Thesis

Szabó Márk

Debrecen

2009

Debreceni Egyetem Informatikai Kar

Command-lines in the web age

Supervisor:

Dr. Juhász István
egyetemi adjunktus
(university assistant professor)
DE-IK

Written by:

Szabó Márk
Programtervező informatikus M.Sc.
(Master of Information Technology)

Table of Contents

Introduction

Abstract

Today's digital world offers various opportunities for both the tech lover and main stream users. They don't just surf on the ever growing internet, but use innovative, social centric web applications day by day. Application designers aim to create easy to use, easy to understand apps for the masses. These sites gain plenty of users in no time.

However information technology's Holy Grail; *application interoperability and information exchange* are still *open issues*. Ordinary users simply can't manage their personal content effectively. They upload videos and photos, share bookmarks, listen to podcasts or on-line radios; they live a rich digital life. Typically these activities involve the use of many sites and computers at different locations. All these apps are good at managing their own functionalities and content; but how can we solve simple problems like the followings?

1. Publish blog entries with funny pictures and videos about our last trip.
2. Collect and present some interesting link, picture, streaming video, related map fragments (mash-ups) and share it with our friends in a human-friendly form.
3. Use our favorite web application's functionality everywhere with ease
4. Apply web apps to store our content securely and distributed
5. Present a given piece of our content in web based form

Service-**O**riented **C**ommand **L**ine (SOCL) tries to solve these difficulties. SOCL system core provides an infrastructure designed with *easy*, *plug-in style extensibility* in mind. Application logic developers can use all the server-side methods they already familiar with to implement their own creative ideas, or wrap *third-party APIs*, like Google, Flickr or Facebook API. SOCL will couple the core with a *revolutionary user interface* based on Mozilla's *Ubiquity to leverage the natural association between high-level web services and command-oriented task execution.*

Current problems with computer interfaces

Computer science was born as a few scientist's toy in the early '50s. The way to the practically computer-based present was long. Unfortunately user interfaces couldn't evolve enough to open the gates of digital world to everyone.

At the very beginning technically skilled crew could use the computers. When large companies started to adopt mainframes to support their business, the need for simpler and easier to learn user interfaces emerged. These were the first operating systems with *command line interfaces* and script support. Executing programs written in *higher level languages*, like *FORTRAN* and *Cobol* is much better, then using assembly and machine languages. However CLIs still require considerable knowledge. CLI is effective in a wide range of tasks, short, fast, and powerful, but fails when average users tries to use a pc even for simple office work like word processing, calculations etc. .

The appearance of *graphical user interfaces* offered to a lot more people the exceptional power of computers. Although GUI's offer sophisticated abstractions like *desktop*, *files*, *objects* with graphical notations, 2D/2,5D navigation, these interfaces simply can't scale well. Before the „digital boom“ users stored only a small amount of files and used a few programs. In these days everybody has large digital photo albums, tons of music and video, ebooks and specialized programs to consume these content types. However, placing only 30 icons on a window or desktop makes it totally nonperspicuous. Menus with more than 3 levels are hard to use and it is very easy to get lost. This again made difficult to average users to handle their content and command their pc effectively.

The solution to this problem can be the *reinvention of the CLI*. Original CLI is powerful when used by an expert, but suffers from issues rooted on the fact that it's main task is to control the computer in the system level. For example CLI expects accurate commands, which usually differ on different systems, and has a large number of optional parameters which make these commands powerful, but also make them really hard to learn. The *power of natural language* coupled with

intelligent search and *suggestion* offers a new possibility to overcome these problems. Users no longer need to memorize the available commands, they just search for the appropriate one. What's more, intelligent suggest system helps to find the right one with *offering multiple possible solution*. Moreover these interfaces can *scale well*, because they are independent from graphical representations.

Google Search

A great example of this type of thinking is the Google Search. We just need to type some search phrases and a few options to refine the search process. The result is a bunch of links which generally closely match our expectation. However it is less known that Google provides much more under the hood in Google Search. A few examples:

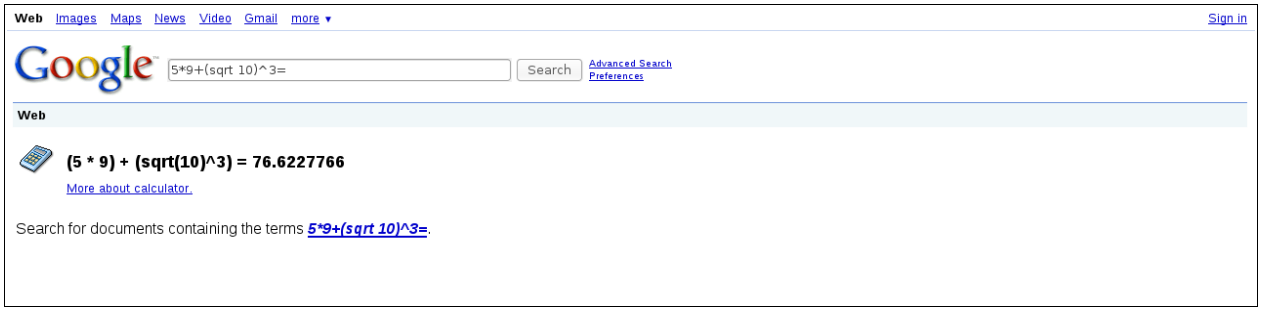

Figure 1. : Calculator

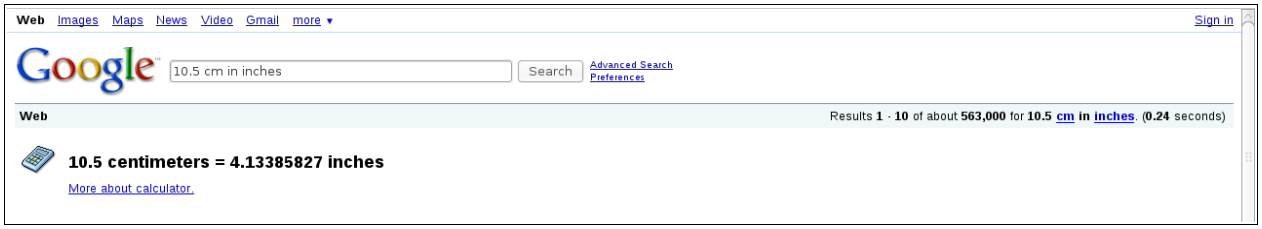

Figure 2. : Unit conversion

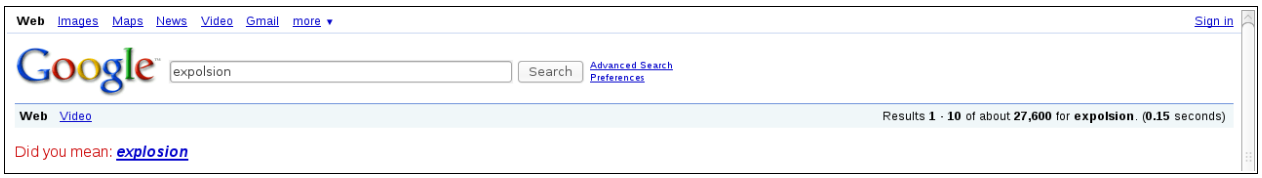

Figure 3. : Spell checker

These commands provide simple services, but are inherently different and exceed others, which are basically just special search options, like sport, weather, stock, book or dictionary searches. These commands are *simple CLI program executions*.

Other mentionable, but underestimated desktop solution are *Quicksilver for Mac, Enso for Windows and GnomeDo for Linux*. All of them apply these ideas to simplify and speed up common tasks like finding a file in a deep filesystem, running programs or controlling certain applications via direct commands. Take a look at their short description and screenshot, which enlight their concepts!

Quicksilver

Let's see how Quicksilver documentation tries to explain the software!

"*At first glance, Quicksilver is a launcher. When opened, it will create a catalog of applications and some frequently used folders and documents. Activate it, and you can search for and open anything in its catalog instantly. The search is adaptive, so Quicksilver will recognize which items you are searching for based on previous experience. It also supports abbreviations, so you can type entire words, or just fragments of each. When not in use, Quicksilver vanishes, waiting for the next time you summon it. "*

Figure 4. : Quicksilver invoked, browsing object suggestions

Figure 5. : More advanced examples, moving a file and emailing a file

Using Quicksilver is straightforward and simple:

1. Invoke Quicksilver, Quicksilver panel fades in.
2. Type file/folder/program name, choose from the given suggestions.
3. Hit tab, and choose a command to execute.
4. Hit enter. Command gets executed, Quicksilver panel fades out.

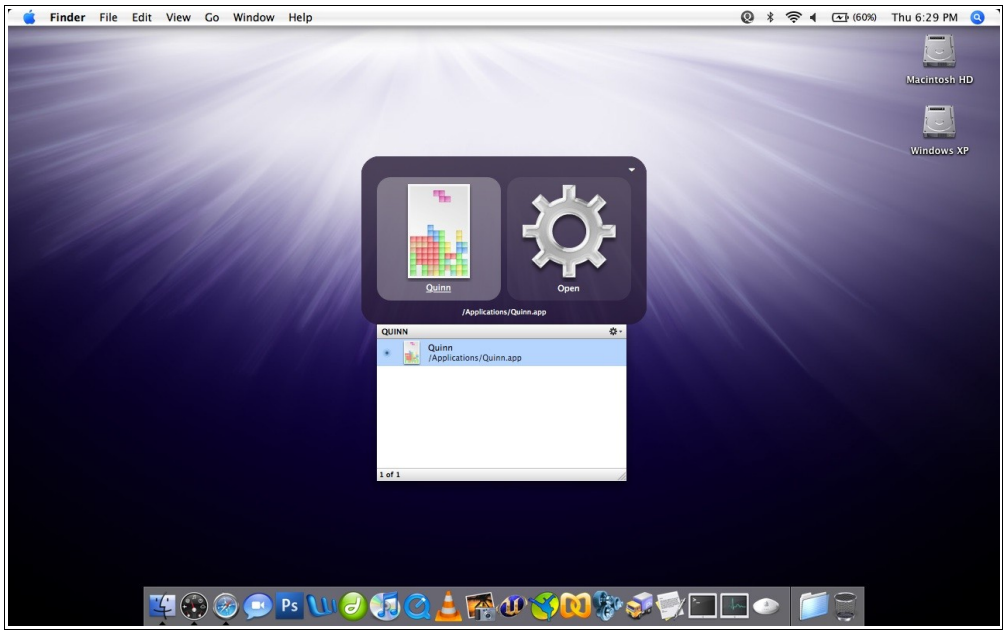

Figure 6. : Quicksilver in action

GnomeDo is practically the same, but implemented under Linux. Both system tries to utilize *extensive search* and *user behavior learning* coupled with a *minimal, nonmodal GUI.*

This approach does shine in boosting everyday's common work. Launching programs? Editing

files? Searching the web? Managing playlists? All these task are equally easy, when launchers have more optional plug-ins installed.

