmus, ale mobilní aplikace, která se dá reálně použít. Vyplývají z toho další omezení:

- bude se jednat o aplikaci pro systém Android
- bude možné vytvářet skupiny s různými účastníky, mezi kterými se dluhy vedou
- bude možné přidat výdaj - cenu, kdo to platil, za koho to platil (defaultně za všechny) a popis výdaje
- aplikace bude napovídat, kdo má platit další výdaj, aby se zjednodušilo následné vyrovnání
- bude možné zobrazit seznam výdajů a to nejdůležitější - kdo má komu kolik poslat, aby se vyrovnali
- bude možné nastavit toleranci výpočtů - např. rozdíl 10 Kč se neřeší

2 Související práce

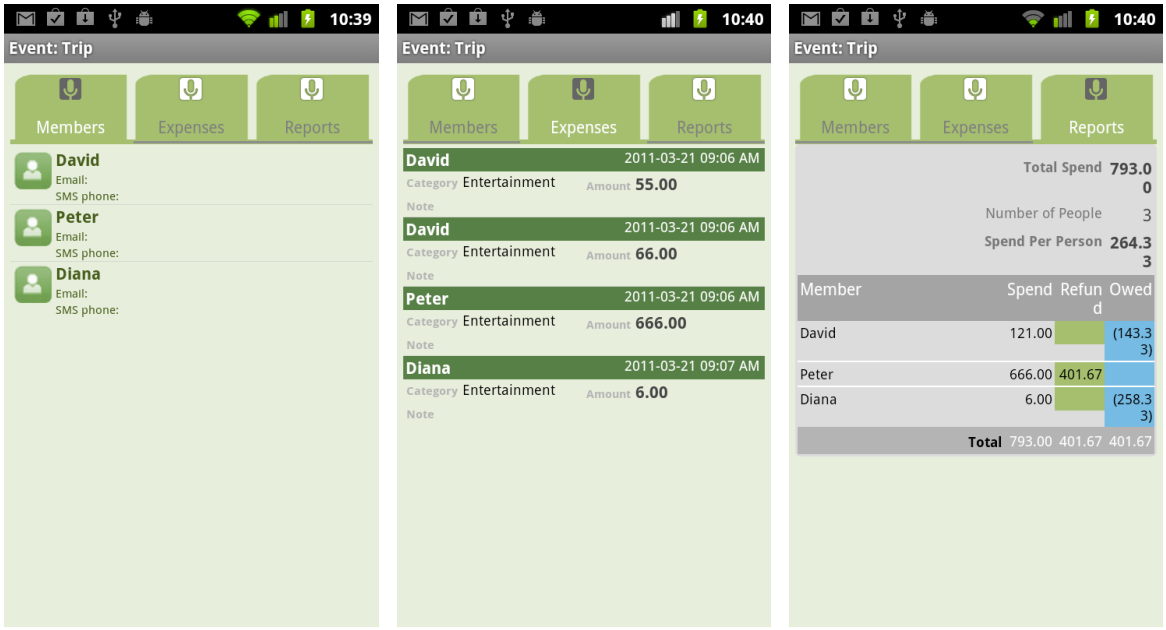
Procházel jsem Android Market[1] a hledal podobné aplikace. Existuje spousta aplikací typu "Tip Calculator"nebo "Bill Splitter". Jsou to docela primitivní aplikace, které umožňují zadat kolik celkově stála útrata v restauraci a procentuální bonus pro číšníka. Aplikace potom spočítá, kolik má každý platit. Moje aplikace je ale složitější - má zaznamenávat výdaje za delší dobu. Našel jsem dvě aplikace, které mojí nejvíc aplikaci konkurují. Tady jsou:

2.1 Expense Share + Tip Calculator

Tato aplikace[2] má dvě části - Tips a Events. Nás zajímá část events, určená pro zaznamenávání výdajů v průběhu např. výletu. Obrazovky z této části jsou zobrazené na screenshotech na obrázku 1. Aplikace má docela zdařilý design, ale pouze zobrazí rozdíly mezi platbami účastníků - není jasné, kdo má komu kolik platit. Zajímavá možnost je poslat souhrn e-mailem nebo přes SMS.

