Grayson Hart
Thomas Mintun
CS4500-Chakraborty-Project1

Requirements Specification Document:

1. Introduction

The people involved in the creation of this software are Thomas Mintun the user/customer, and Grayson Hart the software developer. Thomas is concerned his house will burn down from wildfires, and wants a mobile website for firefighters so they will know what items to clear from his house. Thomas wants to maximize the personal value and number of items that firefighters save. The firefighters in Thomas's area have stated they will make one sweep through houses with a knapsack of variable weight capacity in case wildfires are approaching, but do not know what items to grab from people's houses. This document's audience is: Thomas Mintun and his neighborhood, Grayson Hart the developer, and the firefighters of Clear Lake, California. The scope of this project is small in size. The software is ran on a website, and will only serve Thomas and Clear Lake firefighters.

2. Overall Description

This product will serve Thomas, and firefighters of Clear Lake. This is an emergency scenario in the middle of wildfire season, and the product must be produced in 3 days time. Updates to the product may include: log in features for firefighters, adding more users than Thomas and having log in's for those users, a chat feature, and a better user interface. The features of the product will be: real time software (the software does not take an extended amount of time from input of data to output of data), updatable list with developer help, ability to view list on web, variable weight of firefighter's knapsack, and a maintenance contract for 6 months with possible updates at further cost. User classes will be: Thomas and firefighter. Because the software needs to be functional immediately, there is no log in. Thomas and Grayson decided the maximum number of items possible to be saved is 50 items. The characteristics of the Thomas class will be: items name, weights of items, values of importance for items, and how many pieces the item is (all pieces are one for version one). The characteristics of the firefighter class will be: weight capacity of knapsack. The operating environment will be on the internet; the website is also optimized for mobile browsers. Implementation constraints will include: software must be functional in 3 days, Grayson Hart the developer must maintain a maintenance role for 6 months after the project. Assumptions that are being made: the Clear Lake fire department will adopt and use this because Thomas knows the fire chief, there will be no malicious actors for the first release, pre existing knowledge for both actors that the items saved by the firefighters will be available to Thomas the next day at the fire station, only Thomas and the firefighters have a link to the website and thus the only users will be Thomas and the firefighters for the first version, either the full item is grabbed or the item is not grabbed at all (no partial items for version one), the weights of the items are in pounds and integer values, the value scale of the items is integer values between one and ten inclusive, Grayson and Thomas have worked out what will be on his list of stuff to grab and Thomas knows this list cannot be changed without contacting Grayson and will take up to 48 hours to change.

The use cases for these actors is detailed here:

A. Thomas will put his information into the software with the help of developer Grayson.
    a. Success Scenario: The information is entered into software correctly. Grayson uploads software to server.
    b. Failure Scenario: The information is not entered in incorrectly. Grayson does not upload program to server.
B. Firefighter will get the names of items to take from burning house.
    a. Success Scenario: Software takes in the weight of the firefighter's knapsack and returns the highest value items with the highest total value that fit in the knapsack on the list.
    b. Failure Scenario: Software does not give the most optimal items to grab that will fit in the knapsack.

3. System Features

System features include: server is running all the time and results of the program are displayed in real time. The system is very consistent and reliable. Anyone from the firefighter department will be able to open the webpage and view Thomas's most important items to save.