Enso

A solution developed to be the launcher on Windows platform. Easiest way to describe it to borrow Walt Mossberg's words:

"*Enso is dead simple to use. You just hold down the Caps Lock key and type an Enso command, which is displayed in a translucent overlay. Once the command is typed, you simply release the Caps Lock key to activate it, and the overlay disappears. If you type fast, it all happens in a flash. For instance, to launch the Firefox Web browser, you just hold down the Caps Lock key and type "open firefox." To look up the meaning of the word "proclivity," you just hold down the Caps Lock key and type "define proclivity." "*

Nearly the same as Quicksilver/GnomeDo, but instead a minimal GUI it uses *pure command line* approach. Personally I think that this makes it less user-friendly in a modern OS environment, but Enso is still a good choice on Windows.

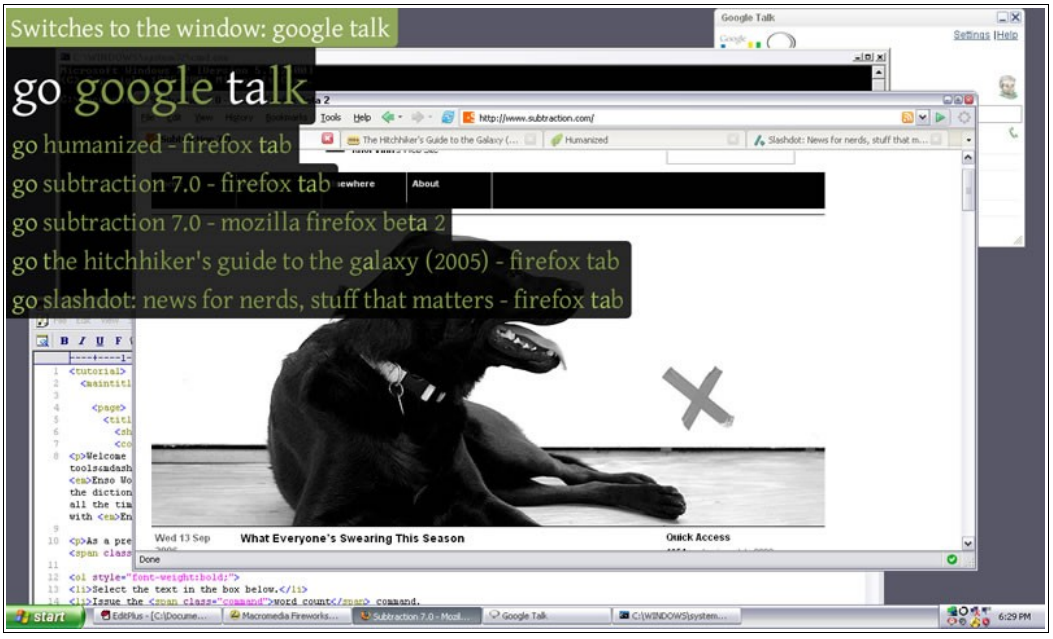

Figure 7. : Enso in action

Mozilla Ubiquity

Ubiquity's story began when *Aza Raskin*, a creator of Enso joined to Mozilla Labs. This project is unambiguously builds upon Enso; it is the *first command line extension* built into a *browser*. Web browsers are key part in today's information systems, however the world wide web is fundamentally different according to desktop or client-server based computer environments. So how can such a command line look like, and why is it useful? For the answer, I quote from the official web page describing Ubiquity:

"Ubiquity is a Mozilla Labs experiment into connecting the Web with language in an attempt to find new user interfaces that could make it possible for everyone to do common Web tasks more quickly and easily.

The overall goals of Ubiquity are to explore how best to:

- *Make it extremely easy to Extend browser functionality, and share new functionality with other users.*
- *Enable on-demand, user-generated mashups with existing open Web APIs. (In other words, allowing everyone—not just Web developers—to remix the Web so it fits their needs, no matter what page they are on, or what they are doing.)*
- *Empower users to control the web browser using a natural-language-like command interface. (With search, users type what they want to find. With Ubiquity, they type what they want to do.)*
- *Use Trust networks and social constructs to balance security with ease of extensibility."*

Benefits of such a system is remarkable. These functionalities can be extremely useful in web environments. Invoking web services and reaching content directly can speed up our internet activity. ***What's more, web services can be arbitrary diverse, coarse or fine grained and can be combined relatively freely. At the moment this technology represents the ultimate solution in reusability, platform independence, interoperability and utilizes standard-based interfaces in addition.***

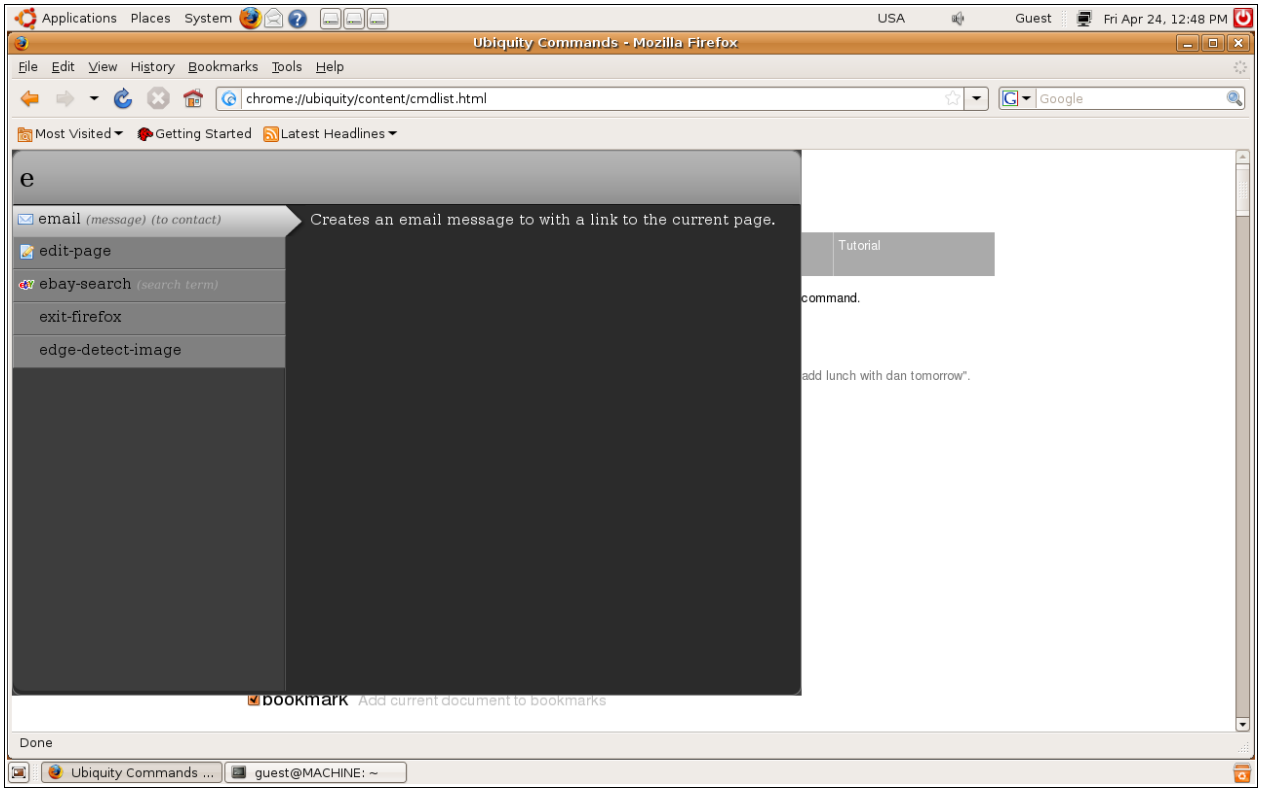

Figure 8. : Ubiquity invoked in Firefox

The screenshot above shows Ubiquity console. It has 3 main sections. The first is an input field where users can type in commands. Second is the suggestion box, where appropriate suggestions appear. The third box is reserved for previews. This segment can contain html code, image or video, so it is very flexible. It can even get it's content using Ajax. However Ubiquity is still under development, about everything can change in the upcoming versions. For example developers are trying to integrate Ubiquity into the Firefox's Awesomebar. (Current software version used in my thesis is 0.1.8.)

Ubiquity offers straightforward tools for command development. In fact developers need just to write Javascipt according to predefined templates, but jQuery, a popular Javascipt library is built in to simplify creating new commands. Furthermore creators can define own parameter types, called nouns even with custom suggestion logic and of course their own previews including nearly arbitrary web content.

Intention

Everything began with a Youtube video presenting an early Quicksilver launcher. It proved me that there can be alternatives, and it doesn't have to be that difficult and time-consuming to use computers. However, I usually use Linux, so I searched the web to find Quicksilver's Linux port. Then I found GnomeDo. Not an official port, but inspired mainly by Quicksilver and written to work under Linux. I have been using it for more then a year, and I feel that it really accelerates my everyday work. *We have alternatives to the good old desktop metaphor!* Later I heard about Enso, and Humanized.com . Except that I am happy with GnomeDo, it seemed to be interesting. I started to take up the question of designing better user interfaces.

Tools developed to control our desktops are evolving on a good way. Nevertheless one serious problem remains. *Desktop applications raise walls around itself and encapsulate all of their functionality without external interfaces.* Truth to say Apple applications can expose a bit of their functionality to use for example in AppleScript scripts. But no other platform aims to standardize application interoperability.

This is why world wide web becomes important. *Web applications share a common presentation layer* (html and active content, like Javascript or Flash), *a common architecture pattern* (thin client-server), *common communication standards* (Http) *and common resource addressing schema* (Url). In addition web service, a promising newcomer in information technology can be the solution for the problems from traditional desktop softwares suffer.

Ubiquity project is good starting point, but limits programmers facilities, because commands have to be implemented on client side. My thesis work builds upon Ubiquity, but places *command execution* to the *server-side*. I think that developers will have much more freedom in this approach, thanks to the advanced enterprise computing solutions, which can handle transactions, persistence, messaging, communication and webservices equally well. This project tries to reach this goal providing a scalable and easy-to-extend infrastructure for high-level, human-friendly command execution.