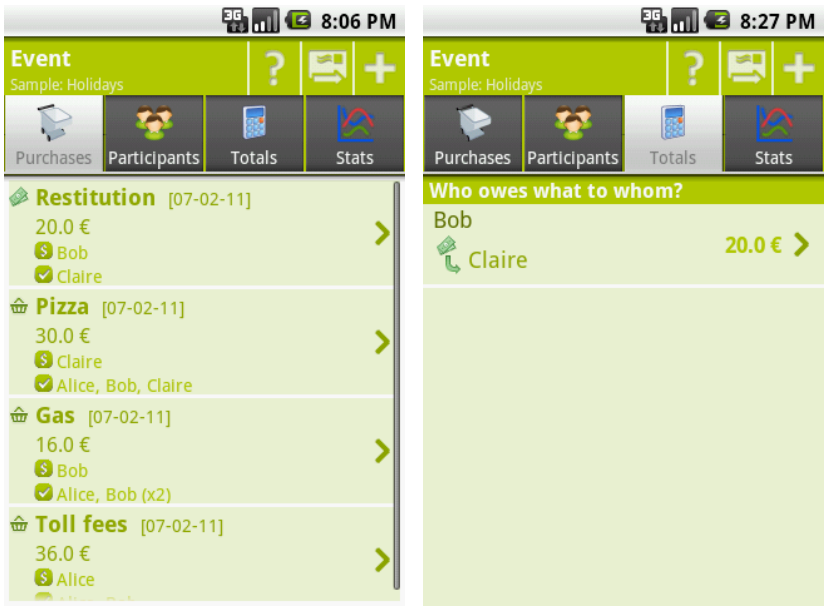
2.2 Kidoikoiaki

Tato aplikace[3] se nejvíc blíží té mojí. Nejdůležitější obrazovky jsou zobrazeny na obrázku 2. Aplikace jako jedna z mála obsahuje obrazovku kdo má komu kolik poslat. Nemá ale funkci napovídání, kdo má platit příště. Aplikace má pěkný design a je docela přehledná. Věřím ale, že to jde udělat ještě lépe a intuitivněji. Zaujala mě možnost zadat, že jeden účastník vlastně znamená víc lidí (např. pár nebo rodina) a v rozpočítávání tento účastník potom platí víc.



(a) Správa účastníků (b) Správa výdajů (c) Souhrn vyrovnání

Obrázek 1: Screenshoty z aplikace Expense Share + Tip Calculator



(a) Přehled nákupů (b) Kdo má komu kolik poslat

Obrázek 2: Screenshoty z aplikace Kidoikoiaki

3 Popis algoritmu

Existuje lineární algoritmus, který problém vyřeší, ale není optimální:

3.1 Algoritmus s n-1 transakcemi

Platby převedu na bilanci - rozdíl toho, kolik celkově účastník platil a kolik měl platit. Účastníky seřadím vzestupně podle bilance. První má zápornou bilanci - pošle druhému tolik peněz, kolik je jeho bilance. Pokud má druhý zápornou bilanci, pošle třetímu svojí bilanci + to, co obdržel. Pokud má kladnou bilanci, pošle dál jenom přebytek. Třetí pošle čtvrtému atd. Všichni pošlou peníze kromě posledního, který peníze jenom obdrží. Transakcí je tedy n-1.

3.2 Vylepšený algoritmus

Při testování jsem ale narazil na jiný problém - algoritmus sice dává nízký počet transakcí, ale v transakcích je zbytečně mnoho peněz. Může nastat situace, kdy někdo dluží jenom 1 Kč, ale protože je v "řetězu transakcí", bude muset tomu dalšímu poslat velkou částku. V reálném životě by s tím byly problémy - bankovní poplatky nebo nutnost čekat na příchod peněz od "toho přede mnou".

Procházel jsem diskuze na StackOverflow, abych zjistil, jak problém řeší druzí. Pročetl jsem otázky [4], [5] a [6]. Zaujala mě myšlenka v transakcích párovat vždy toho s nejmenší bilancí s tím s největší bilancí. Proběhne mezi nimi transakce a buď jeden nebo oba budou mít bilanci na nule - nemusí se s nimi dál počítat. Algoritmus tedy vždy problém vyřeší s n-1 transakcemi, někdy i lépe. Autor nápadu tvrdil, že je toto řešení optimální. Přemýšlel jsem nad tím a nalezl tento protipříklad:

Jméno	Bilance
Alice	+8 Kč
Bob	-4 Kč
Cyril	-5 Kč
David	+7 Kč
Eva	-6 Kč
František	+3 Kč
Gustav	-3 Kč

