FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

75.42 – Taller de Programación I

Grupo 2

2do Cuatrimestre 2016

Self Interpreter

Documentación Técnica

Índice

Requerimientos de software

Como requisitos de software para poder compilar, desarrollar y depurar el programa es necesario contar con:

OS:

- Alguna distribución de linux que soporte GTK+3 (Se recomienda Debian sid)

Bibliotecas:

- GTK+ 3.22
- gtkmm versión 3.22
- glibmm versión 2.50
- glib versión 2.50.2
- GNU make >= 3.81
- GNU g++ >= 4.8.2

Herramientas de desarrollo:

- Eclipse
- glade

Herramientas de depuración:

- valgrind
- gdb

Descripción general

El proyecto consiste en dos aplicativos. Un aplicativo servidor, y uno cliente. Los mismos son los dos grandes módulos del proyecto.

El servidor tiene dos modalidades. La primera es levantar un archivo con código self de forma local y ejecutarlo. Es decir, en este modo el aplicativo no actúa de servidor. El otro modo consiste en levantar un servidor y esperar conexiones de clientes con peticiones para trabajar sobre workspaces.

Para más información sobre las modalidades del servidor ver **Manual de Usuario - Formas de uso**.

El aplicativo del cliente permite al usuario interactuar con el sistema a través de una interfáz gráfica. A través de la misma el cliente envía peticiones al servidor para trabajar sobre los distintos workspaces.

A continuación se desarrolla cada módulo en detalle.

Módulo Servidor

Descripción general

Este módulo representa al servidor.
El servidor se puede ejecutar en dos modos. Uno para ejecutar scripts locales con código self y otro para levantar un servidor propiamente dicho. (ver **Manual de Usuario - Formas de uso** para más detalles).

Tenemos una clase ModeSelector que se encarga de derivar la consulta según el tipo de modo. En el modo de script local llama directamente al workspaces para ejecutar el script. En el modo servidor se crea un server y un accepter en otro hilo que irá aceptando las conexiones de los clientes (representadas por el proxyClient) en nuevos hilos.

Los ProxyClients (cada uno en su respectivo hilo) le hacen peticiones al servidor que llegan desde el cliente. El servidor es el encargado de asegurrar la correcta concurrencia de hilos protegiendo los recursos compartidos.

Los recursos compartidos son la lista de workspaces y los workspaces en si mismos con todos sus objetos y la máquina virtual de cada workspace.

El servidor deriva las consultas de los ProxyClients al workspaces que el ProxyClient le diga (los ProxyClientes tienen identificadores de los objetos que fueron viendo y el workspaces sobre el que el cliente se encuentra trabajando). Una vez obtenido el objeto que se debe devolver al cliente el servidor le pide al ParserProtocoloServidor que genere una cadena para devolversela al ProxyClient y que el mismo la re-envíe por la red al cliente (a través de la clase Socket).

El workspaces cuando es creado instancia un objeto lobby y una máquina virtual.
La máquina virtual es la encargada de crear todos los objetos self que se vayan solicitando y le pasara la información al objeto lobby que apila la lista para luego poder hacer la recolección de objetos.

La clase object es la encargada de encapsular a los objetos self.

La clase Parser es llamada por el workspace ejecutar scripts. A medida que analiza el contenido del script el Parser le pide a la Máquina virtual que instancie los objetos según el tipo deseado. El parser se analiza a detalle a continuación.

Parser

Introducción

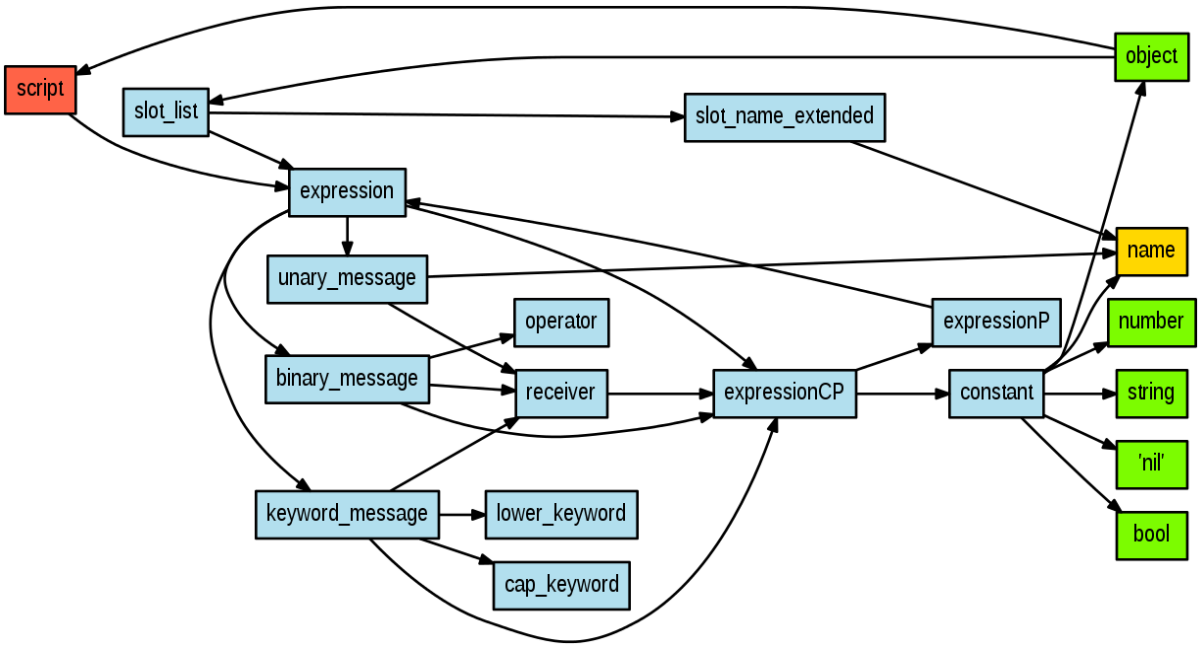
El *parser* recibe tres argumentos, una cadena con código self, un objeto contexto y una referencia a la máquina virtual.