System requirements

Definitions, acronyms

SOCL	Service-Oriented Command Line: common name for the whole system including Ubiquity-based GUI, core infrastructure and CEUs.
CEU	Command Executor Unit: system components, which connect to the core infrastructure with implementing expected interfaces and include specific commands.
Command	See Core.Model.1 requirement for complete definition. I use this term for all representation of commands. (Including client, and server side forms.) Context will refine the concrete meaning.
Ubiquity	A command line extension plugin for Mozilla Firefox.
JQuery	Popular Javascript library. Ubiquity contains and utilizes this framework as default.
AJAX	Asynchronous JavaScript + XML: Common method used to create Rich Internet Applications. (RIA)
REST	Representational State Transfer: Collection of network architecture which outline how resources are defined and addressed. In my thesis this term refers to the Jersey framework.
JEE, JEE 5	Acronyms for Java Platform, Enterprise Edition 5. Popular framework for distributed, web-based enterprise software development.
Jersey, JAX-RS	REST framework for enterprise Java environment. Currently in test phase, but will be the part of the upcoming JEE 6.
JAXB	Java Architecture for XML Binding, a JEE component for XML handling.
JMS	Java Message Service: Java technology which realizes message-oriented communication. It supports point-to-point (Queue) and broadcast style (Topic) message forwarding. Part of JEE 5.
Session bean	JEE component for server-side processing.
Message-driven Bean	JEE component for server-side asynchronous (message) processing. Closely related to JMS.
Web service	Synchronous server-side processing component with standardized communication interface.

Core requirements

Requirement ID: Core.Comm.1

Description: SOCL system must conform to common web standards. This means among others that SOCL is inherently client-server based and has to communicate a standard message-driven way.

Requirement ID: Core.Comm.2

Description: SOCL system has to expose it's services as standard REST-style web services.

Requirement ID: Core.Model.1

Description: SOCL system consumes commands. Command model is the following:

1. Command name
2. Command namespace: Dot separated character string. All command has to be a part of one and only one namespace. Namespaces delimit commands from each other. Example: socl.builtin.pictures
3. Attribute list containing attribute names and types. Attributes types are abstract constructs, which model a high-level entity. Attribute types are predefined in the system. Example: photo, picture, video, text

Requirement ID: Core.Model.2

Description: SOCL system commands return HTML fragments, which represent response values.

Requirement ID: Core.Services.1

Description: SOCL system has to implement a component, which generates suggestions for a given command fragment. This command fragment must be matched against all available commands and a similarity metric must be computed. Suggestions are the commands with similarity metric value lower then a predefined limit. Suggestions must be returned in 2 seconds.

Requirement ID: Core.Services.2

Description: SOCL system has to implement a component, which returns detailed description of a given command. Descriptions must contain all command attributes, namely: command name,

command namespace, command attributes and related attribute type.

Requirement ID: Core.Services.3

Description: SOCL system has to implement a component, which executes commands described in requirement Core.Model.1 and generates HTML page as a response.

Requirement ID: Core.Services.4

Description: SOCL system must offer an easy namespace installation and deletion mechanism, which requires only system restart, but not recompilation.

User interface requirements

Requirement ID: UI.Ubiquity.1

Description: SOCL system must implement one Ubiquity command, which can communicate with services described in requirement Core.Services.1, Core.Services.2, and Core.Services.3 .

Requirement ID: UI.Ubiquity.2

Description: Ubiquity command described in UI.Ubiquity.1 must implement an Ubiquity noun type for suggestion retrieve.

Requirement ID: UI.Ubiquity.3

Description: Ubiquity command described in UI.Ubiquity.1 must show command description for the currently selected command in Ubiquity preview section.

Non-functional requirements

Requirement ID: NonFunct.Performance.1

Description: All suggestion operation must provide response in 2 second from the time, when the corresponding request was sent.

Requirement ID: NonFunct.Performance.2

Description: Executing commands related to external systems can't be slower then 130% execution time, compared to realizing the same functionality in that external system is 100% execution time.

Requirement ID: NonFunct.Extensibility.1

Description: Main software components should be designed using IT best practices. Extensive use of interfaces and reusable design is a must.

Requirement ID: NonFunct.Portability.1

Description: SOCL system must be portable among application servers of the same programming platform with ease; this means it can't use application server specific solutions and porting it to an other, standard application server can desire only descriptor file changes.

Requirement ID: NonFunct.Scalability.1

Description: SOCL system has to scale well with massively increasing user community and command repository.

Requirement ID: NonFunct.Reliability.1

Description: The system must log all relevant events.

Requirement ID: NonFunct.Documentation.1

Description: All source code must contain basic documentation for key methods

System design

Use cases

It's obvious from the introduction chapter that the system is an *interpreter*. This means that real, useful functionality is encapsulated in the form of commands, SOCL core provides the required infrastructure for command execution. So use case diagram is pretty clear as shown on figure number 9.

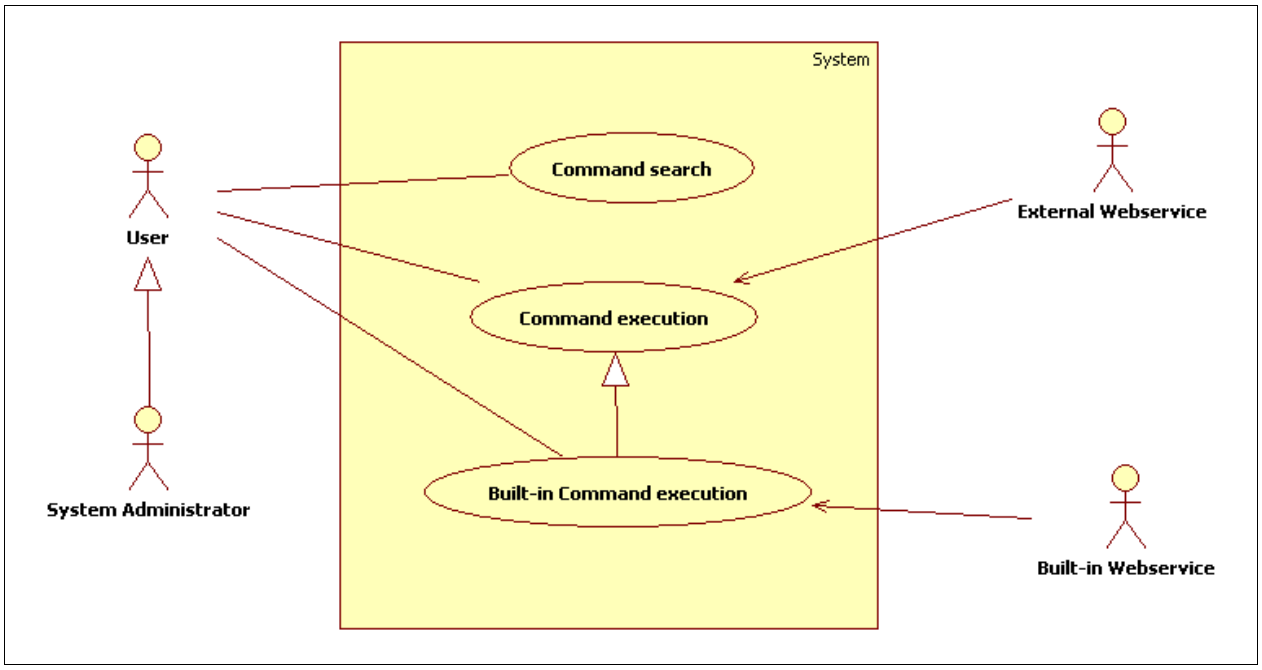

Figure 9. : Use case diagram

Explanation

Users are divided into two main groups: ordinary users and system administrators.

System administrators have the rights to setup and maintain both the system core and CEUs. They can utilize predefined administrator-only commands for their tasks.

Ordinary users can search for commands, request command descriptions and execute chosen commands.

CEUs can connect to external web services. In fact, some CEU just a wrapper for external web service APIs. All exact scenario depends on the chosen command. Generic scenarios are the following:

Scenario 1.

1. Invoke Ubiquity.
2. Run SOCL command, called “execute”.
3. Type in a command name or command name fragment.
4. Choose from the suggestions.
5. Type in required and optional arguments
6. Execute chosen command.
7. (Optional) Save response link as bookmark.

Scenario 2.

1. Open a bookmarked link.

High-level design

This section describes SOCL system from high-level viewpoint. At first look at the class diagram on figure number 10.!