Algoritmus by transakce vyřešil následovně:

Kdo má dát	Kolik	Komu
Eva	6 Kč	Alice
Cyril	5 Kč	David
Bob	3 Kč	František
Gustav	2 Kč	David
Bob	1 Kč	Alice
Gustav	1 Kč	Alice

To je celkem 6 transakcí, což odpovídá n-1. Problém lze ale vyřešit na méně transakcí:

Kdo má dát	Kolik	Komu
Gustav	3 Kč	František
Eva	6 Kč	Alice
Cyril	5 Kč	David
Bob	2 Kč	Alice
Bob	2 Kč	David

Použili jsme 5 transakcí, algoritmus tedy není optimální, ale je lepší než ten v podsekci 3.1, protože se neposílá zbytečně více peněz, než je potřeba.

3.3 Optimální algoritmus

V některých případech lze problém vyřešit na méně než n-1 transakcí. Jsou to ty případy, kdy jsou mezi účastníky podskupiny, kde se členové se vyrovnají mezi sebou. Každou z těchto podskupin lze vyřešit s n-1 transakcemi (s použitím algoritmu v podsekci 3.2). Pokud nalezneme podskupinu, ušetříme jednu transakci.

Jak nalézt tyto podskupiny? Pomocí vygenerování všech dvoj-kombinací, troj-kombinací, ... (n-1)kombinací a ověření, zda součet bilancí v každé kombinaci není 0. To je optimální řešení, ale výrazně tím vzrůstá časová složitost - počet kombinací roste faktoriálně. Naštěstí ale v praxi bude počet účastníků malý, takže úloha půjde v rozumném čase vyřešit.

3.4 Funkce Kdo má platit a tolerance

V podsekci 1.2 je definovaná funkce Kdo má platit. Má v průběhu výletu napovídat, kdo má zrovna platit, aby bylo konečné vyrovnání co nejjednodušší. Při použití této funkce ještě není jasná částka platby. Proto platit má vždy ten s nejmenší bilancí.

Tolerance se dá aplikovat do stávajících algoritmů. Místo porovnávání součtu bilancí nule se bude porovnávat $\leq$tolerance. Z výsledných transakcí se také odeberou transakce s částkou $\leq$tolerance.

4 Implementace

Implementace nebyla pouze o algoritmu rozpočítávání peněz, rozhodl jsem se aplikaci dotáhnout do konce a zveřejnit jí na Android Marketu.

4.1 Rozpočítávání peněz

Pro rozpočítávání peněz jsem použil algoritmus popsaný v podsekci 3.3. Pro výpočet jsem potřeboval generátor všech kombinací. Použil jsem volně dostupnou implemen-

taci v Javě[7]. Používá Knuthův algoritmus pro generování kombinací[8]. Zdrojový kód algoritmu v Javě je k dispozici v příloze A.

4.2 Náročnost algoritmu a experimenty

Počet kombinací se spočítá podle vzorce:

$$\frac{n!}{k!(n-k)!}$$

Protože musím kombinace generovat, časová a paměťová náročnost algoritmu je v $O(n!)$.

Testoval jsem výpočet na přístroji Nexus One (procesor Snapdragon 1 Ghz). Zde jsou výsledky experimentů:

Počet členů	Doba běhu
5	3 ms
10	82 ms
15	953 ms
18	8 s
20	55 s

Z výsledků je vidět, že pokud bude členů ve skupině méně než 18, nebude s dobou výpočtu problém. V praxi bude počet členů obvykle menší, takže faktoriální složitost nevadí.