¿Qué es el contexto?

El *parser* ejecuta código de forma relativa al objeto en el que se encuentre posicionado. Ese objeto es el contexto. El contexto influye en la cadena de *lookup* que se utilice para buscar los *slots*.

La máquina virtual se utiliza para realizar las instanciaciones de objetos, ya que la misma es la encargada de manejar la pila de objetos del workspace.

El *parser* realiza un análisis secuencial *char* a *char* en el que va tratando de predecir el tipo de código que se está interpretando.



El diagrama es una representación del parser. No se encuentran representadas todas las funciones con el fin de mejorar la visualización y el entendimiento del mismo.

El *parser* comienza su ejecución en la función *script()* (como podemos ver en la celda roja en la imagen). Luego irá atravesando una a una las diferentes funciones tratando de interpretar el código self. Cuando se tope con incongruencias de código (es decir, que falló su predicción actual) realizará un *rollback* hasta el último *checkpoint* válido. El *checkpoint* guarda la última posición válida de la cadena analizada.

Los objetos se instancian cuando se define que se encontró un objeto primitivo o un objeto complejo. Se puede visualizar esta situación en las celdas verdes de la imagen. El objeto

complejo es un caso particular ya que primero se crea un objeto vacío y luego se irán cargando uno a uno los *slots* que se vayan detectando.

Por otro lado, cuando se llegue al *'name'* lo que se hará es un *lookup* y se devolverá un puntero de un objeto preexistente. Se puede visualizar esta situación con la celda amarilla.

El parser posee un atributo llamado *flagExecute* que determinan cuando se realiza ejecución de código. Cada vez que se ingrese a un posible script se incrementa el *flagExecute* en uno. Cuando el *flagExecute* es igual a 1 se ejecuta código (es decir se llama al *recvMessage* del objeto). Esto nos asegura de no ejecutar código cuando estamos instanciando objetos con bloques de código.

La método más determinante del parser es *receiveMessage(..)* . El mismo es la encargado de determinar qué objetos deben ser clonados antes de realizar el recvMessage de objeto.

Problemas actuales del parser

- No se soportan *keyword messages* de más de un argumento.
- Los *strings* no soportan comillas simples dentro de su cadena.
- Actualmente sólo se soportan números enteros.

Clases

A continuación se listan las clases relacionadas al módulo con una breve descripción de las mismas. Para ver en profundidad el funcionamiento y los métodos que implementan dirigirse al Documento Anexo **Documentación de clases**.

Thread

Encapsula los métodos para iniciar, correr y joinear hilos.

Proxy

Esta clase es la base para las clases *ProxyClient* y *ProxyServer*.
Contiene los métodos y atributos comunes a ambos.

ProxyClient

El *ProxyClient* es el encargado de responder las peticiones del cliente (*ProxyServer*). Para resolver las peticiones delega las consultas al servidor(Server/Modelo de Negocio).
El *ProxyClient* solo conoce los IDs del *Workspace* y de los objetos con los que trabaja.

Socket

Representa una encapsulación de los sockets provistos por el sistema operativo que estan en las librerías de Unix.

Accepter

Es el encargado de aceptar nuevos clientes abriendo *proxies* en nuevos hilos. Un hilo por cada cliente conectado.

ModeSelector
Es la clase selectora para los dos modos del servidor.
El modo servidor propiamente dicho y el modo para levantar archivos locales con código self.

Server
Representa el modelo de negocio. Resuelve las peticiones de los *ProxyClient*'s y administra los recursos que se deben proteger.

Workspace
Representa un ambiente de trabajo (*workspace*) y es el encargado de crear el *lobby* y de llamar al *parser* para ejecutar los scripts de código self.

VirtualMachine
Es la encargada de crear y almacenarlos en una pila en el objeto *lobby*.
Tanto la máquina virtual como el objeto *lobby* son únicos por *workspace*.

Object
Representa un objeto del lenguaje Self.

Parser
Es la clase encargada de parsear scripts en código self y de indicarle a la máquina virtual (VM) qué objetos y mensajes se deben emitir en consecuencia.