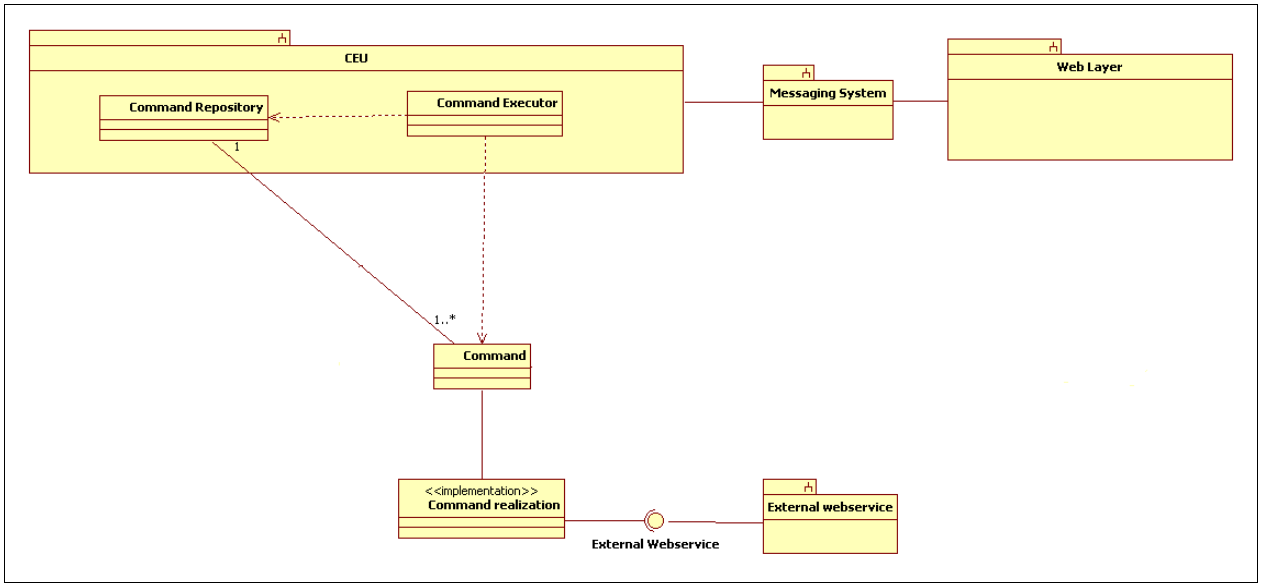

Figure 10. : Class diagram

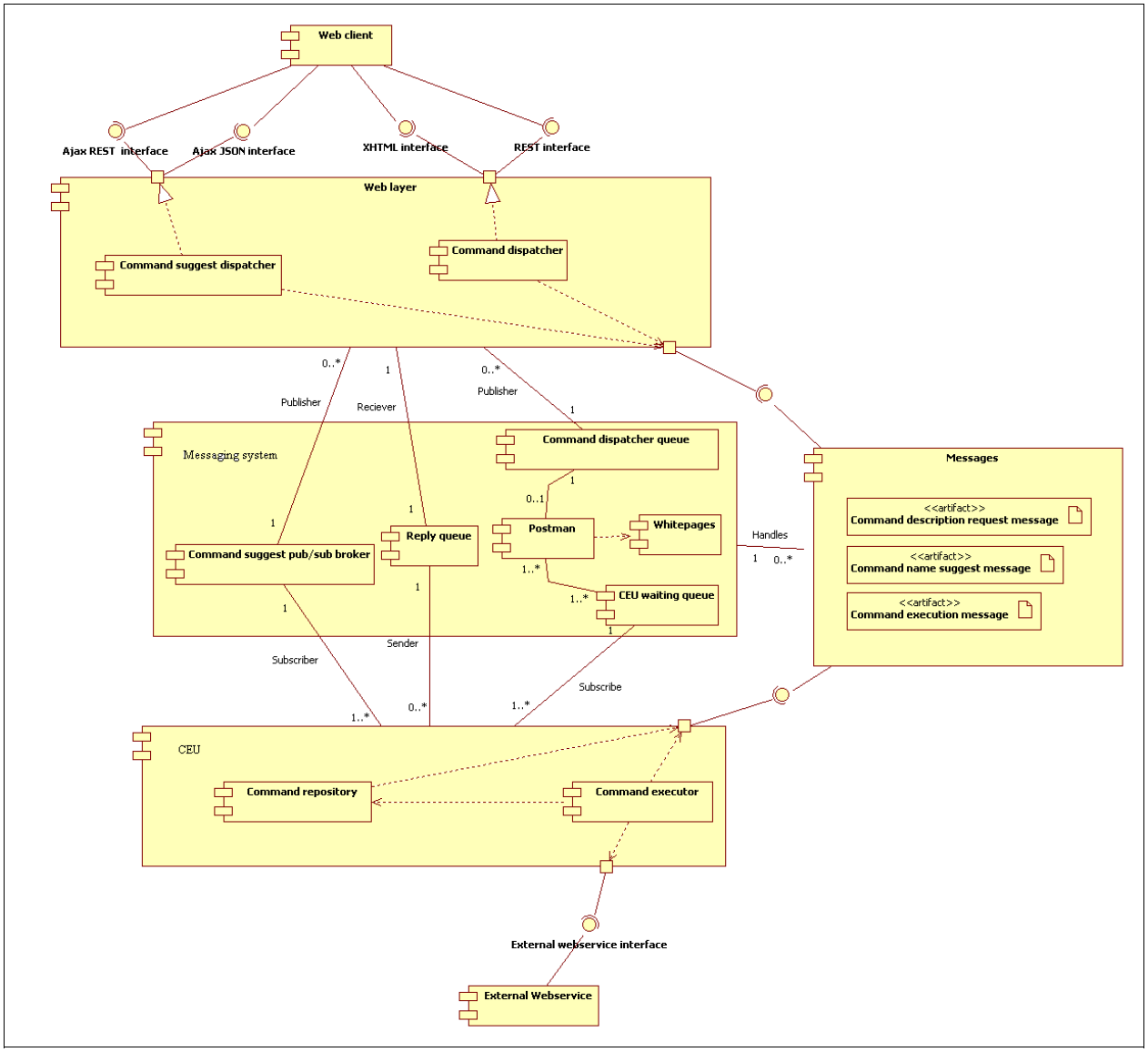

Figure 11. : Component architecture diagram

Explanation (class diagram)

This is an *abstract blueprint*, where all main concept appear. SOCL system has *3 logical component*:

1. ***Web layer***
2. ***Messaging system***
3. ***CEUs***

Web layer is responsible for *communication with the clients*. It doesn't contain business logic, but processes and transforms user requests, and generates responses.

Messaging system connects CEUs and the web layer. *Communication is message oriented*. Web layer transforms user request into appropriate request messages, and gets back results as response messages from one or more CEU. SOCL system defines all required messaging interfaces with agreements between the receiver and sender side.

CEUs, or command executor units contain all necessary information about implemented commands. All of them has a command repository, which is a central storage for command description. It also stores information about implementation for each command. *All CEU represents a namespace*. All commands in a CEU (a namespace) must have different names! System administrators can install, replace or delete CEUs one by one.

SOCL system's adaptable infrastructure bases on message-oriented communication, and easily extendable collection of CEUs.

Commands in SOCL system are *abstract entities*. These commands provide *unified access* for various *resources*. For example underlaying realization can be a simple server-side enterprise application, a static web page or a web service. This abstraction makes high-level commands extremely valuable, because clients don't have to be aware of diverse technologies, but can reach and transform content using the same method, namely typing commands. I believe that this approach brings in *task-oriented web usage* instead of application-oriented. Now examine figure number 11. !

Explanation (component architecture diagram)

This diagram visualizes a more fine-grained view of the system. I explain the components in a top-down order.

Client

SOCL system is web based, so clients are simple web browsers. Web layer provides two interfaces for them. First is an Ajax interface for command suggest, description and execution

requests initiated by Ubiquity. Second is a standard XHTML interface for the command response.

Web layer

As mentioned above web layer is responsible for communicating with clients. It dispatches Ajax calls to dispatcher objects.

Command suggest dispatcher is responsible for wrapping suggest requests. *Command suggest dispatcher* then sends *Command name suggest messages* to the *Command suggest pub/sub broker*.

Command dispatcher packages describe and execution requests into messages. *Command dispatcher* forwards the messages to the *Command dispatcher queue*.

Both dispatcher objects use temporary queues as unique post-office boxes. CEUs send response messages to the appropriate reply queue.

Messaging system

Messaging system provides the communication channel between the web layer and CEUs. *Messaging system* manages *Command suggest pub/sub broker, Command dispatcher queue, CEU waiting queues* and individual *Reply queues*.

Defined message formats represent the conversational agreements between communication partners. *Command suggest request messages* are delivered to all subscribers (publish/subscribe model).

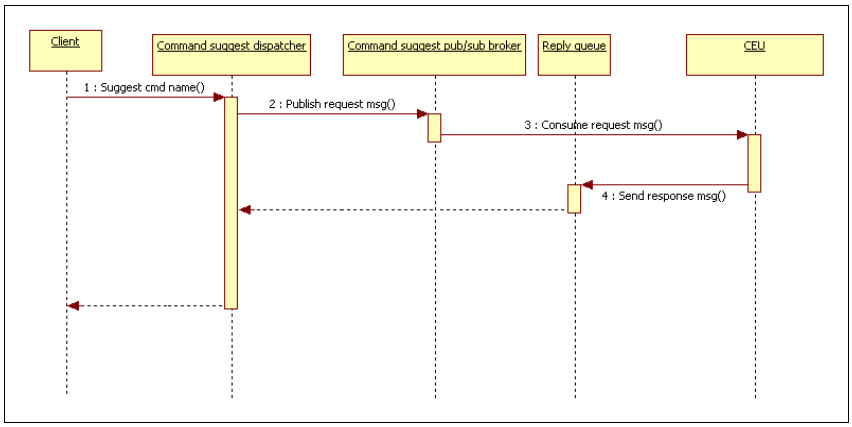

Figure 13. : Sequence diagram describing suggest request message processing

Command description request and *Command execution message* delivery is different. At first both type of messages arrive at the *Command dispatcher queue*. A special object, called *Postman*, then looks up the recipient in the *Whitepages*, and forwards the message to the appropriate *CEU waiting queue.*

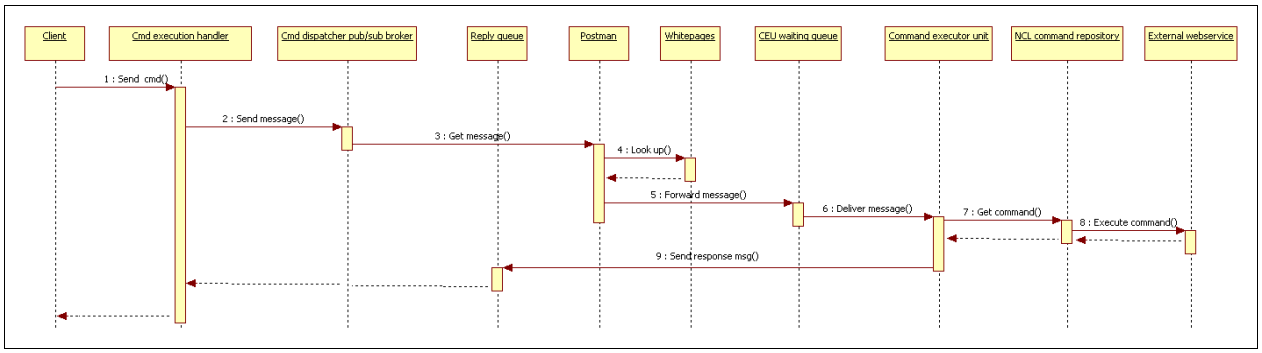

Figure 14. : Sequence diagram showing execution message processing

Command Executor Unit

CEUs encapsulate business logic and represent an abstract namespace for commands. These components answer command suggestion, describe and execution message. *Command Repository* subcomponent is responsible for storing static data, such as command names, attributes and implementation information. *Command Executor* components implement exactly one command.

Now advance to the next chapter, which deal with SOCL system implementation.

System realization

Technology background

This section briefly demonstrates applied technologies. Most of them is stable and production ready, but key features are implemented using experimental software versions. Although remaining bugs are annoying, these versions indicate near future trends.

Mozilla Ubiquity

Introduction chapter contains a whole section about Ubiquity. I had to get acquainted with main design concepts and *JQuery* before I was able to develop commands for this plugin. Ubiquity UI communicates with SOCL web layer using Ajax requests. Current version at the time of writing is 0.1.8. .

JQuery

JQuery is an exciting newcomer in the Javascript world. JQuery is built around the idea called *Unobtrusive JavaScript*. This framework tries to help page authors to write as few Javascript code as possible. It separates client-side data from client-side presentation logic. This makes HTML markup less cluttered, because all scripts are defined at the beginning of the page.

JQuery framework heavily rests on a CSS-like selector mechanism. A common scenario for doing some transformation is selecting all elements we want to change, and apply the chosen transformation to these elements. Since nearly all type of functions return the selected elements (called the *wrapped set*), it's easy to apply more than one transformations for a set of elements.

What's more, JQuery team worked hard to hide browser differences as much as possible. Thanks to their effort, JQuery users can create cross-browser compatible scripts. Of course, JQuery provides event handling support, many effects, UI components and much more. Tons of plugins realize various useful functionalities. This makes this relatively new framework even stronger. JQuery is included in Ubiquity as default. Current version at the time of writing is 1.3.2. .

JEE

Java, Enterprise Edition is a well-known framework for developing enterprise applications. It offers *component-based*, highly *secure* and *fault-tolerant platform*, which inherently *supports multi-tier, distributed applications*. Java EE programs run on *application servers*.

JEE is *de facto standard*. *Java Community Process* governs the continuous development of the *JEE specification*. An application server is *JEE compliant*, if it conforms to a predefined set of

requirements. Current JEE version is JEE 5, but my thesis use a feature coming with JEE 6. I adopted *Glassfish v2* application server.

