PORTA ONE

PortaSwitch

Use Cases

Documentation

www.portaone.com

Copyright Notice & Disclaimers

Easy Way To a Better Development Process, September 2013
Maintenance Release 36
V1.36.4

Please address your comments and suggestions to: Sales Department, PortaOne, Inc. Suite #408, 2963 Glen Drive, Coquitlam BC V3B 2P7 Canada.

Table of Contents

Preface

This document provides PortaSwitch® users with a description of the effective use case creation process necessary for gathering Feature Request requirements. This document aims to provide examples of use cases and demonstrate that they can be the deciding factor for developing functionality that meets users' needs.

Where to get the latest version of this guide

The hard copy of this guide is updated upon major releases only, and does not always contain the latest material on enhancements that occur in-between minor releases. The online copy of this guide is always up to date, and integrates the latest changes to the product. You can access the latest copy of this guide at: **www.portaone.com/support/documentation/**

Conventions

This publication uses the following conventions:

- Commands and keywords are given in **boldface**
- Terminal sessions, console screens, or system file names are displayed in `fixed width font`

The **exclamation mark** draws your attention to important information or actions.

NOTE: Notes contain helpful suggestions about or references to materials not contained in this manual.

Timesaver means that you can save time by taking the action described here.

Tips provide information that might help you solve a problem.

Trademarks and Copyrights

PortaBilling®, PortaSIP® and PortaSwitch® are registered trademarks of PortaOne, Inc.

Bridging the Gap between You and Developers

Naturally, you (the requestor of some new functionality) have a specific business need (to launch a new service, reduce the amount of time spent on manual work daily, etc.) and a pretty good idea of how you want things to work. You do not know, however, what and where exactly changes should be made in the program code to accomplish this.

The developer is in the opposite position: he knows the program code well, but he does not know your business.

So the goal of software requirements is to make sure the developers understand exactly (or at least as well as possible) what functionality you need, and to allow them to choose the best way to implement it. Then the final product will meet your expectations and fulfill its role.

What is wrong with "normal" specifications?

Defining and describing a required system functionality is a tricky process, and there is always a struggle to make it easier and more efficient. Traditionally, requirements would be produced by an "analyst", who would communicate with you and then write down what needs to be added / changed in the system. Unfortunately, there are many things that can go wrong here: you may need to spend a great deal of time educating the analyst about the details of your business; there can be a genuine misunderstanding on the analyst's part; the specification is written in the developers' language, so it is difficult for you to review it thoroughly; and so on.

Imagine a developer working with the code of a system he has never used to perform business operations (e.g. to provide VoIP services). If you ask the developer simply to "add a button for closing a customer record", he will certainly do that. But once the button is delivered, you are likely to become frustrated – because you did not mention that the button should be visible to administrators only (as of course it should), or that the button, when closing the customer record, also should generate a final invoice for that customer. And the developer could not have guessed those extra requirements, because a business context was missing.

Moreover, the more sophisticated the system you are running (and PortaSwitch® is certainly one such system), the more likely you are to get the wrong button from such a "simple request". As additional time is spent on fixing the button, frustration grows on both sides – you lose

valuable time to market (especially if the button is required to launch a new service), while the developer could have implemented other useful features instead of fixing a badly formulated one.

A better approach needs to be taken in order to avoid such a situation.

Use Cases and Their Benefits

So what if instead we allow you to stay in control of the specifications?

You will use your ordinary language to write them (no need to use special words or syntax, or fancy drawings), and simply describe what should happen for you (or the end-user, or someone else involved) when the desired functionality is in place. As a result:

- Reduced production time – the specifications can be easily prepared by anyone in your organization
- Less confusion – the specifications are far more accurate, and it is easy for all project members to review them
- Increased software quality – since your original specifications are used by the QA team, they will actually test that the code fits your needs (and not just the developer's design)

Defining Use Cases

So defining a functionality is done by describing from a participant's perspective what he is experiencing (sees, hears, does, etc.). Note that this participant can be a system administrator, customer service representative, end customer, or even a device such as an IP phone or payment terminal.

A complete description of a situation is called a use case (usage scenario). Use cases normally consist of a name, the involved roles and preconditions, and a sequence of events, which are all better described in plain human language. Here are a few examples:

Use Case "Prorated free monthly minutes"
Roles: Administrator, User
Preconditions: A new product offer includes "1000 free domestic minutes per month" (regular rate is applied to calls after that). All customers have a monthly billing cycle (starting on 1st of every month).