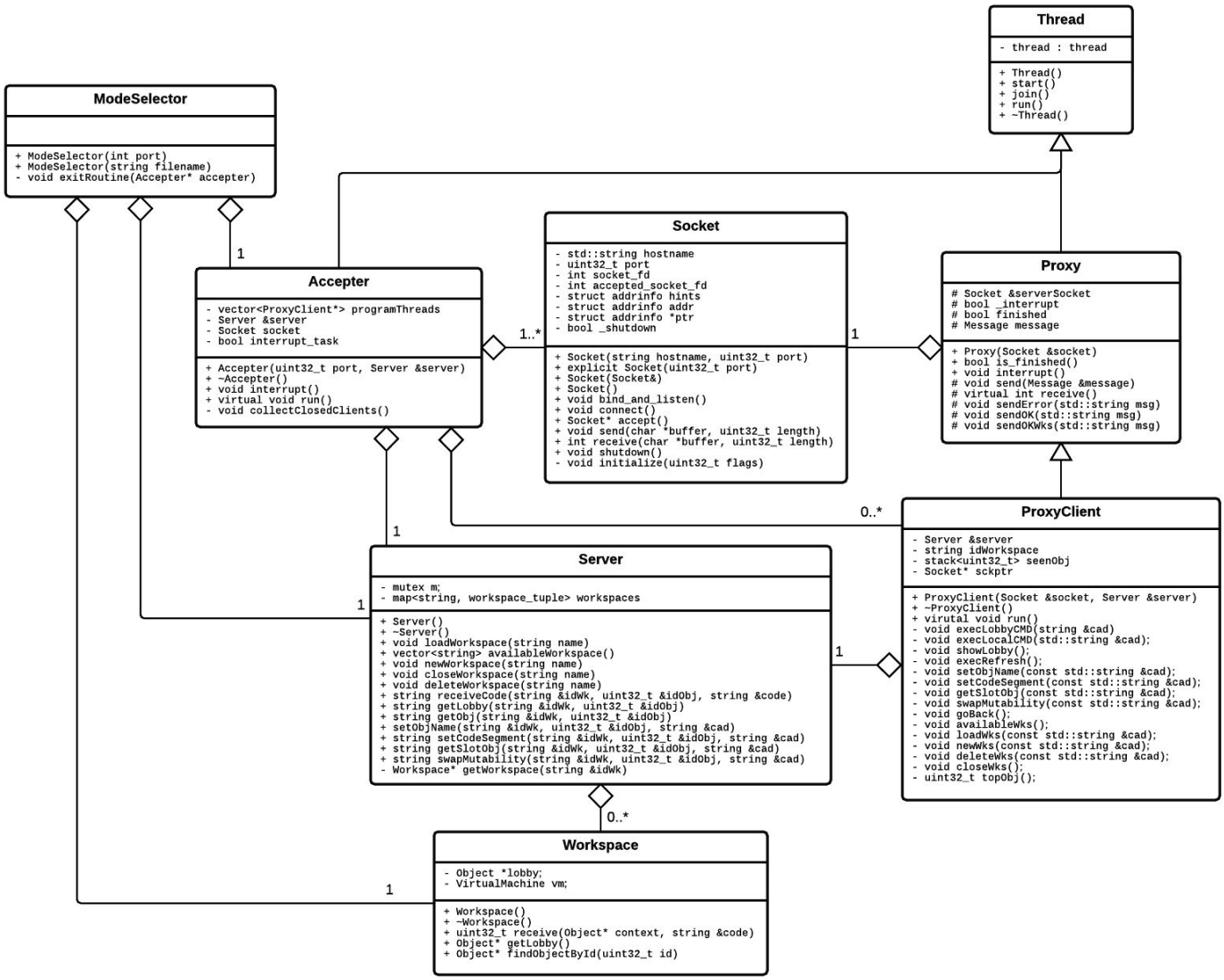
ParserProtocoloServidor
Esta clase se encarga de generar una cadena con el formato especificado por protocolo en función de un objeto dado por el servidor para que la misma sea enviada por el *ProxyClient* al cliente.

Diagramas UML

Diagrama de clases

Se dividió el diagrama en dos partes para obtener una mejor visualización.
Las clases compartidas son Server y Workspace.