Web layer was developed as a collection of *REST* web services. These serve as the communication endpoint for Ajax requests. REST web services in JEE are extensions of the existing *Servlet* technology. Jersey framework introduces a lot of interesting new features. Developers create *resource classes*. Methods in these classes are the *resources*. A resource can be reached with a unique U*rl pattern*. Resources can consume and produce various MIME-typed data, like simple text, html, XML and JSON. All configuration is accomplished using *annotations*. Simple, but yet powerful solution. Moreover Jersey framework provides automatic transformation between objects and XML or JSON with the help of *JAXB*.

Java Architecture for XML Binding, or *JAXB*, is a JEE component, with which developers can define Java object mapping for XML documents. Building such representation is called *unmarshalling*, while creating XML markup from the Java mapping is called *marshalling*. This tool greatly simplifies XML and JSON handling, because it's easy to switch representation between communication and programming-friendly form.

Session beans are widely used throughout the layers. These are simple *POJOs* (Plain Old Java Objects) annotated with @Stateless or @Stateful annotations. However, session beans are much more, than simple objects, but objects managed by the applications server's *EJB container*. EJB container offers important services for enterprise beans, like *lifecycle management*, *transaction* support, *security*, *remote communication*, *timer service*. Enterprise beans are the tools for developing *scalable* and *fault-tolerant* components.

Message-driven beans are enterprise beans annotated with @Messagedriven annotation. MDBs differ from session beans in communication manner. While session bean methods are invoked directly, MDB is message-oriented. It listens for incoming messages from a *JMS destination*. When it receives a message, it consumes the payload and initiate some actions. These components are the main programming instruments for *asynchronous processing*.

Java Message Service, or *JMS*, provides facilities for *message-oriented middleware*. JMS

supports *point-to-point*, and *publish-subscribe* style *messaging*. *Destinations* realize different strategies. *Queue* is responsible for the first, while *Topic* for the second strategy. Both of them can persist messages to achieve *reliable messaging*. Most applications servers have built in JMS implementation, however it's possible to send messages to remote JMS servers. It's even solvable to build highly reliable, clustered JMS servers.

Web service is an actual buzzword in information technology. It seems to solve old and complicated problems like standard inter-machine communication between different platforms and technologies. W3C definition is the following: “A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.” W3C standards define all important aspects of web services. Major IT companies like Microsoft, Sun, Oracle, or IBM support this technology.

Web API

Internet is growing faster and faster in every year. Pioneers in the new industry created innovative applications nowadays called *Web 2.0*. Best examples are search engines (Google, Yahoo, Msn), wikis (Wikipedia), online photo galleries (Picasa, Flickr), social networking sites (Myspace, Facebook, Hi5, Orkut), blogs, rss feeds, online video sites (Youtube), microblogs (Twitter) and much more (digg, del.icio.us, etc.).

Many of these sites can offer their services as web services, typically REST-style, but most of them can handle “traditional” SOAP-based, too. We call the collection of all web services provided by a Web 2.0 site *Web APIs*. With the help of these APIs, developers can produce new web sites with functionality borrowed from popular Web 2.0 applications. *Mashup* term refers these mixed web apps.

XML

Extensible Markup Language is a subset of *SGML*. Both are *meta languages*, which purpose is creating *custom markup languages*. *XML specification* formulate all the expectations against such language, so developers can define their own. Custom markup languages are good at describing *structured data*, these are widely applied to *machine-to-machine communication*. But XML is much more. There are many other useful tools like *XSLT*, a standard for document transforming or *XPath*, a standard for searching in documents.

Yet still all XML documents are just *simple texts*. Processing text is an easy task even in older programming languages, so *legacy systems* can be extended to *adopt XML*. This is why XML gains more and more popularity in IT and major companies uniformly agreed to favor this W3C standard.

JSON

Javascript Object Notation,or *JSON*, is a lightweight data exchange format. It is based on a subset of Javascript programing language, however considered language independent and cross-platform solution. Because it is lightweight and can be natively processed in Javascript scripts, it gained popularity on the web, especially in browser-server communication.

Implementation details

This section is about what solutions I developed, and what problems I had to face with during development. I'll discuss the layers in top-to-down order.

Client-side

My client software is based on Mozilla Ubiquity. This means that my approach is browser-dependent, which is also a drawback. But this plugin offers revolutionary UI and clear API for 3rd party command development. I think these advantages are worth sacrificing portability. Ubiquity UI is divided into 3 main parts. Let's have a look at figure 15., where I marked these parts!

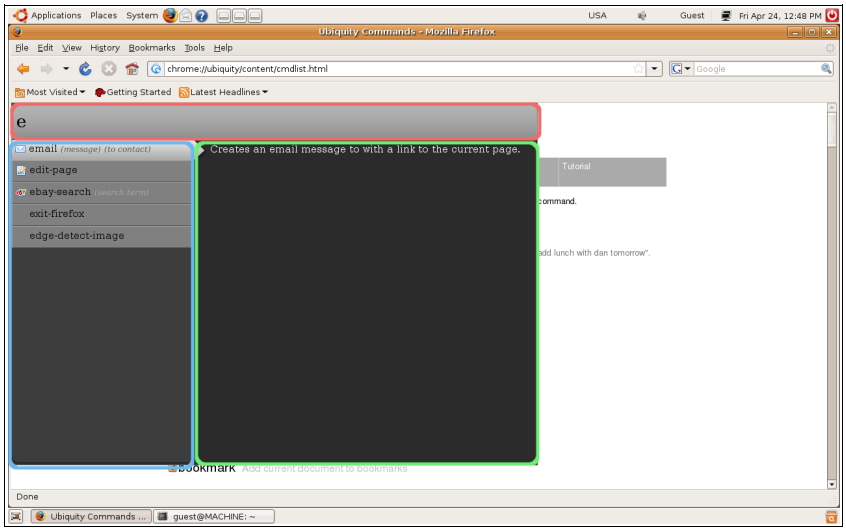

Figure 15. : Ubiquity UI partitions framed

Red-boxed part is just an unlimited length input field. Users type in command fragments here.

Suggested commands appear in the part with blue border. By default system generates suggestions from *command feeds*, but when one creates a custom *noun type* as an argument, then noun type suggestions will appear in this part, too. Suggestion process is pretty complicated and – I don't know why – Ubiquity limits resulted suggestion's number to five. Fortunately I was able to bypass this limit with a hard-coded value.

Commands can have *previews*. These previews show up in the part with green border. Previews are Html documents, so it's possible to insert arbitrary markup, text or image. Content can be rendered with Javascript and of course via Ajax.

I had to use all these opportunities to achieve Ubiquity-like, but server-side command line. I produced a command, called “execute” with 2 arguments. The first is a custom noun type, which is responsible for getting command suggestions from the server side. The second is an ordinary text, which represents the arguments of SOCL commands.

I am going to explain all client-side script in the following pages to demonstrate Ubiquity command development. At first examine the following code, which defines the custom noun type!

```
1.  var noun_type_socl = {
2.    _name: "Service-oriented command line noun",
3.
4.    suggest: function suggestCmd( text, html, callback ) {
5.     if(text.search(" ")==-1){
6.      jQuery.ajax( {
7.        url:"http://localhost:8088/SOCLApp-war/socl/suggest",
8.        dataType: "json",
9.        data:{q :text},
10.       type: "get",
11.       async: true,
12.       success: function suggestCommands( response ) {
13.         var i;
14.         var results = response.result;
15.         for ( i = 0; i < results.length; i++) {
16.           callback( CmdUtils.makeSugg( results[i]) );
17.         }
18.       }
19.     });
20.     return [ CmdUtils.makeSugg('') ];
21.    }
22.  }
23. };
```

My custom noun type implements a suggest function. This function initiates an Ajax call to the web layer component, which is responsible for answering suggest requests. Outgoing data transfer is sent in query string, while incoming data is packaged in JSON. I use CmdUtils.makeSugg API function to add returned values to the internal suggestion repository. I had to place an additional conditional branch in line 5, because after users have chosen a server-side command and started to type in arguments, Ubiquity still sent all arguments to request suggestions. Of course this is not the intended behavior, so I had to prevent this.

Next code snipplet shows “execute” Ubiquity command. Let's have a look at it!

```
1.  var timer = null;
2.  CmdUtils.CreateCommand( {
3.    name: "execute",
4.    takes: { cmd: noun_type_socl},
5.    modifiers: {args: noun_arb_text},
6.    preview: function(pblock, cmd, args){
7.      if (cmd.text!="" || args!=null){
8.        if(timer){
9.           Utils.clearTimeout(timer);
10.       }
11.       timer=Utils.setTimeout(4000,jQuery.ajax({
12.                         url: "http://localhost:8088/SOCLApp-war/socl/describe",
13.                         data: {cmd: cmd.text },
14.                         success: function(data){
15.                          var description = "<ul style=\"{list-style:none;}\"";
16.                          for (var i=0; i<data.response.length; i++){
17.                             description = description + "<li>" +
18.                              data.response[i].key + ": " + data.response[i].value
19.                              + "<li>  Description: " + data.response[i].description
20.                              + "</li></li>";
21.                          }
22.                          pblock.innerHTML = description + "</ul>";
23.                         },
24.                         type: "get",
25.                         dataType: "json",
26.                         cache: false
27.       })
28. );
29.   }
30.   else{
31.      pblock.innerHTML = "Service-oriented command line (SOCL) let users search and execute various
      commands on a remote server. Current release is experimental and part of a thesis project.";
32.   }
33.   }
```

```
34.   ,
35.   execute: function executeCommand( cmd, parameters ) {
36.     if ( parameters ) {
37.       displayMessage("Executing "+cmd.text+" command with the following parameters:
      "+parameters.args.text);
38.       Utils.openUrlInBrowser( "http://localhost:8088/SOCLApp-
      war/socl/execute?"+encodeURIComponent("cmd") + "=" + encodeURIComponent(cmd.text)
      +"&"+encodeURIComponent("args") + "=" + encodeURIComponent(parameters.args.text));
39.     }
40.   }
41. } );
```

CmdUtils.CreateCommand API method is used to create new user commands. My command takes 2 arguments as mentioned before.

I created a preview function (line 6), which sends the command name to the web layer as command description request. Response data then rendered in the preview window. Data transfer schema is the same as before. Code between lines 8 and 10 protects server from unnecessary load; without it every action in suggestion window would generate Ajax calls. This solution only dispatches requests, when users stay on a suggestion for more than 2 seconds effectively eliminating false calls.

Execute function calls command execution web service in the web layer. Response Html will be opened in a new window, or tab depending on the browser settings.

That's all, what is required on the client side! Pretty impressive and short. Now, let's see what happens in the web layer.

Web layer

Web layer consists REST-style web services and stateless session beans. Web services take care of client requests and generate responses, while session beans communicate with CEUs via the messaging system.