Scenario:

- The administrator assigns a bundle of 1000 free minutes to the user on March 18th.
- The user gets 452 minutes to be used in March, which is 1000 prorated according to the days remaining in this month, i.e. 1000*(31-(18-1))/31.

- The user makes a call 20 minutes long; his remaining free minutes in March are 432.
- The user uses up all 452 minutes and makes another phone call 2 minutes long; he is charged according to the regular rate for those 2 minutes.
- When April starts, the user gets the full 1000 free minutes.

Use Case "Subscriptions not charged while suspended"
Roles: Administrator, Customer
Preconditions: The customer has a monthly billing period, the grace period for the invoice is 15 days, and the subscription is $50/month, charged at the end of month.

Scenario:

- The administrator assigns the subscription to the customer on April 15th.
- The customer is charged $25 at the end of April ($50 prorated for the days when the service was used, i.e. 50*(30-15)/30).
- The customer gets the April invoice on May 1st.
- The customer does not pay the invoice by May 15th, and gets suspended.
- The customer is not charged for any subscriptions while suspended.
- The customer finally pays the invoice on June 21st, and the suspension is removed.
- The customer is charged $24 immediately (50*15/31=24 for the first 15 days in May).
- The customer is charged $17 at the end of June (50*(30-20)/30=17 for the last 10 days in June).
- The customer is charged $50 at the end of July.

Use Case "Prompts for corporate autodialers"
Roles: Traditional PBX, PBX user, PortaSwitch®
Preconditions: The service provider owns a toll-free number 18001234567. PBX is provisioned to use this number and PIN 12345 for outgoing international calls.

Scenario:

- The user dials international number 3155111222 from his office phone (connected to PBX).
- Traditional PBX, detecting an international call, dials 18001234567.
- When the call is connected, PortaSwitch® plays a special prompt (beep) confirming that it is ready to accept a PIN.
- PBX dials the preconfigured PIN 123456.

- PortaSwitch® plays a special prompt (beep) confirming that it is ready to accept a destination number.
- PBX dials 3155111222.
- PortaSwitch® plays a special prompt (beep) once the outgoing leg is answered.
- PBX connects the user to the call.

Use Cases: the Good, the Bad and the Ugly

A good use case possesses the following characteristics:

- It is **observable** by the end-user, i.e. it describes what the user sees, hears or does.
- It is **specific**, so there is a clear and unambiguous way to repeat the described sequence of actions.
- It is **testable**, so there is a clear definition of the measurable final result.
- It is **beneficial**, so we know not only the steps being done by the end-user, but also his initial motivation to do them.

If any of these characteristics are missing, then we have a "bad" or "ugly" use case, which will not do any good for your project. So let's discuss each of these characteristics in detail.

Observable

This is perhaps the simplest one to do correctly, and yet it is the one which is most frequently ignored. The idea is that you should treat the system as a magic box – and thus only talk about how it will interact with the end-user. Something like this:

- You rub the lamp three times.
- A genie appears.
- The genie says, "What do you wish for, master?"
- You say, in a loud, clear voice, "I want a Ferrari."
- Your wish is granted.

That's it – no specifying where the genie is located in the lamp, which protocol the lamp uses to trigger the "appear" event, or in what database column the genie stores the history of fulfilled wishes.

Coming back to the real world, a use case must describe something that the end-user (who may be your customer or an administrator, or even an external web portal) is directly interacting with (information on the screen, data entry via DTMF, invoice PDF, etc.). The end-user does not care which particular piece of software or what data structures are involved to accomplish the results – and neither should you.

Good	Bad/Ugly
User sees $10.00 charge on his next invoice	System charges user $10.00
User enters street address, which may contain up to 100 Unicode characters	Database column address1 is changed to varchar2(100)

Specific

You should always make a use case "specific" by including sample numbers, calculations, and the like. In fact, these are the best use cases, since they allow not only understanding of the desired logic, but also testing of the implementation later on. Use cases which are too abstract, on the other hand, may be interpreted differently by various project members.

The less that is left for the other people to "invent" while interpreting your use case, the lower the chance that something will go wrong. If, for instance, when requesting a Ferrari from the genie, you do not specify where exactly the car should appear, the genie's interpretation may be that it should appear right where this conversation is taking place. But if you and the genie are in an underground cave, or on a desert island, the use-case may work exactly as described, but you will hardly be satisfied with the outcome.