Parte 1



Parte 2

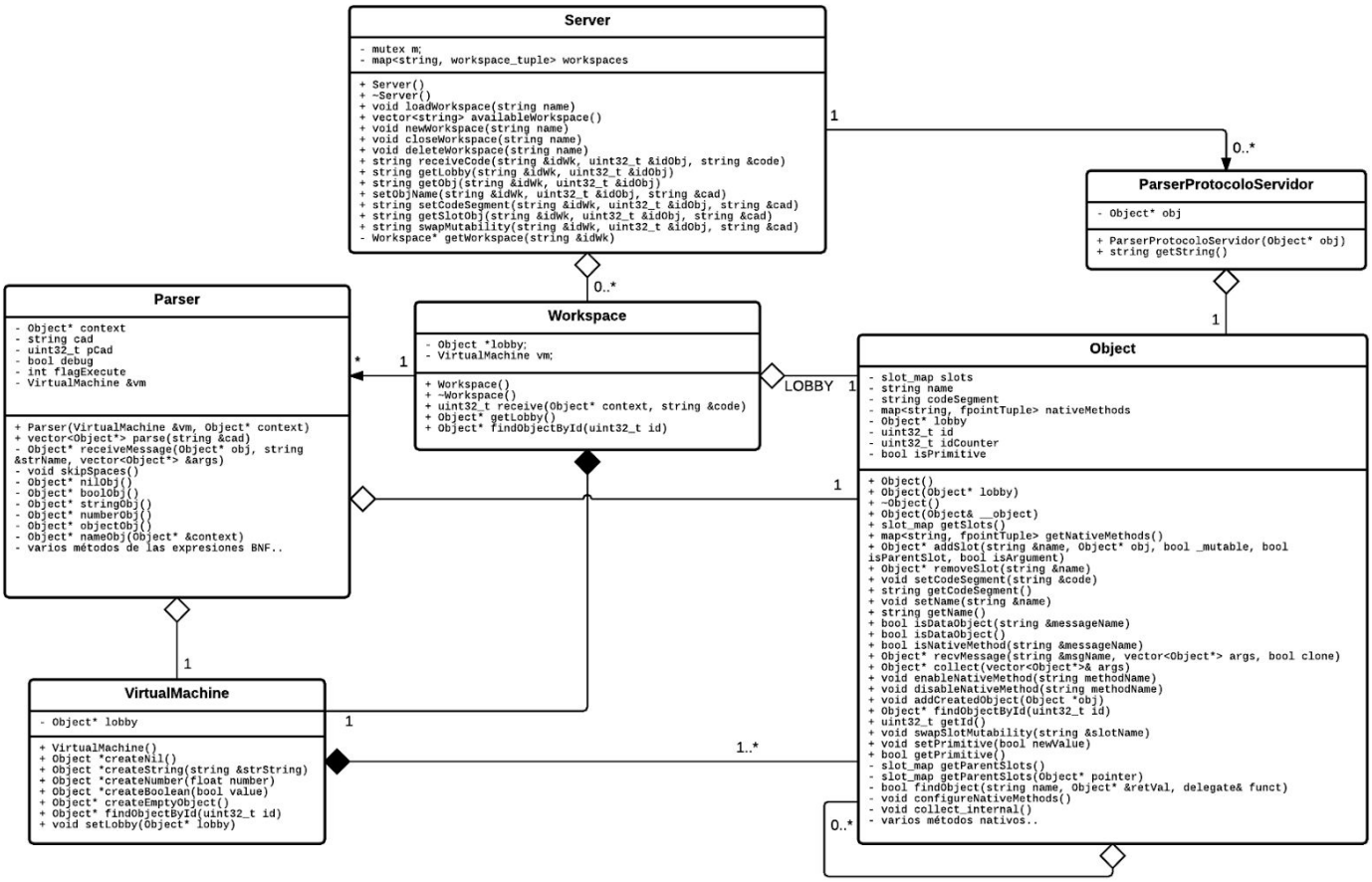
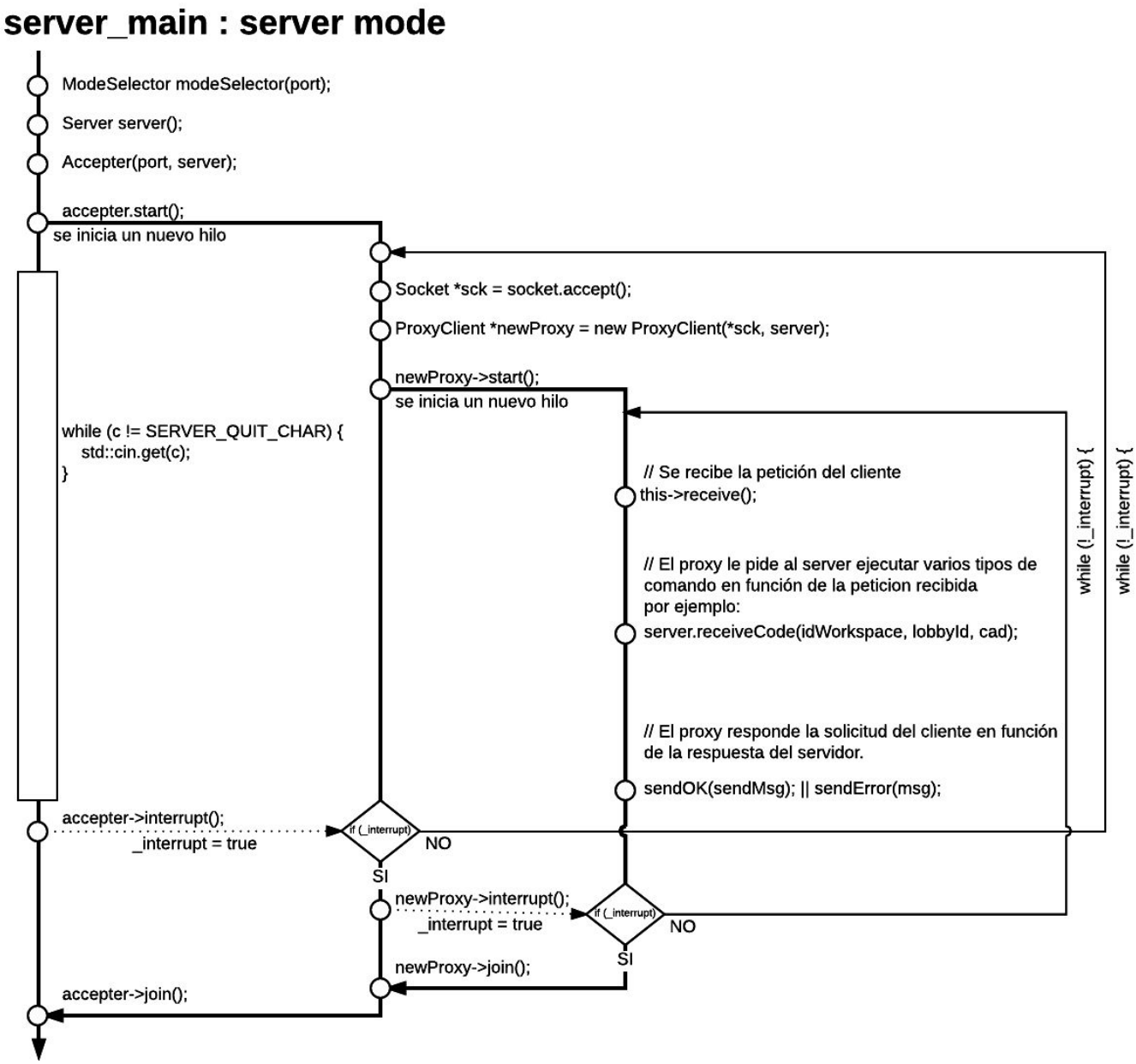