REST-style web services just receive suggest, describe and execute requests, transform them to

simple text representation and then pass these data to the appropriate session bean. After a while the services check the acquired session bean, if it got reply from CEU layer. Hopefully they did, so response data is ready for sending back to the client. Web services transform all data into communication form, namely JSON or Html, and respond to the client.

Suggest, describe and execution session beans are the logical client endpoints within SOCL system. Their task is to generate and consume system messages. Meanwhile they convert the message payload between simple text and XML representation. Suggest request messages are sent to the *SuggestRequestTopic*, while description and execution messages to the *CommandDispatcherQueue*.

Messaging system

Not surprisingly messaging system bases on JMS. It handles 2 destinations for incoming messages, temporary queues for reply messages and waiting queues for CEUs.

SuggestRequestTopic is responsible for suggest request message delivery, while CommandDispatcherQueue stores messages while they are forwarded to the recipient. Postman component is implemented as a Message-driven bean. These enterprise beans route messages on the basis of the custom JMS property called namespace. This property contains the command's namespace, which unambiguously designate target waiting queue.

Message communication follows request/reply pattern. Every session bean in the web layer gets a unique and personal *temporary queue*. CEUs send their reply messages to this queue.

All waiting queue belong to one and exactly one CEU. In fact, system administrators create a new waiting queue every time they add a new CEU to the system. Because waiting queues contain describe and execution request messages, message selectors are applied. Thanks to this queue consumers can get just that type of messages they interested in. For further explanation see next section.

Command Executor Unit

Command executor units are the hearth of SOCL systems. They *contain executable commands* and *accompanying information*.

Command repositories include all necessary command descriptors. Abstract command is represented with a distinct object. It stores command name, namespace, all attributes and types for these attributes. An other registry contains command names and corresponding session bean (executor units, see below) JNDI names. Command repository is practically singleton. A dedicated session bean loads command descriptions from an XML document and creates the repository. Since command repository can't change after initialization it can be safely shared between all executor units.

Executor units are session beans, which realize a command. They have to implement an interface called *ExecuteableCommand*, but there aren't any more specific expectation or restriction in connection with these components. They form the business endpoint of SOCL system; any other resource used by them are not necessarily the part of the containing CEU (external web service, etc.).

Message-driven enterprise beans process incoming messages. They are the same in all CEU. There is a specific one for fulfilling suggestion, description, and execution requests. Unified execution request handling is based upon an agreement: every executor unit has to implement ExecutableCommand interface, and provide it's functionality through this interface. In addition, all executor unit generate Html markup as result.

Offering suggestions are particularly interesting, because SOCL system tolerant towards *light spelling errors*. Light spelling error can be an extra character between two characters (like 'c' in 'ccommand'), transposition of two, immediately following characters (like 'c' and 'o' in 'ocmmand') and combination of this errors (like 'ccommadn'). I implemented a metrics, which can compute the difference between two character strings taking into account that light spelling errors can be presented. Using this SOCL system can offer good suggestions, even when the user input is erroneous. Tolerance limit can be configured. The following function realizes this metrics.

```
1.  private static int computeDiversibility(String str1, String str2){
2.          String s1, s2;
3.          int diversibility;
4.          if(str2.length()>str1.length()){
5.                  s1=str2.toLowerCase();
6.                  s2=str1.toLowerCase();
7.              }
8.           else{
9.              s1=str1.toLowerCase();
10.             s2=str2.toLowerCase();
11.          }
12.         if (s1.contains(s2)){
13.             return 0;
14.         }
15.         else{
16.              diversibility=0;
17.                  int offset=0;
18.                  for (int i=0; i<s1.length(); i++){
19.                    if (i-offset>=0 && i-offset<s2.length()
20.                                    && s1.charAt(i)==s2.charAt(i-offset)){
21.                          }
22.                          else{
23.                   if(i+1<s1.length() && i-offset>=0
24.                                    && i-offset+1<s2.length()
25.                                    && (s1.charAt(i+1)==s2.charAt(i-offset) && s1.charAt(i)==s2.charAt(i-offset+1))) {
26.                              }
27.                              else{
28.                                 if(i-1>=0 && i-offset-1>=0
29.                                          && i-offset<s2.length()
30.                                    && (s1.charAt(i)==s2.charAt(i-offset-1) && s1.charAt(i-1)==s2.charAt(i-offset))) {
31.
32.                                   }
33.                                       else{
34.                                         if(i+1<s1.length() && i-offset>=0 && i-offset<s2.length()
```

```
35.                                         && s1.charAt(i+1)==s2.charAt(i-offset)) {
36.                                             offset++;
37.                                     }
38.                                    else{
39.                                     diversibility++;
40.                                     offset++;
41.                                     }
42.                                   }
43.                                }
44.                             }
45.                          }
46.          }
47.           return diversibility;
48.  }
```

Command repository applies this function instead of traditional string comparison, when searches for matching commands. The following screenshots show this method in action.