People often think that a lower total number of use cases means that the whole task is "simpler" and "easier to implement". So they create one "mega" use case full of "or" and "may" clauses which are supposed to describe all the possible scenarios. The truth is totally the opposite. It is always better to produce several specific use cases instead of just one which covers everything, and yet describes nothing.

Good	Bad/Ugly
On November 20th user makes a payment of $15.00	User makes a payment
Use case #1: Administrator defines a discount threshold of 100 minutes Use case #2: Administrator defines a discount threshold of $10.00	Administrator defines discount threshold based on either minutes or cost
User makes a call to 44212345678; the regular charged amount is $5.00 First a discount of 10% (according to plan A) is applied Then a discount of 20% (according to plan B) is applied	When the user makes a call (sends an SMS, etc), billing applies discounts in the order that has been specified

Testable

A use case which includes specific data (prices, phone numbers, etc.), a sequence of actions, and a clearly defined end result (which can be observed or measured) can then be properly tested. In other words, this scenario will be reproduced exactly, and the results obtained will be compared with the ones described in the use case. This significantly simplifies the testing process, ensures that the new functionality works exactly as intended, and also serves as a regression testing tool (so one can verify that functionality is still correct in future releases).

Going back to our "genie" example: if you do not specify which exact model or color of Ferrari you want, this may lead to disappointment.

Good	Bad/Ugly
The final charged amount for the call is $3.50	The call is charged with a discount
Invoice includes an extra charge of $0.60 marked as "E911 tax"	Customer is taxed for 911 service

Beneficial

It is important to know not only who the end-user is and what actions he performs, but also why he has initiated this process in the first place. This is usually because the process, when completed, will yield a benefit (saved time or money, a better experience, increased revenue, etc.) for a person or group of persons. These are the beneficiaries of the process, i.e. those who are interested in implementing this process.

Sometimes the end-user is also the beneficiary (for instance, an improved self-care IVR will allow him to check his balance while away from the computer, or a new IP Centrex feature will allow him to handle call transfers more efficiently). Sometimes the beneficiary is a different person within the company (e.g. each employee being able to change the forwarding settings reduces the load on the IT administrator) or even a different company (a new online sign-up portal will allow increased sales).

Knowing the intended benefit is important, since it allows all project participants (you as the requestor, the PortaOne engineering team, etc.) to better understand the ultimate goal. This makes it possible to suggest some improvements or, if some parts of the original scenario are not feasible, to provide an alternative design.

People often assume that the benefits can be deduced from the actual use-case, but this is actually not the case in most situations. Your interpretation of the benefits may be quite different from what others think, even though the process is exactly the same.

In our "genie" use case, someone might assume you need a car to impress your neighbors and friends, so if the genie is out of the desired Ferrari model today, an even more expensive and faster Lamborghini or Aston Martin may be offered as a substitute. But if you were planning to drive to the annual Ferrari owners' club meeting and impress everybody there with your charm, then this would turn out to be a poor substitute. The benefits to you will be much greater if a slightly older model of Ferrari is provided.

So, a good use case should always include a "user story": a description of **why** this whole process is being initiated.

Good	Bad/Ugly
Provider does not want to send customers invoices with small amounts *in order to reduce mailing costs* • At the end of August the total amount of consumed services and subscriptions is $8.00 • An invoice for the August period is not produced	• At the end of August the total amount of consumed services and subscriptions is $8.00 • An invoice for the August period is not produced

Use Cases Versus Functional Requirements

There is always a temptation to skip usage scenarios and go straight to a design / implementation suggestion like "Add option such-and-such to Accounts, and add the following behavior when an Account makes a call". If you jump straight to defining the system functionality, the required options and so on, this means that all of the above analytical work is done on your side. Although your intentions are only the best ("save some work for the developers", "reduce implementation time", etc.) quite often this does more harm than good – just like when patients tell the doctor not their symptoms, but how they think they should be treated.

In fact, quite often the self-proposed "treatment plan" is something that would only make things worse, or even kill the patient. It is just the same with software: frequently a design suggestion will overcomplicate things and not allow the software to operate as intended.

Here is a real-life example showing how change requests and design suggestions may actually lead to *wasted* development efforts.