Server
- mutex m;
- map<string, workspace_tuple> workspaces
+ Server()
+ ~Server()
+ void loadWorkspace(string name)
+ vector<string> availableWorkspace()
+ void newWorkspace(string name)
+ void closeWorkspace(string name)
+ void deleteWorkspace(string name)
+ string receiveCode(string &idWk, uint32_t &idObj, string &code)
+ string getLobby(string &idWk, uint32_t &idObj)
+ string getObj(string &idWk, uint32_t &idObj)
+ setObjName(string &idWk, uint32_t &idObj, string &cad)
+ string setCodeSegment(string &idWk, uint32_t &idObj, string &cad)
+ string getSlotObj(string &idWk, uint32_t &idObj, string &cad)
+ string swapMutability(string &idWk, uint32_t &idObj, string &cad)
- Workspace* getWorkspace(string &idWk)
1
0..*
ParserProtocoloServidor
- Object* obj
+ ParserProtocoloServidor(Object* obj)
+ string getString()
1
Parser
- Object* context
- string cad
- uint32_t pCad
- bool debug
- int flagExecute
- VirtualMachine &vm
+ Parser(VirtualMachine &vm, Object* context)
+ vector<Object*> parse(string &cad)
- Object* receiveMessage(Object* obj, string &strName, vector<Object*> &args)
- void skipSpaces()
- Object* nilObj()
- Object* boolObj()
- Object* stringObj()
- Object* numberObj()
- Object* objectObj()
- Object* nameObj(Object* &context)
- varios métodos de las expresiones BNF..
*
1
Workspace
- Object *lobby;
- VirtualMachine vm;
+ Workspace()
+ ~Workspace()
+ uint32_t receive(Object* context, string &code)
+ Object* getLobby()
+ Object* findObjectById(uint32_t id)
LOBBY
1
Object
- slot_map slots
- string name
- string codeSegment
- map<string, fpointTuple> nativeMethods
- Object* lobby
- uint32_t id
- uint32_t idCounter
- bool isPrimitive
+ Object()
+ Object(Object* lobby)
+ ~Object()
+ Object(Object& __object)
+ slot_map getSlots()
+ map<string, fpointTuple> getNativeMethods()
+ Object* addSlot(string &name, Object* obj, bool _mutable, bool isParentSlot, bool isArgument)
+ Object* removeSlot(string &name)
+ void setCodeSegment(string &code)
+ string getCodeSegment()
+ void setName(string &name)
+ string getName()
+ bool isDataObject(string &messageName)
+ bool isDataObject()
+ bool isNativeMethod(string &messageName)
+ Object* recvMessage(string &msgName, vector<Object*> args, bool clone)
+ Object* collect(vector<Object*>& args)
+ void enableNativeMethod(string methodName)
+ void disableNativeMethod(string methodName)
+ void addCreatedObject(Object *obj)
+ Object* findObjectById(uint32_t id)
+ uint32_t getId()
+ void swapSlotMutability(string &slotName)
+ void setPrimitive(bool newValue)
+ bool getPrimitive()
- slot_map getParentSlots()
- slot_map getParentSlots(Object* pointer)
- bool findObject(string name, Object* &retVal, delegate& funct)
- void configureNativeMethods()
- void collect_internal()
- varios métodos nativos..
1
1
VirtualMachine
- Object* lobby
+ VirtualMachine()
+ Object *createNil()
+ Object *createString(string &strString)
+ Object *createNumber(float number)
+ Object *createBoolean(bool value)
+ Object* createEmptyObject()
+ Object* findObjectById(uint32_t id)
+ void setLobby(Object* lobby)
1
1..*
0..*

Diagrama de secuencia para representar los hilos en modo Servidor



Módulo Cliente

Descripción general

El cliente, como es un aplicativo destinado al usuario final posee una interfáz gráfica.
Cuando se inicia el aplicativo se debe indicar a que servidor y puerto se desea conectar.
En el hilo principal corre la GUI, que se compone de dos ventanas modales. Una selectora de workspaces y otra que es la Morph View. (ver detalles de las ventanas en Manual de Usuario - Formas de uso).

Se abrirá en un hilo a parte un ProxyServer que será el encargado de enviarles las peticiones (generadas desde la GUI) al servidor.

A su vez se creara un Morph y una lista de workspaces, el primero que representa la información que debe dibujar la ventana Morph view y el segundo la información que debe dibujar la ventana selectora de workspaces.

Para coordinar cuando se están esperando respuestas del servidor y cuando la GUI está apta para generar nuevas consultas el ProxyServer posee un flag.

Cuando la fui genera una consulta, setea el comando que desea enviar y el mensaje del mismo. Al mismo tiempo setea el flag del ProxyServe (en modo esperando respuesta del servidor). Mientras se esperen respuestas del servidor la GUI no podrá generar nuevas peticiones para que el ProxyServer le envie al servidor. Como se encuentran en hilos separados, para la GUI es transparente, ya que la interfáz no se laguea durante la espera de la respuesta.

Cuando llega la respuesta del servidor, según en que ventana se encuentre se llamará al ParserProtocoloMorph o al ParserProtocoloWorkspaces para que actualicen la información del objeto Morph y la lista de nombres de los workspaces respectivamente según sea el caso

Los recursos compartidos que deben ser protegidos en el aplicativo cliente son el flag del ProxyServer, el objeto Morph (único) y la lista de nombres de workspaces. Los mismos deben ser protegidos ya que son editados o leidos desde ambos hilos (el principal con la GUI y el del ProxyServer).

Clases

A continuación se listan las clases relacionadas al módulo con una breve descripción de las mismas. Para ver en profundidad el funcionamiento y los métodos que implementan dirigirse al Documento Anexo **Documentación de clases**.

Thread
Encapsula los métodos para iniciar, correr y joinear hilos.

Proxy
Esta clase es la base para las clases *ProxyClient* y *ProxyServer*.
Contiene los métodos y atributos comunes a ambos.

ProxyServer
Es la encargada de enviar las peticiones generadas desde la GUI al servidor.

Socket
Representa una encapsulación de los *sockets* provistos por el sistema operativo que estan en las librerías de Unix.

Morph
Este objeto contiene la información que ve el cliente desde la GUI y representa al objeto que está visualizando.

MorphWindow
Se encarga de dibujar la ventana que representa al *Morphic* de Self
Permite visualizar la información de un objeto de por vez.

SelectWkWindow
Se encarga de dibujar la ventana selectora de *workspaces*.

ParserProtocoloMorph
Esta clase se encarga de parsear los mensajes que lleguen al cliente con el comando OK_MSG_MORPH y cargar el objeto *Morph* con la información obtenida.

ParserProtocoloWorkspaces
Esta clase se encarga de parsear los mensajes que lleguen al cliente con el comando OK_MSG_SELECT_WKS y cargar el vector de *workspaces* con la información obtenida.