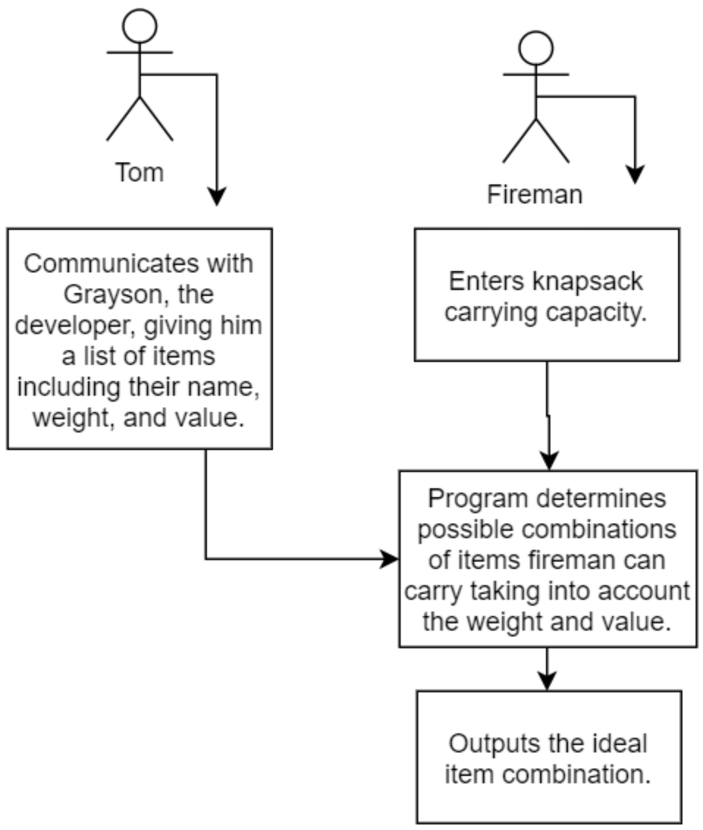
4. External Interface Requirements

The external interface must clearly show the items that the firefighter needs to grab. The firefighter must be able to input the name of the resident, the address of the resident, and the weight of their knapsack. The firefighter's interface must not contain advertisements or other distractions.

5. Other Nonfunctional Requirements

Grayson stays on as maintenance role for 6 months, the rest of wildfire season. Grayson answers phone calls or calls back within 8 hours. Grayson will update Thomas's list within 48 hours of Thomas requesting it.

Program Design Document:


Tom
Fireman
Communicates with Grayson, the developer, giving him a list of items including their name, weight, and value.
Enters knapsack carrying capacity.
Program determines possible combinations of items fireman can carry taking into account the weight and value.
Outputs the ideal item combination.

Source Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdexcept>
#include <memory>
#include <sys/time.h>

using std::cout;
using std::endl;
using std::cin;

struct KnapsackTask
{
        struct Item
        {
        std::string name;
        unsigned w, v, qty;
        Item(): w(), v(), qty() {}
        Item(const std::string& iname, unsigned iw, unsigned iv, unsigned iqty):
        name(iname), w(iw), v(iv), qty(iqty)
        {}
        };
        typedef std::vector<Item> Items;
        struct Solution
        {
        unsigned v, w;
        unsigned long long iterations, usec;
        std::vector<unsigned> n;
        Solution(): v(), w(), iterations(), usec() {}
        };
        //...
        KnapsackTask(): maxWeight_(), totalWeight_() {}
        void add(const Item& item)
        {
        const unsigned totalItemWeight = item.w * item.qty;
        if(const bool invalidItem = !totalItemWeight)
        throw std::logic_error("Invalid item: " + item.name);
        totalWeight_ += totalItemWeight;
        items_.push_back(item);
        }
        const Items& getItems() const { return items_; }
        //void setMaxWeight(unsigned maxWeight) { maxWeight_ = maxWeight; }
        void setMaxWeight(unsigned maxWeight) { maxWeight_ = maxWeight; }
        unsigned getMaxWeight() const { return std::min(totalWeight_, maxWeight_); }

public:
        unsigned maxWeight_, totalWeight_;
        Items items_;
};

class BoundedKnapsackRecursiveSolver
{
public:
        typedef KnapsackTask Task;
        typedef Task::Item Item;
        typedef Task::Items Items;
```

```
	typedef Task::Solution Solution;

	void solve(const Task& task)
	{
	Impl(task, solution_).solve();
	}
	const Solution& getSolution() const { return solution_; }
private:
	class Impl
	{
	struct Candidate
	{
	unsigned v, n;
	bool visited;
	Candidate(): v(), n(), visited(false) {}
	};
	typedef std::vector<Candidate> Cache;
	public:
	Impl(const Task& task, Solution& solution):
	items_(task.getItems()),
	maxWeight_(task.getMaxWeight()),
	maxColumnIndex_(task.getItems().size() - 1),
	solution_(solution),
	cache_(task.getMaxWeight() * task.getItems().size()),
	iterations_(0)
	{}
	void solve()
	{
	if(const bool nothingToSolve = !maxWeight_ || items_.empty())
		return;
	Candidate candidate;
	solve(candidate, maxWeight_, items_.size() - 1);
	convertToSolution(candidate);
	}
	private:
	void solve(Candidate& current, unsigned reminderWeight, const unsigned itemIndex)
	{
	//++iterations_;