A customer requested that PortaBilling® functionality be extended by introducing "Second Off-Peak Preference" attribute for the Rate information (in addition to the already available "Preference" and "Off-

Peak Preference" attributes). The explanation given was that this must be done in order to facilitate better routing.

Nothing seemed to be wrong with the request and the developer could have just gone ahead and modified the database schema, prepared the upgrade scripts to convert the tables for all the existing customers, changed the web editor for the rates as well as for the tariff upload procedures, etc.

However, we were curious, since from our experience, a majority of our customers do not even regularly use the "Off-peak preference." Deeper investigation into the actual business-driven reason behind the change turned out to provide interesting information.

Apparently, the service provider was operating at a very low profit margin and there were only a few vendors who could be used to connect calls without suffering significant loss for each call. Therefore, the administrator was assigning preferences to place the "lower-cost" vendors on top of the routing list and was then using huntstop to "cut off" vendors whose prices appeared to be too expensive. Note that this was done based on the administrator's opinion about pricing, so later on, even if some vendors' prices increased or customers' prices decreased – the selection was the same and might still lead to losses.

Consequently, the actual requirement they had in mind was, "Prevent financial losses by not routing a call to what appear to be too-expensive vendors," or (when the use case was written) it looked like this:

Use Case "Vendor selection to prevent loss"

Pre-requisites: Administrator decides on the maximum acceptable amount of loss per minute (0.01). Customer is assigned a rate of 0.05/min for domestic calls. There are 4 vendors capable of terminating calls to Belgium, with the following rates:

Carrier	Regular Price	Weekend Pricing?	Weekend Price	Night-time Pricing?	Night-time Price
A	0.09	Yes	0.07	9pm-7am	0.045
B	0.06	No	--	10pm-8am	0.05
C	0.07	No	--	No	--
D	0.075	Yes	0.045	8pm-7am	0.055

Scenario #1:
Customer calls 32212345678 at 8:30pm on Saturday.
While arranging potential routes (A: 0.07/min; B: 0.06/min; C: 0.07/min and D: 0.055/min) and considering customer's price per minute (0.05), PortaSwitch® should also account for maximum per-minute loss.
As a result, vendors A and C are too expensive so the routing list is B, D.

Scenario #2:
Customer calls 32212345678 at 10:30pm on Saturday.
While arranging potential routes (A: 0.045/min; B: 0.05/min; C: 0.07/min and D: 0.055/min) and considering customer's price per minute (0.05), PortaSwitch® should also account for maximum per-minute loss.
As a result, vendor C is too expensive so the routing list is A, B, D.

After this was documented and approved by the customer, it became obvious that the only modification required was to allow the "profit per minute" parameter of the "Profit Guarantee" module to contain negative values (enabling routes with limited losses). As a result, the modification time was much shorter than the original request – and (most importantly), it addressed the original customer's problem with a targeted solution!

Route selection is now done automatically based on the current price of the vendor – reducing the quantity of manual operations and eliminating the risk of revenue loss because of administrator mistake or negligence.

This is why it is crucial to share your actual business needs (or required improvements or desired processes) and document them in the form of use cases:

- The solution will be designed to address the actual business need
- You can write the specification yourself
- A development team can then find the best way to address the problem in the shortest implementation time.

Appendix A. Use Case Template

Copy&paste the text below into your own document.

Use case #1: Name (*Redirect from IP phone)*
Roles: Involved parties, for instance: *Administrator, User*
User Story: Allowing end-users to program a "forward to" number directly into their phone, so they can use a function they are already used to from traditional PBX. This will reduce the amount of training required for new customers.
Preconditions: Actions to be performed beforehand, e.g.: *The user has VoIP number 1604111 and mobile number 1604555.*

Scenario:
Sequence of actions or events, for instance:

- *The administrator allows the user to redirect calls from his IP phone.*
- *The user dials a special prefix on his IP phone and enters 1604555 as the forwarding destination.*
- *An incoming call to 1604111 is received on the IP phone and forwarded to 1604555.*
- *The user receives a call on his mobile phone.*

Check-list:

- There is a "user story", which mentions **why** this process is required (what the benefits will be).
- The use case describes events or data directly **observable** by the end-user.
- There is **specific** data (numbers, names, prices, etc.) which allows only one way to reproduce the use case, so there is no need to invent anything.
- There is a definition of the required end result, so the use case is **testable**.