Diagramas UML

Diagrama de clases

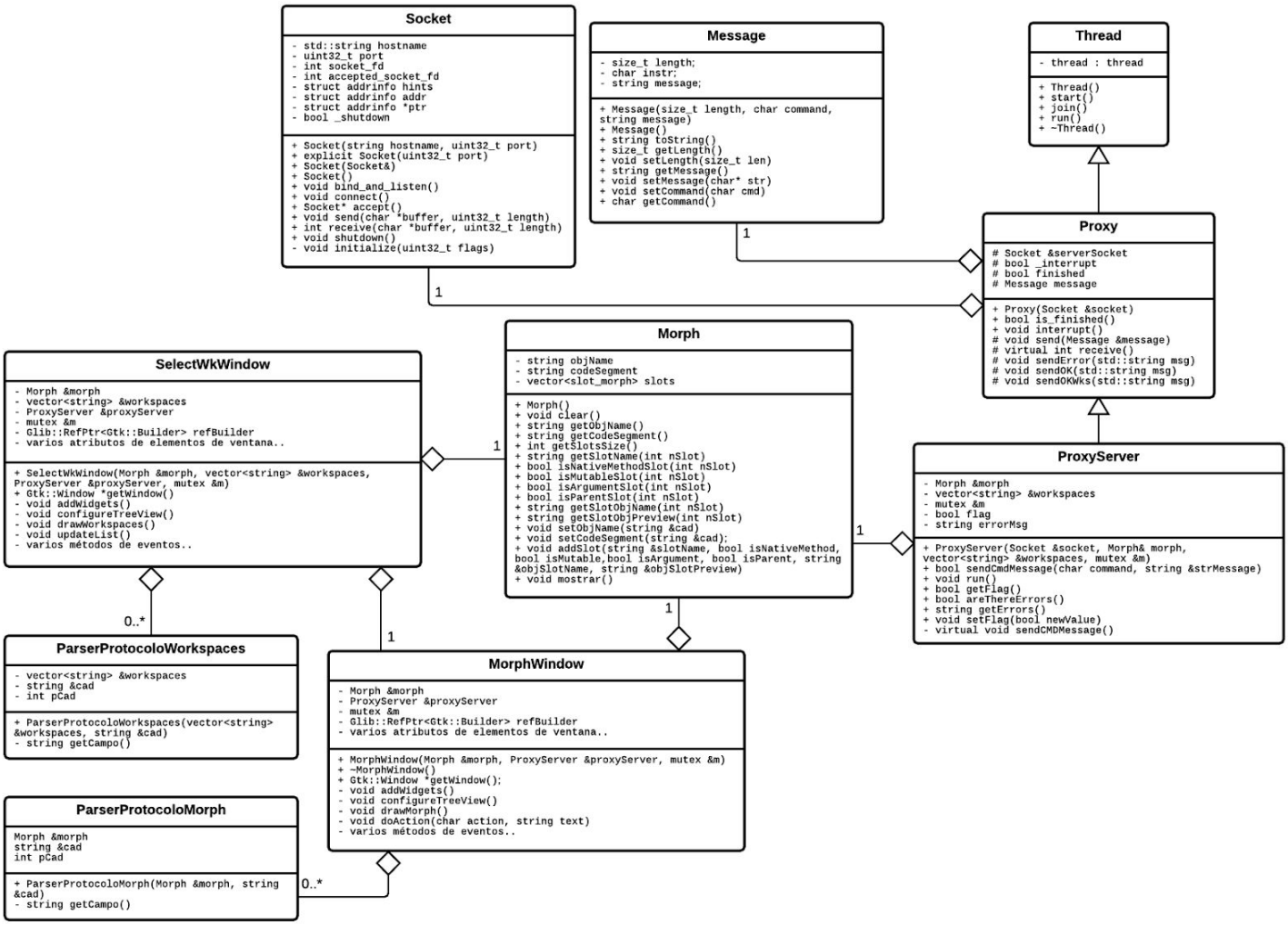
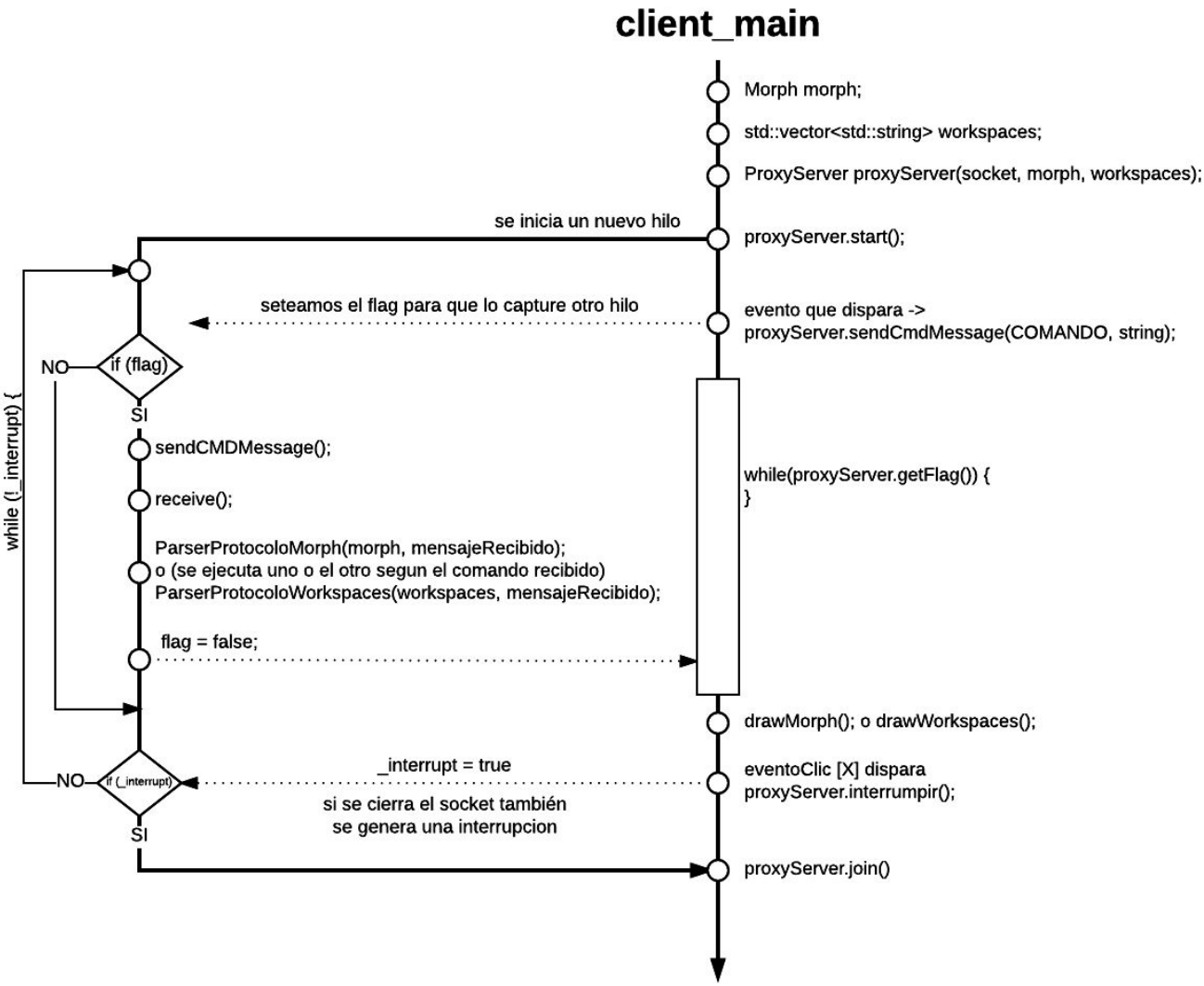