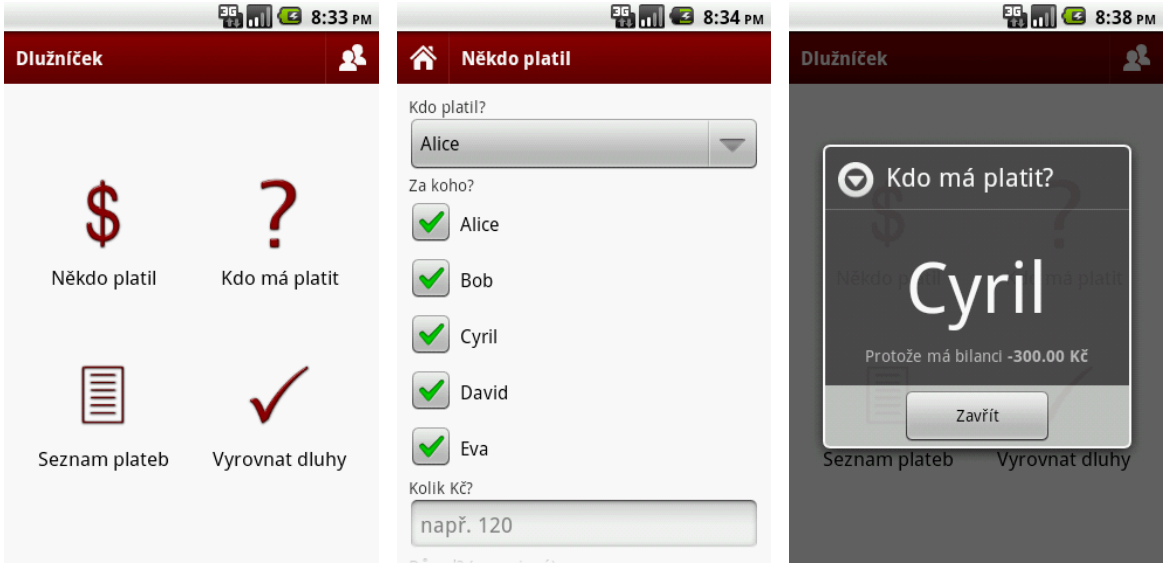
4.3 Android aplikace

Z povahy mobilní aplikace vyplynulo mnoho dalších funkcí, které jsem do programu postupně implementoval:

- Snažil jsem se o maximální uživatelskou přívětivost a dodržování konvencí pro Android aplikace[9].
- Pro horní lištu jsem použil volně dostupnou implementaci návrhového vzoru ActionBar[10].
- Ikonky jsem získal z databáze volně dostupných ikonek[11] a ikonu aplikace vyrobil pomocí nástroje Android Asset Studio[12].
- Veškerá data aplikace jdou sdílet e-mailem.
- Integrace s kontakty v telefonu při zadávání členů skupiny (a pamatování e-mailových adres).
- Definice měny pro skupinu, možnost zadat libovolnou měnu.
- Lokalizace do angličtiny a češtiny.

- Možnost přidávání, přejmenování a mazání členů v průběhu.
- Možnost označení transakce jako splacené, možnost mazání plateb.
- Našeptávač pro důvody transakcí s pamatováním posledních důvodů.

Výsledná podoba aplikace je vidět na obrázku 3 a obrázku 4.



(a) Hlavní obrazovka (b) Někdo platil (c) Kdo má platit

Obrázek 3: Screenshoty z výsledné aplikace Dlužníček (Settle Up v angličtině)

5 Závěr

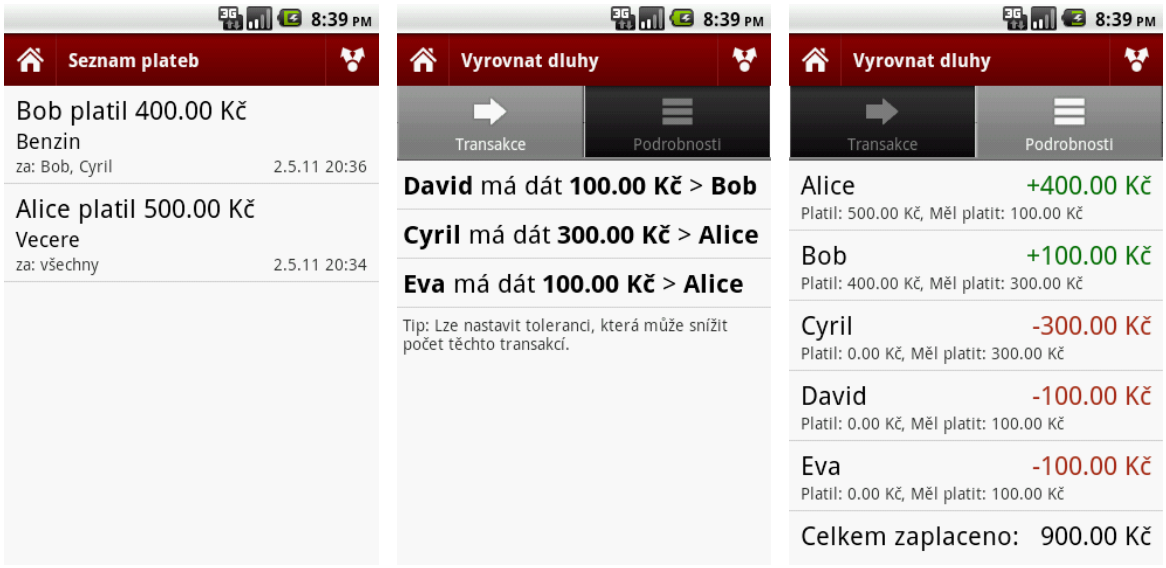
Aplikace je volně ke stažení zde:

https://market.android.com/details?id=cz.destil.settleup

K 30.4.2011 si aplikaci stáhlo 197 uživatelů - za 11 dní od publikování aplikace. Mám na aplikaci pozitivní reakce od známých, reagoval jsem na jejich připomínky a aplikaci průběžně vylepšoval. Díky podnětu z tohoto předmětu se mi podařilo vytvořit zajímavou aplikaci, která má šanci se prosadit na globálním trhu.

5.1 Práce do budoucna

Plánuji dále reagovat na podněty uživatelů a aplikaci vylepšovat. Největší nedostatek vidím v omezeném sdílení a zálohování dat (pouze pomocí e-mailu). Jde o citlivá finanční data, které je potřeba chránit. To vše hodlám vyřešit synchronizací přes webový



(a) Seznam plateb (b) Výsledné transakce (c) Detail pro vyrovnání dluhů

Obrázek 4: Screenshoty z výsledné aplikace Dlužníček (Settle Up v angličtině)

server. Lidé se budou moci připojovat i do skupin, které sami nevytvořili. Odpadne nutnost zapisovat si všechny dluhy sám. Data skupin se budou synchronizovat na pozadí, aplikace ale bude plně fungovat offline (všechny výpočty zůstanou na klientovi). Na serveru se bude uchovávat pouze historie změn ve skupině. Z historie změn se nikdy nebude nic mazat, proto se kdykoli půjde vrátit k předchozímu stavu. Tím bude zajištěno zálohování a ochrana dat.

Jednoduchý server plánuji napsat na Google AppEngine. Zajistí se tím škálovatelnost a ušetří náklady za hosting. Klient bude se serverem komunikovat pomocí RESTového API, které plánuji zveřejnit. Kdokoli potom bude moci napsat kompatibilní iPhone aplikaci, webovou verzi, desktopovou aplikaci, ... Poté získá aplikace výraznou konkurenční výhodu nad podobnými aplikacemi a je možné, že se stane opravdu úspěšnou.

Reference

[1] "Android Market - oficiální seznam Android aplikací." https://market.android.com.

[2] Greenbeansoft, "Expense Share + Tip Calculator." https://market.android.com/details?id=com.greenbeansoft.ExpensesShareLite.

[3] Appwall, "Kidoikoiaki." https://market.android.com/details?id=kidoikoiaki.main.

[4] "Algorithm to share/settle expenses among a group."
http://stackoverflow.com/questions/974922, červen 2009.

[5] "Who owes who money optimization problem."
http://stackoverflow.com/questions/4554655, prosinec 2010.

[6] "What algorithm to use to determine minimum number of actions required to get the system to "zero" state?."
http://stackoverflow.com/questions/877728, květen 2009.

[7] M. Gilleland, "Combination Generator."
http://www.merriampark.com/comb.htm.

[8] K. H. Rosen, *Discrete Mathematics and Its Applications*. NY: McGraw-Hill, 2nd ed., 1991.

[9] "User Interface Guidelines."
http://developer.android.com/guide/practices/ui_guidelines/index.html.

[10] J. Nilsson, "Action Bar for Android."
https://github.com/johannilsson/android-actionbar.

[11] "Royalty Free Icons & Clipart Stock Images."
http://icons.mysitemyway.com/.

[12] "Android Asset Studio."
http://android-ui-utils.googlecode.com/hg/asset-studio/dist/index.html.