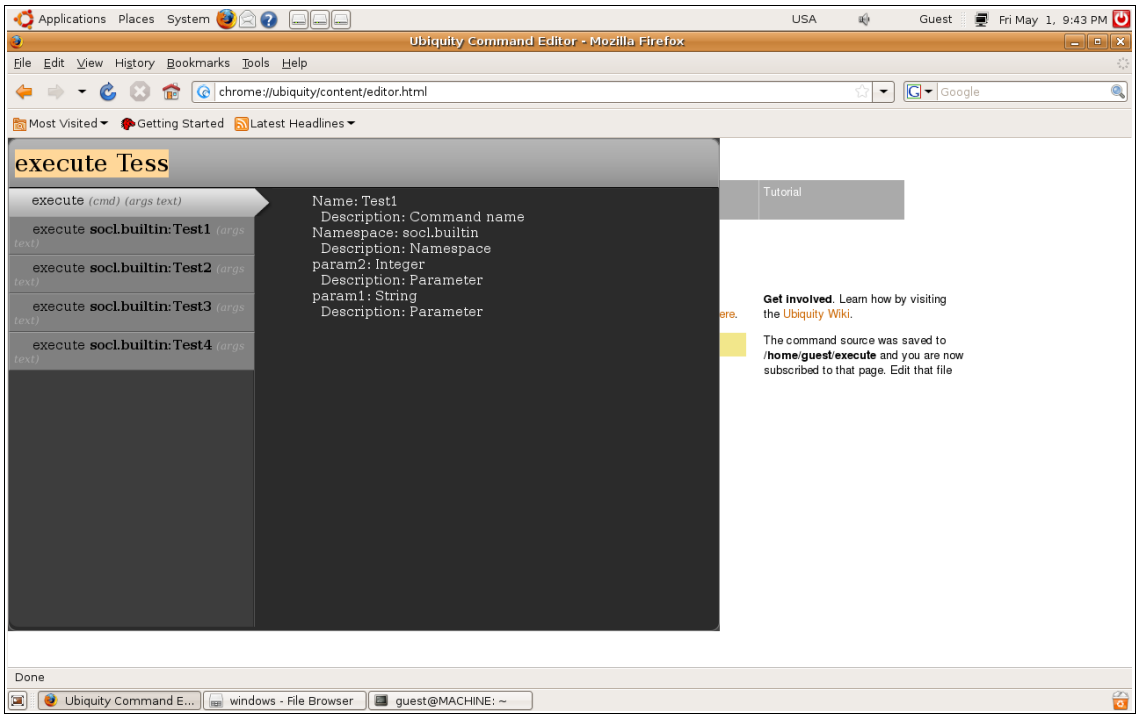

Figure 16. : Extra character error

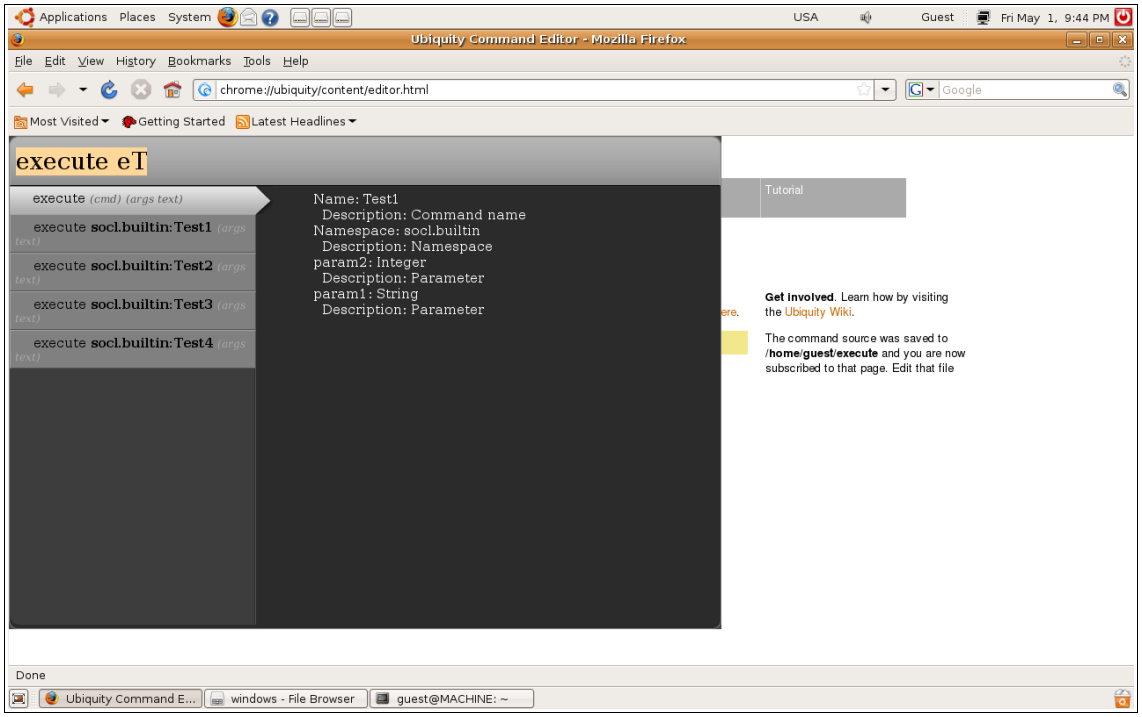

Figure 17. : Character transposition error

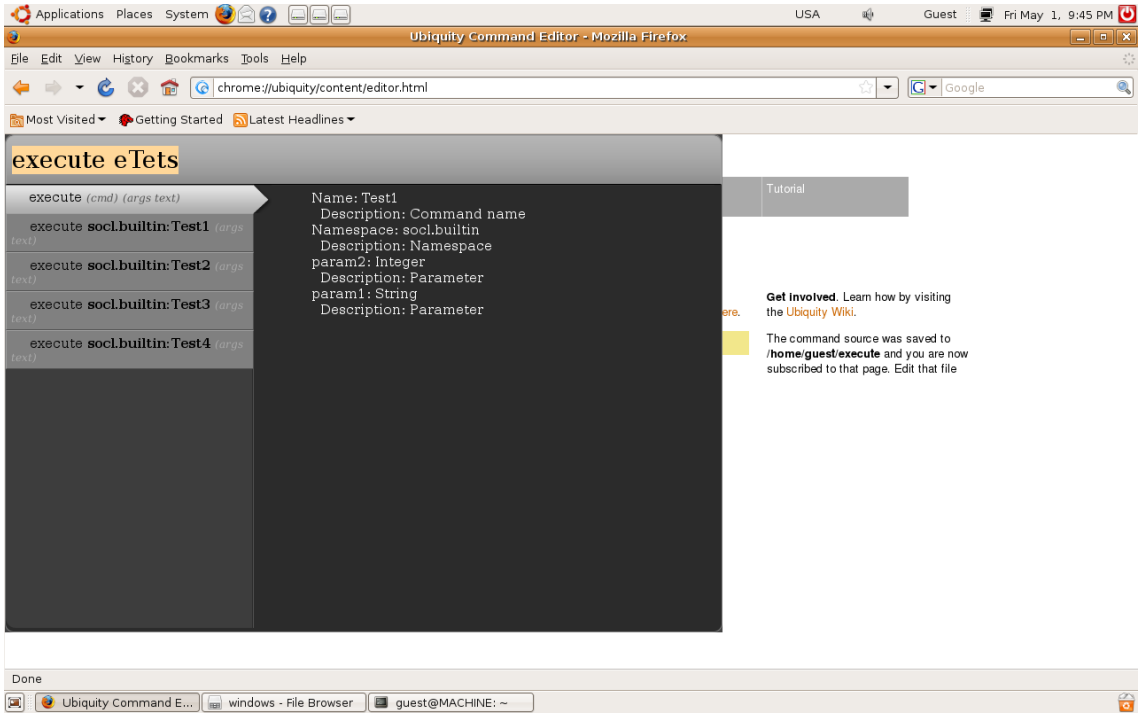

Figure 18. : Mixed error

Screenshots and comments

I created all the key parts of SOCL system. Screenshots visualizing main features and the generic scenario can be seen below.

I consider my thesis as a successful project. However, I had to face with serious problems, and solutions are sometimes just like symptomatic treatments. Current Ubiquity is a beta version, this means that documentation is incomplete and many times system behaves in an unexpected manner.

For example command suggestion mechanics is not clear. I mentioned before that Ubiquity always reforms it's internal suggestion repository, when users type in new arguments. This means that it always calls suggestion function for all the noun types used as arguments, if they have it. Unfortunately this server call will fail, because the actual string sent to the server is the command fragment *and* other arguments in one piece. In addition, when the web layer doesn't send a right answer back, Ubiquity refreshes suggestions, but with meaningless values. (See Figure 24.)

Other shortcoming is the lack of Ajax support. Officially the API has functions to encapsulate Ajax calls, but they doesn't seem to work. I had to implement my own solution, for instance to prevent server overload, when the user browses suggestions. Ubiquity generated Ajax calls every time without this, even when I used *previewAjax* API function, which task is to handle exactly the same problem.

Moreover I didn't find any method to open the response Html in the same tab, where Ubiquity has been invoked. The *Utils.openUrlInBrowser* function can open it in a new tab, but why does this simple functionality missing? Albeit Ubiquity is far from perfect at the time of writing, it's still a great piece of software, which can completely change the way we think about the web!

I also had to solve lesser incompatibility issues on the server side, but Java is a great language with great community behind, so fixing them was not a big deal. Now examine the following pictures, showing SOCL system in action!

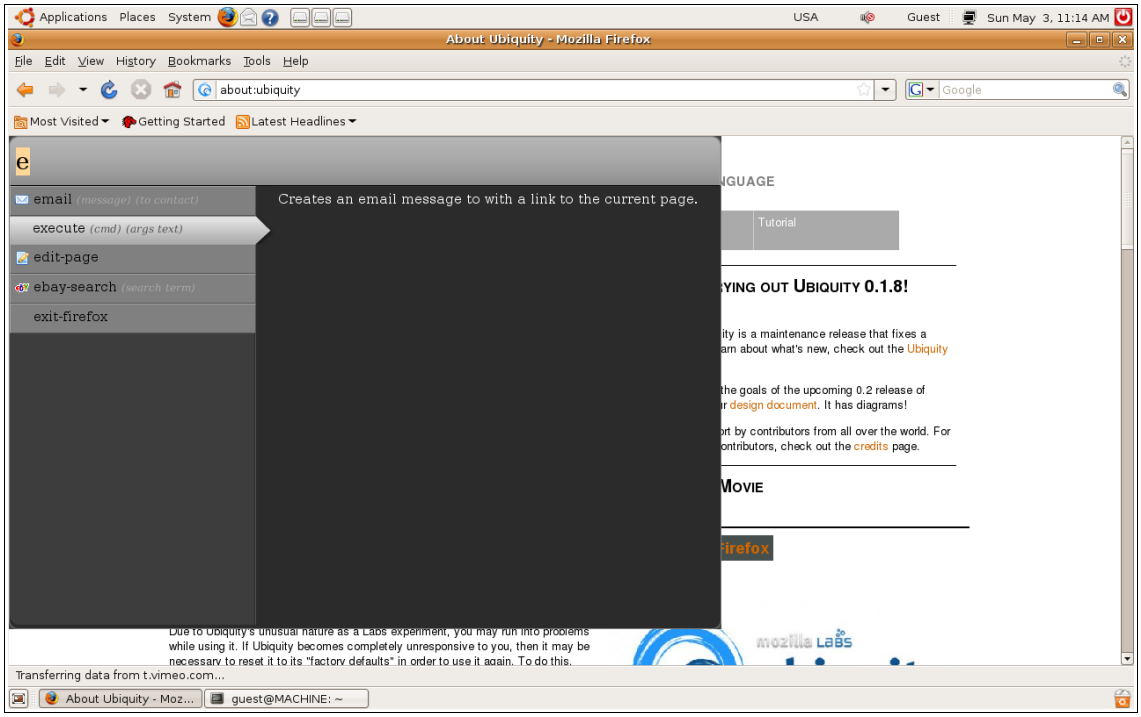

Figure 19. : Ubiquity invoked, "execute" command installed

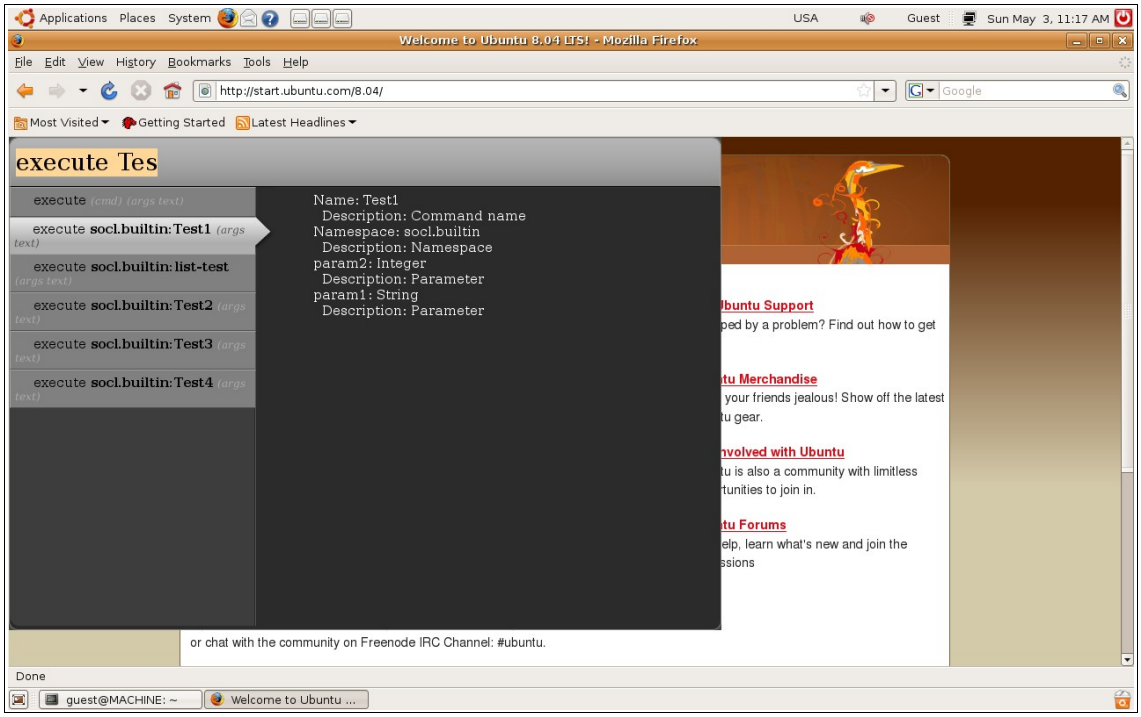

Figure 20. : Searching for commands like "Tes".

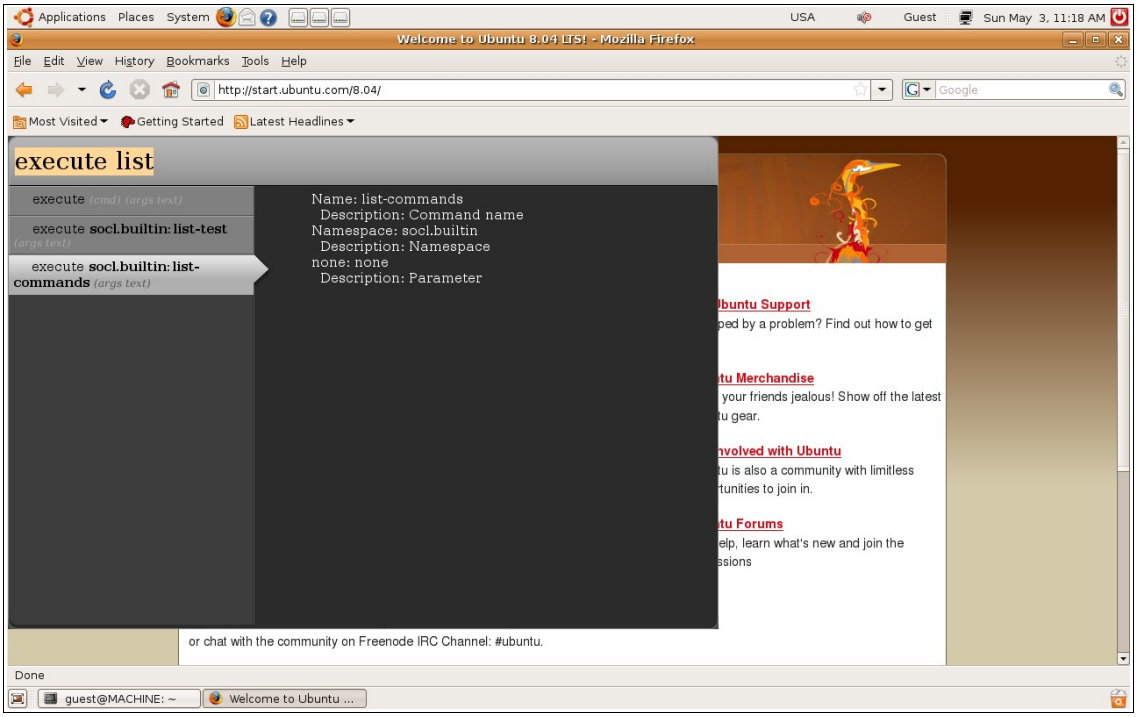

Figure 21. : Searching for commands like "list".