Diagrama de secuencia para representar los hilos



Protocolo

Se estableció un protocolo para definir de qué forma se comunican el cliente y el servidor.
Las cadenas de *bytes* enviadas (aplica tanto para cliente como para el servidor) se componen de 3 campos:

- **Campo 1:** La longitud del mensaje (*length*) en formato binario con un tamaño *size_t*.
- **Campo 2:** Un comando en formato binario de tamaño *char*.
- **Campo 3:** Un mensaje asociado al comando en formato *char**.

Se definió un caracter especial '~' que no debe ser utilizado en el código self ni en los nombres de los *workspaces*. Se escapea el caracter en ambos casos, por lo que su uso indebido disparara una excepción advirtiendo la situación.

Servidor

Envia:	**Campo1**	**Campo2**	**Campo3**
ErrorMessage	size_t(lenght)	0x50	strError
OKMsgMorph	size_t(lenght)	0x00	strWksList
OKMsgSelectWks	size_t(lenght)	0x01	strObjView

strWksList =
[<nombreWorkspace>]{'~'<nombreWorkspace>}

strObjView =
<nombreObjeto>'~'<codeSegment>
{'~'<nombreSlot>'~'<isNativMethod>'~'<isMutable>'~'<isParent>'~'<isArgument>'~'<nombreObjSlot>'~'<previewObjSlot>}

Por ejemplo la cadena:
Lobby~~X~0~1~0~0~int~3.
Representa:
Objeto: Lobby
Slot: X<-3

Cliente

Envia:	**Campo1**	**Campo2**	**Campo3**
ErrorMessage	size_t(lenght)	0x50	strError
execLobbyCMD	size_t(lenght)	0x01	strExecCode
showLobby	size_t(lenght)	0x02	""
execLocalCMD	size_t(lenght)	0x10	strExecCode
refresh	size_t(lenght)	0x11	""
setObjName	size_t(lenght)	0x12	strObjectName
setCodeSegment	size_t(lenght)	0x13	strCodeSegment
addSlot	size_t(lenght)	0x14	strSlotSelfCode
removeSlot	size_t(lenght)	0x15	strSlotName

```
swapMutability    size_t(lenght)    0x16        strSlotName
getSlotObject     size_t(lenght)    0x17        strSlotName
goBack            size_t(lenght)    0x18        strSlotName
availableWks      size_t(lenght)    0x20        ""
loadWk            size_t(lenght)    0x21        strWkName
newWk             size_t(lenght)    0x22        strWkName
deleteWk          size_t(lenght)    0x23        strWkName
closeWk           size_t(lenght)    0x24        strWkName
```

El *ProxyServer* se mantiene iterando en un ciclo *while* revisando su *flag* para saber cuándo enviar la solicitud al servidor.
La GUI hace uso del método *sendCmdMessage()* que es el encargado de setear el comando y el mensaje que deben ser enviados por el *ProxyServer* al servidor. Este mismo método setea el flag en verdadero para que el *ProxyServer* sepa que ya hay una petición para ser enviada.
Cuando el *ProxyServer* detecta que el flag cambió le hace un *send()* al Server con el comando y el mensaje.
En tanto y en cuanto no se reciba respuesta del servidor la GUI no se puede setear nuevamente el flag. Esto asegura que si el usuario desde la GUI da repetidas veces clic al botón, solo el primero clic tenga verdadero efecto. Permitiendo así asegurar que no haya una sobrecarga innecesaria (incluso contraproducente) de peticiones al servidor.

Archivos

Arbol de Carpetas:

bin

- client
- server
- mainWindow.glade

src

- client_select_wk_window.cpp
- mainWindow.glade
- server_parser_protocolo_servidor.cpp
- client_column_record.h
- client_select_wk_window.h
- Makefile_client_server
- server_parser_protocolo_servidor.h
- client_column_record_wk.h
- server_proxy_client.cpp
- client_main.cpp
- common_define.h
- server_proxy_client.h
- client_morph.cpp
- common_message.cpp
- server_server.cpp
- client_morph.h
- common_message.h
- server_accepter.cpp
- server_server.h
- client_morph_window.cpp
- common_proxy.cpp
- server_accepter.h
- server_virtual_machine.cpp
- client_morph_window.h
- common_proxy.h
- server_main.cpp
- server_virtual_machine.h
- client_parser_protocolo_morph.cpp
- common_socket.cpp
- server_mode_selector.cpp
- server_workspace.cpp
- client_parser_protocolo_morph.h
- common_socket.h
- server_mode_selector.h
- server_workspace.h
- client_parser_protocolo_workspaces.cpp
- common_thread.h
- server_object.cpp
- windows.glade
- client_parser_protocolo_workspaces.h
- server_object.h
- client_proxy_server.cpp
- server_parser.cpp
- client_proxy_server.h
- server_parser.h