	const Item& item(items_[itemIndex]);
```

```
        if(const bool firstColumn = !itemIndex)
        {
                const unsigned maxQty = std::min(item.qty, reminderWeight/item.w);
                current.v = item.v * maxQty;
                current.n = maxQty;
                current.visited = true;
        }
        else
        {
                const unsigned nextItemIndex = itemIndex - 1;
                {
                Candidate& nextItem = cachedItem(reminderWeight, nextItemIndex);
                if(!nextItem.visited)
                solve(nextItem, reminderWeight, nextItemIndex);
                current.visited = true;
                current.v = nextItem.v;
                current.n = 0;
                }
                if(reminderWeight >= item.w)
                {
                for (unsigned numberOfItems = 1; numberOfItems <= item.qty;
++numberOfItems)
                {
                reminderWeight -= item.w;
                Candidate& nextItem = cachedItem(reminderWeight, nextItemIndex);
                if(!nextItem.visited)
                        solve(nextItem, reminderWeight, nextItemIndex);

                const unsigned checkValue = nextItem.v + numberOfItems * item.v;
                if ( checkValue > current.v)
                {
                        current.v = checkValue;
                        current.n = numberOfItems;
                }
                if(!(reminderWeight >= item.w))
                        break;
                }
                }
        }
        }
        void convertToSolution(const Candidate& candidate)
        {
        solution_.v = candidate.v;
        solution_.n.resize(items_.size());

        const Candidate* iter = &candidate;
        unsigned weight = maxWeight_, itemIndex = items_.size() - 1;
        while(true)
        {
                const unsigned currentWeight = iter->n * items_[itemIndex].w;
                solution_.n[itemIndex] = iter->n;
                weight -= currentWeight;
                if(!itemIndex--)
                break;
                iter = &cachedItem(weight, itemIndex);
        }
        solution_.w = maxWeight_ - weight;
        }
        Candidate& cachedItem(unsigned weight, unsigned itemIndex)
        {
        return cache_[weight * maxColumnIndex_ + itemIndex];
        }
        const Items& items_;
        const unsigned maxWeight_;
        const unsigned maxColumnIndex_;
        Solution& solution_;
        Cache cache_;
        unsigned long long iterations_;
        };
        Solution solution_;
};

void populateDataset(KnapsackTask& task)
{
        int mWeight;
        cout << "Enter the weight of your knapsack: \n";
        cin >> mWeight;
        typedef KnapsackTask::Item Item;
        task.setMaxWeight(mWeight);
```

```
	task.add(Item("Cat-Chelsea",9,10,1));
	task.add(Item("Pictures",3,8,1));
	task.add(Item("Jewlery",2,3,1));
	task.add(Item("Hard Drives",2,5,1));
	task.add(Item("Laptop",2,7,1));
	task.add(Item("Guns",1,3,1));
	task.add(Item("Clothes",3,2,1));
	task.add(Item("Wife's Clothes",7,5,1));
}

int main()
{
	KnapsackTask task;
	populateDataset(task);

	BoundedKnapsackRecursiveSolver solver;
	solver.solve(task);
	const KnapsackTask::Solution& solution = solver.getSolution();

	cout << "ITEMS TO SAVE:" << endl;
	for (unsigned i = 0; i < solution.n.size(); ++i)
	{
	if (const bool itemIsNotInKnapsack = !solution.n[i])
	continue;
	cout << "  " <<  task.getItems()[i].name << endl;
	}

	return 0;
}
```

Sample Input/Output Data:

Input from Grayson (developer) and Thomas (user) at the time of software creation:

```
Cat-Chelsea,9,10,1
Pictures,3,8,1
Jewelry,2,3,1
Hard Drives,2,5,1
Laptop,2,7,1
Guns,1,3,1
Clothes,3,2,1
Wife's Clothes,7,5,1
```

Input of firefighter when about to enter burning house:

```
13
```

Output (for above example):

```
ITEMS TO SAVE:
Pictures
Jewelry
Hard Drives
Laptop
Guns
Clothes
```

Document Detailing Testing Strategies:

The software will be tested in a simulation. Grayson will communicate with Tom about the items, of which he will enter into the program. The firefighter will simultaneously be at Tom's home. After Grayson makes sure the website is operational, the firefighter will access it via mobile, ensuring the cellular connection is reliable enough for the program to run. The fireman will enter his carrying capacity and run the program. The program will then display the ideal objects to save. Tests should also be run for every signal band: 4G, 3G, 2G, and roaming under the assumption that a fire may have disrupted service in the area.