A Zdrojový kód algoritmu pro rozpočítávání peněz

```
1  package cz.destil.settleup.utils;
2
3  import java.util.Collections;
4  import java.util.HashMap;
5  import java.util.Iterator;
6  import java.util.LinkedList;
7  import java.util.List;
8
9  import cz.destil.settleup.data.Member;
10
11 /**
12  * Debt calculator - calculates who should pay how much to who with
       optional
13  * tolerance value
14  * Semestral work for A4M35KO
```

```
15  *
16  * @author David Vavra
17  *
18  */
19 public class DebtCalculator {
20
21  /**
22   * Main algorithm, calculates who should send how much to who It
       optimizes
23   * basic algorithm
24   *
25   * @param members
26   *          List of members with their credit and debts
27   * @param tolerance
28   *          Money value nobody cares about
29   * @return List of Hashmaps encoding transactions
30   */
31  public static List<HashMap<String, Object>> calculate(List<Member>
       members,
32    double tolerance) {
33   tolerance += 0.00001; // increasing tolerance due to double
       precision issues
34   List<HashMap<String, Object>> results = new LinkedList<HashMap<
       String, Object>>();
35   // remove members where debts are too small (1-pairs)
36   Iterator<Member> iterator = members.iterator();
37   while (iterator.hasNext()) {
38    Member member = iterator.next();
39    if (Math.abs(member.getBalance()) <= tolerance)
40     iterator.remove();
41   }
42   // find n-pairs, starting at 2-pairs, deal with them using basic
43   // algorithm and remove them
44   int n = 2;
45   while (n < members.size() - 1) {
46    CombinationGenerator generator = new CombinationGenerator(members.
       size(), n);
47    boolean nPairFound = false;
48    while (generator.hasMore()) {
49     double sum = 0;
50     int[] combination = generator.getNext();
51     for (int i = 0; i < combination.length; i++) {
52      sum += members.get(combination[i]).getBalance();
53     }
54     if (Math.abs(sum) <= tolerance) {
55      // found n-pair - deal with them
56      List<Member> pairedMembers = new LinkedList<Member>();
57      for (int i = 0; i < combination.length; i++) {
58       pairedMembers.add(members.get(combination[i]));
59      }
60      List<HashMap<String, Object>> values = basicDebts(pairedMembers,
          tolerance);
```

```
61      results.addAll(values);
62      members.removeAll(pairedMembers);
63      nPairFound = true;
64     }
65     if (nPairFound)
66      break;
67    }
68    if (!nPairFound)
69     n++;
70   }
71   // deal with what is left after removing n-pairs
72   List<HashMap<String, Object>> values = basicDebts(members, tolerance
       );
73   results.addAll(values);
74   return results;
75  }
76
77  /**
78   * Non-optimal debts algorithm - it calculates debts with N-1
       transactions
79   *
80   * @param members
81   *          List of members with their credit and debts
82   * @param tolerance
83   *          Money value nobody cares about
84   * @return List of Hashmaps encoding transactions
85   */
86  private static List<HashMap<String, Object>> basicDebts(List<Member>
       members,
87    double tolerance) {
88   List<HashMap<String, Object>> debts = new LinkedList<HashMap<String,
       Object>>();
89   int resolvedMembers = 0;
90   while (resolvedMembers != members.size()) {
91    // transaction is from lowest balance to highest balance
92    Collections.sort(members);
93    Member sender = members.get(0);
94    Member recipient = members.get(members.size() - 1);
95    double senderShouldSend = Math.abs(sender.paid - sender.spent);
96    double recipientShouldReceive = Math.abs(recipient.paid - recipient
       .spent);
97    double amount;
98    if (senderShouldSend > recipientShouldReceive) {
99     amount = recipientShouldReceive;
100   } else {
101    amount = senderShouldSend;
102   }
103   sender.spent -= amount;
104   recipient.paid -= amount;
105   // create transaction
106   HashMap<String, Object> values = new HashMap<String, Object>();
107   values.put("from", sender);
```

```
108     values.put("amount", amount);
109     values.put("to", recipient);
110     debts.add(values);
111     // mark members who are settled
112     senderShouldSend = Math.abs(sender.paid - sender.spent);
113     recipientShouldReceive = Math.abs(recipient.paid - recipient.spent)
            ;
114     if (senderShouldSend <= tolerance)
115       resolvedMembers++;
116     if (recipientShouldReceive <= tolerance)
117       resolvedMembers++;
118
119   }
120   // limit transactions by tolerance
121   Iterator<HashMap<String, Object>> iterator = debts.iterator();
122   while (iterator.hasNext()) {
123     HashMap<String, Object> debt = iterator.next();
124     if ((Double) debt.get("amount") <= tolerance)
125       iterator.remove();
126   }
127   return debts;
128  }
129 }
```