doc

manuales

- Manual de Usuario.pdf
- Manual de Proyecto.pdf

- Documentacion Tecnica.pdf

anexos

- Caratulas.pdf
- Codigo Fuente.pdf
- Documentacion de Clases.pdf
- Enunciado.pdf

diagramas

- Diagrama de clases del Cliente.png
- Diagrama de clases del Servidor.png
- Diagrama de secuencia del Servidor.png
- Diagrama de clases del Servidor 2.png
- Diagrama de secuencia del Cliente.png
- Parser.png

pruebas

- 00 - Simil Ejemplo Enunciado.self
- 01 - Set pruebas Martin Di Paola.self
- 02 - prints.self
- 03 - unaryMessage.self
- 04 - addSlots.self
- 05 - removeSlots.self
- 06 - Exec Codesegment simple.self
- 07 - Exec Codesegment con lookup.self
- 08 - Exec Codesegment con lookup.self
- 09 - Parent Slot.self
- 10 - Set variable con persistencia.self
- 11 - Clonacion para Method Objects.self
- 12 - espacios.self
- 13 - Sumando puntos sin sobrecarga de metodos nativos.self
- 14 - Sumando puntos con sobrecarga de metodos nativos.self
- 15 - Operadores booleanos.self

Como el servidor no implementa **persistencia** de *workspaces* (ver sección Análisis de puntos pendientes en el Manual de Proyecto) no se utilizaron archivos en este aspecto.

El servidor, por otro lado, tiene uno de sus modos que consiste en leer un archivo local con código self para luego ejecutarlo y mostrar la salida por consola.

Si bien no se verifica la extensión de los mismos, se recomienda fuertemente utilizar extension *.self.

Los mismos deben contener pura y exclusivamente codigo self ejecutable. En otras palabras, los archivos son leídos y procesados como si fueran scripts de código self.

Programas intermedios y de prueba

No se crearon programas intermedios para realizar pruebas, sino que se reutilizó el modo del *server* para procesar archivos con código self.
Aprovechando esta funcionalidad se armó un set de pruebas para el *parser* (el *parser* que ejecuta código self).

Se puede realizar cada prueba con la siguiente línea de comandos desde el servidor.
$ server f <path carpeta pruebas>/<nombreArchivoDePrueba.self>

Resumen del set de pruebas confeccionado

00 - Simil Ejemplo Enunciado.self
Se ejecuta un test similar al del enunciado contemplando que no se soportan *keywordMessages*.

01 - Set pruebas Martin Di Paola.self
Se ejecuta un test que provee la cátedra (elaborado por Martín Di Paola) adaptado para los prints.

02 - prints.self
Se prueba que se realicen prints de objetos primarios y que se creen con los métodos nativos correctos.

03 - unaryMessage.self
Se prueba mensajes unarios.

04 - addSlots.self
Se prueba el método _AddSlots

05 - removeSlots.self
Se prueba el método _RemoveSlots

06 - Exec Codesegment simple.self
Se prueba ejecucion de codigo simple.

07 - Exec Codesegment con lookup.self
Se prueba ejecucion de codigo con variables usando lookup con self.

08 - Exec Codesegment con lookup.self
Se prueba ejecucion de codigo con variables usando lookup con self un poco más compleja.

09 - Parent Slot.self
Se prueba la ejecución de métodos a través de un *parent slot*.

10 - Set variable con persistencia.self
Se prueba la ejecución de un *code segment* que logre persistir cambios en otro objeto.

11 - Clonacion para Method Objects.self
Se prueba que se clonen los method objects antes de afectar sus variables locales.

12 - espacios.self
Se prueba que se ejecute correctamente código con caracteres especiales.

13 - Sumando puntos sin sobrecarga de metodos nativos.self
Un ejemplo complejo en el que se setea puntos, se los multiplica por escalares y se los suma entre sí.

14 - Sumando puntos con sobrecarga de metodos nativos.self
Mismo ejemplo que la prueba 13 pero se sobrecargan métodos nativos * + y print.

15 - Operadores booleanos.self
Se testean los operadores == y != de números y strings.

Mostramos la salida en consola de la prueba 01 elaborada por el ayudante:

```
INICIA TEST.
Se ejecuta un test que provee la cátedra (elaborado por Martin Di Paola) adaptado para los prints.
Debería dar 1.
1
Debería dar 2.
2
Debería dar 1.
1
Debería dar 3.
3
Se ejecutan mensajes de los que no se indica que se espera obtener.
Debería dar 6.
6
Debería dar 6.
6
Debería dar 5.
5
Debería dar 5.
5
Debería dar "hello".
hello
Debería dar "hello".
hello
Probando clones.
no tiene el slot q <- nil.
0x86a7a0 object: (| sayhello = 0x877d00. <_AddSlots>. <_RemoveSlots>. <clone>. <printObj>. |  )
0x877d00 object: (| <_AddSlots>. <_RemoveSlots>. <clone>. <printObj>. |  'hello'. )
si tiene el slot q <- nil probando que es un clone.
0x8b9a70 : (| q <- 0x8bdbb0. sayhello = 0x8b9e50. <_AddSlots>. <_RemoveSlots>. <clone>. <printObj>. |  )
0x8bdbb0 nil: (| <_AddSlots>. <_RemoveSlots>. <clone>. <print>. <printObj>. | nil. )
0x8b9e50 : (| <_AddSlots>. <_RemoveSlots>. <clone>. <printObj>. |  'hello'. )
Imprime 3.
3
Deberia fallar por inmutable.
Error.
El slot no es mutable
```

Código Fuente

Ver documento anexo:

Código Fuente