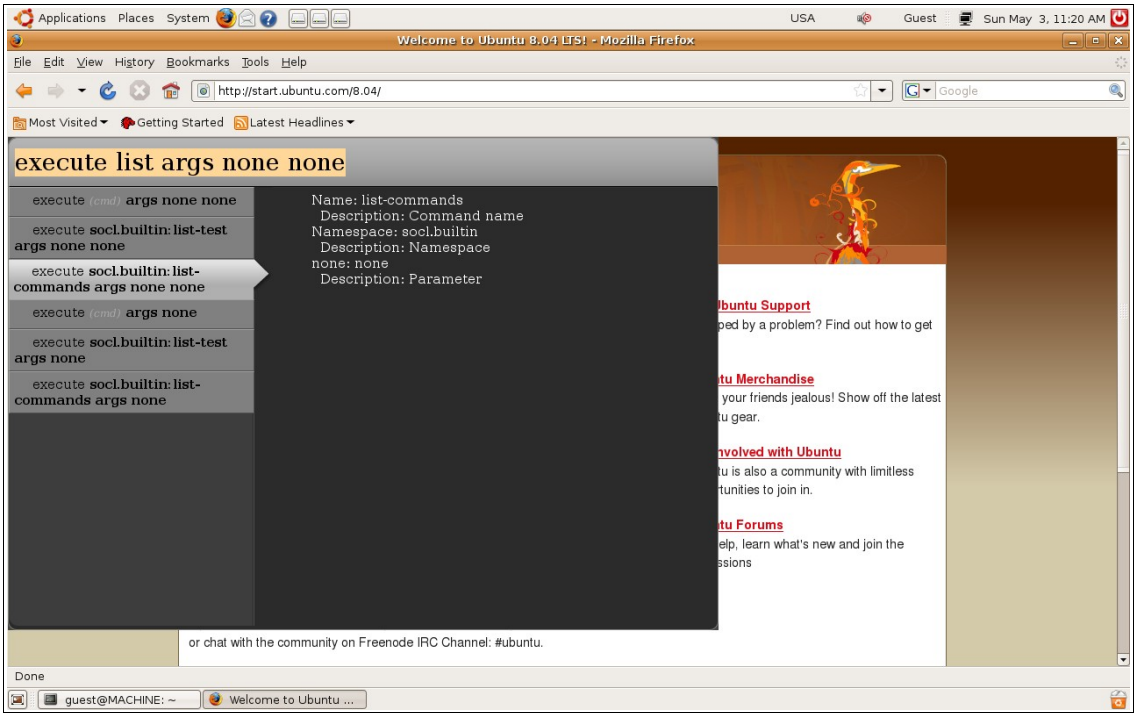

Figure 22. : Arguments typed in.

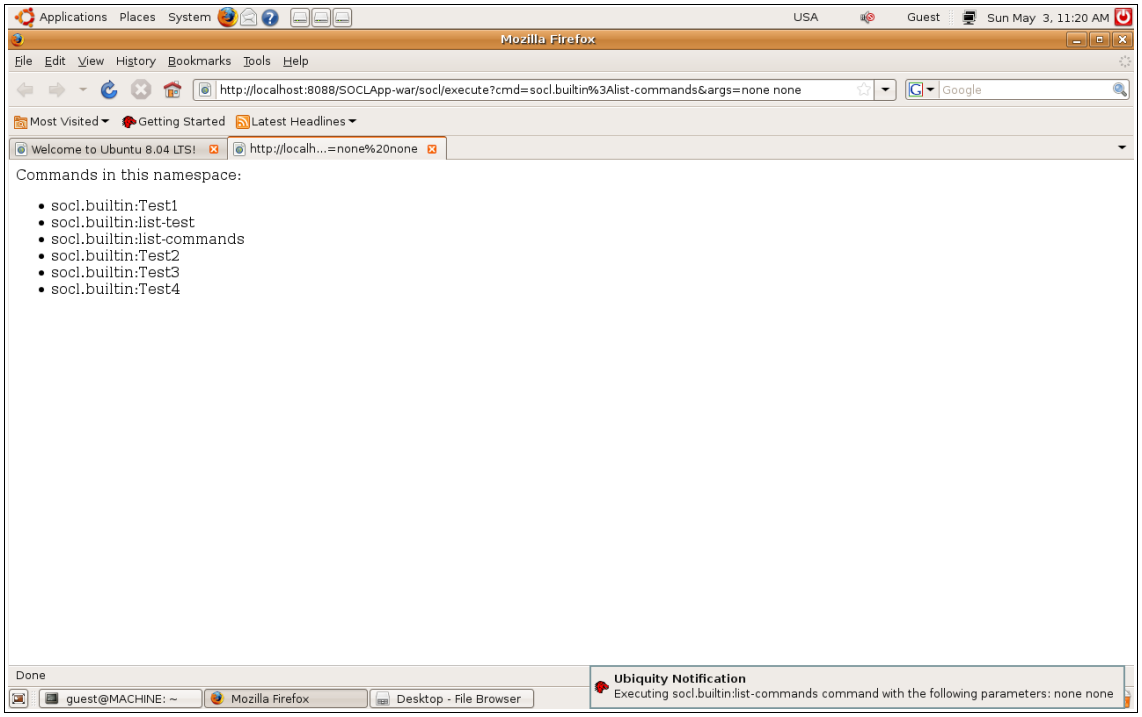

Figure 23. : list-commands invoked, which list all available commands.

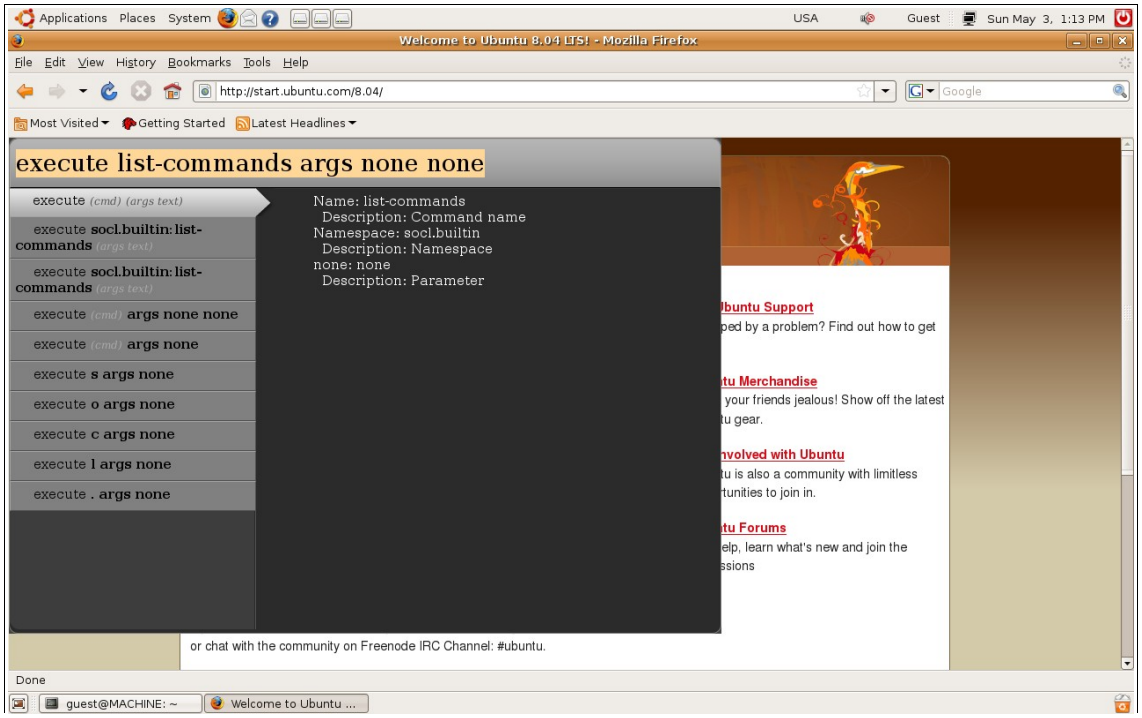

Figure 24. : Suggestion behavior without fix

Conclusion

I proved in my thesis that reimplementing command line and fitting in to the modern web environment is achievable and what is more important, useful. Unfortunately I didn't have enough time to create a more advanced version, but developing arbitrary enterprise application behind executor units is possible. They can serve a content management system, Web APIs, existing enterprise or web application, legacy systems, etc. . Potential developers just need to create executor units and alter command repository XML to develop a brand new CEU. System administrators can adopt CEUs with installing them into the application server and creating a new waiting queue for each CEU. I believe that both of them are really straightforward processes, which doesn't force programmers to get acquainted with large APIs, thus making both development and administration easier. SOCL system is a pilot project in the current stage. Further extensions can even improve the the software's capabilities.

Creating a pure Javascript/Ajax client-side could resolve browser dependency. It could complement other web applications and offer a faster way to reach functionality within application or it could serve as a basis for dynamic mashup generators and interactive systems.

Applying more SOA principles and tools could refine base architecture and improve stability, scalability and higher availability. Other way could be transforming the system into a web application to allow the new, lightweight system to be placed in less powerful computing environments.

However, every interactive software's popularity depend on the available commands, thereby on the people using it. If the community is strong, then they develop more and more useful components, which makes the full system better. But if the potential users loose their interest, the software slowly gets outdated, which surely indicates the project's end. *People*, who use the services have the real power in the world of Web 2.0 . I hope that this is the way, which leads to better user interfaces and brighter future, at least within information technology.

Bibliography

1. Java EE main page:
   http://java.sun.com/javaee/
2. Java EE tutorial:
   http://java.sun.com/javaee/5/docs/tutorial/doc/
3. Glassfish:
   https://glassfish.dev.java.net/
4. Ubiquity main page:
   http://labs.mozilla.com/2008/08/introducing-ubiquity/
5. Ubiquity Author tutorial:
   ttps://wiki.mozilla.org/Labs/Ubiquity/Ubiquity_0.1_Author_Tutorial
6. JQuery:
   http://jquery.com/
7. Manning: jQuery in Action (Bear Bibeult , Yehuda Katz)
8. Web service definition:
   http://www.w3.org/TR/ws-arch/
9. Quicksilver:
   http://docs.blacktree.com/quicksilver/what_is_quicksilver
10. Humanized.com (Enso):
    http://www.humanized.com/enso/
11. GnomeDo:
    http://do.davebsd.com/wiki/index.php?title=Main_Page
12. Many articles from http://wikipedia.org