

Tutorial alla programmazione intelligente con **ROS**

Alessio Levratti

20 agosto 2018

Indice

| | | |
|----------|---|-----------|
| 1 | Introduzione | 3 |
| 1.1 | Perché questo tutorial? Anche l'autore se lo domanda... | 3 |
| 1.2 | Cosa aspettarsi dal futuro | 3 |
| 2 | Ubuntu | 4 |
| 2.1 | Il terminale | 5 |
| 2.1.1 | Il comando "sudo" | 6 |
| 2.1.2 | Alcuni comandi bash | 6 |
| 2.1.3 | Altri comandi utili per la bash | 7 |
| 2.1.4 | Installare e aggiornare il software | 7 |
| 2.2 | La <i>root</i> | 8 |
| 2.3 | Programmi utili... | 8 |
| 2.4 | Bash aliases e altri trucchetti | 9 |
| 3 | ROS e l'<i>Object Oriented Programming</i> | 10 |
| 3.1 | Introduzione all' OOP | 10 |
| 3.2 | Creare oggetti | 10 |
| 3.3 | L'ereditarietà | 12 |
| 3.4 | Re-introduzione a ROS | 14 |
| 3.5 | Creare oggetti in un nodo ROS | 14 |
| 3.5.1 | Esportare classi da un pacchetto ROS | 16 |
| 3.5.2 | Il server dei parametri di ROS: chi è questo sconosciuto? | 17 |
| 3.5.3 | Ascoltare più voci e usare TF | 19 |
| 4 | Commentare il codice per evitare maledizioni | 22 |
| 5 | Git e Version Control | 24 |
| 5.0.1 | Repository | 24 |
| 5.0.2 | Commit | 24 |
| 5.0.3 | Intestazioni | 25 |
| 5.0.4 | Il primo repository | 25 |
| 5.0.5 | .gitignore | 26 |
| 5.0.6 | Branching | 26 |
| 5.0.7 | Risoluzione dei conflitti | 28 |
| 5.0.8 | Rebasing | 28 |
| 5.1 | Collaborare con Git | 28 |
| 5.1.1 | Al lavoro con altri repository | 28 |
| 5.1.2 | Sistema di controllo versione distribuito | 29 |
| 5.1.3 | Copia del repository | 29 |
| 5.1.4 | Ricezione di cambiamenti dal repository remoto | 29 |
| 5.1.5 | Invio di cambiamenti al repository remoto | 30 |
| 5.1.6 | Tag | 30 |
| 5.2 | Bitbucket | 31 |
| 5.2.1 | Creare un repository remoto su Bitbucket | 31 |
| 5.2.2 | README.md | 31 |
| 5.2.3 | Issue tracking | 33 |
| 5.2.4 | Lavorare in team su di un repository Bitbucket | 34 |
| 6 | Concludendo, dulcis in fundo... | 35 |
| A | Appendice A: trasformazioni nello spazio | 36 |
| A.0.1 | Esempio: rotazione $z - y - z$ in terna corrente | 38 |
| A.0.2 | Esempio: rotazione $x - y - z$ in terna fissa | 38 |
| B | Appendice B: Glossario dei termini | 40 |

1 Introduzione

1.1 Perché questo tutorial? Anche l'autore se lo domanda...

Caro lettore, la prima cosa che ti sarà venuta in mente quando hai aperto questo pdf, con ogni probabilità, è stata: *“Ma perché diavolo devo sbattermi a leggere l'ennesimo tutorial sulla programmazione in ROS? Sul sito ufficiale c'è già abbastanza materiale per imparare a lavorare con ROS in maniera più che decente! Per non parlare delle innumerevoli pubblicazioni a riguardo [3–5, 7–9, 13] e chi più ne ha più ne metta! Senza contare che basta chiedere sul forum per avere una risposta in brevissimo tempo! E se nemmeno questo funziona, posso sempre chiedere a qualche abitante del laboratorio esperto di ROS (come ad esempio l'autore di questo tutorial) per avere aiuto.”*¹. Chiunque enunci una frase del genere ha pienamente ragione. Infatti, lo scopo di questo tutorial è quello di insegnare al lettore a programmare in maniera **INTELLIGENTE** in ROS. Questo significa:

1. Programmare ad oggetti: l' *Object Oriented Programming* (OOP) facilita ENORMEMENTE l'ampliamento e il riuso del codice. Il copia-incolla del codice porta alla stessa probabilità di avere un bug nel proprio codice di essere infettati dal morbillo se non si è vaccinati;
2. Lavorare per il proprio team: se il software che avete prodotto funziona alla perfezione, è efficiente in termini prestazionali e fa esattamente quello che deve fare, ma se nessuno poi potrà mettere mano al vostro codice, costringendo costui a re-inventare l'acqua calda e ricreare il vostro software da zero, non avete fatto un buon lavoro e, soprattutto, non avete lavorato per la vostra squadra. Quando scrivete del codice bisogna sempre pensare ad un altro programmatore che metterà mano al vostro codice;
3. Commentare il codice: anche il programmatore più navigato lancerà potentissime maledizioni voodoo sull'autore di un codice senza commenti;
4. Tenere traccia della storia del software: conoscere l'intera evoluzione del codice è di fondamentale importanza per tenere traccia dei bug e delle feature da implementare, avere un modo veloce ed efficace per “riavvolgere” il tempo e tornare a versioni precedenti ed evitare di ripetere errori fatti in passato. Avete mai sentito parlare di Version Control System (VCS)? Se non sapete cos'è, sappiate che questo gruppo di ricerca (ARSCControl + IT-I) usa Git e BitBucket, che vi piaccia o meno. Correte a informarvi [6]! Inoltre un Repository remoto è un modo utile per aver salvato il codice in un luogo sicuro...;
5. Avere un'idea della struttura del codice da implementare: **MAI** e poi **MAI** iniziare il proprio lavoro di programmazione senza avere la minima idea di come sarà strutturato il codice.

In breve, questo tutorial mira a portare i lettori ad avere una metodologia di lavoro comune volta ad aiutarsi l'uno con gli altri e, in particolare, ad avere una buona metodologia di programmazione in ROS.

Questa guida sarà la prima di una serie (non tutta scritta da me) con indicazioni su come lavorare in squadra su progetti di natura diversa.

Alessio Levratti

1.2 Cosa aspettarsi dal futuro

Questo tutorial è strutturato in questo modo:

- La sezione 2 propone una breve introduzione al mondo di Ubuntu;
- La sezione 3 introduce della programmazione C++ e introduce ROS, ampliando i concetti visti nel tutorial ufficiale con alcuni concetti di programmazione ad oggetti;
- La sezione 4 verranno spiegate alcune utili strategie per commentare in maniera corretta il codice senza passare il doppio del tempo impiegato a scrivere il codice per commentarlo. Verrà inoltre introdotto il concetto di “Doxygen” e documentazione di un software;
- In sezione 5 verrà introdotto il concetto di “*Version Control System*” e di Git (pronunciato “*ghit*”);
- Infine, in sezione 6 verranno tratte alcune conclusioni

¹Da notare che la sequenza delle operazioni da fare per risolvere un problema in ROS è proprio quello indicato nell'ipotetico dialogo; prima di tutto controllare sul sito ufficiale di ROS, controllare in letteratura, chiedere aiuto sul forum ufficiale e SOLO ALLA FINE chiedere aiuto a qualcuno del laboratorio! Ricordate sempre “RTFM”!

2 Ubuntu

Ubuntu è un sistema operativo “open-source” basato su kernel Linux. È pubblicato dall’azienda Canonical Ltd. che fornisce anche supporto commerciale. Perché preferire un sistema operativo open-source? I motivi sono molteplici:

- L’open-source incoraggia la modifica e la personalizzazione del software, in opposizione alla concezione in cui una singola versione deve soddisfare molteplici approcci di natura diversa;
- La filosofia open-source ritiene che il lavoro e la collaborazione di tanti superino i vantaggi del lavoro in un progetto chiuso e controllato con un limitato numero di sviluppatori il cui solo ed unico interesse è il “vile denaro”;
- In caso di bisogno, chiunque abbia una sufficiente conoscenza informatica può modificare a piacere il software.

Per dimostrarvi che l’autore *non è assolutamente di parte* (Ahahahahaha!!!! Non scherziamo! Sono ESTREMAMENTE di parte!) in tabella 1 mettiamo a confronto due sistemi operativi: Ubuntu vs. MS Windows².

| Attributo di confronto | Ubuntu | MS Windows |
|----------------------------|---|---|
| Costo | È gratis!!!! | Ogni singolo utente deve pagare una licenza di utilizzo |
| Rilascio di nuove versioni | Stessa versione per utilizzo professionale o per utilizzo domestico. Ogni sei mesi una nuova versione. Ogni due anni una nuova versione LTS. | Edizioni “Home” e “Professional” separate. Frequenza di rilascio versioni MOLTO bassa. |
| Sicurezza | Privilegi di amministratore (root user) per modificare i file di sistema. Raramente soggetto a malware, virus o attacchi informatici. | Accesso estremamente semplice a qualsiasi file, anche i più critici. Regolarmente è soggetto ad attacchi hacker, virus e malware. |
| Personalizzazione | Facile da personalizzare. È possibile far girare diverse versioni in parallelo. | La versione standard del sistema operativo non è personalizzabile. Del resto, se piace a Bill Gates perché non deve piacere a te? |
| Memorizzazione dati | L’upgrade e il downgrade sono banali. I dati utenti sono salvati nella “home” directory. Per migrare da un computer ad un altro basta copiare la “home” | Upgrade obbligatorio sempre nei momenti peggiori. Downgrade praticamente impossibile. Gli utenti possono salvare i propri dati in qualsiasi posizione del file system. Valli a ritrovare poi tutti i tuoi file... |

Tabella 1: Ubuntu Vs. MS Windows

Dalla prima release ufficiale del 2005, Ubuntu 4.10, si sono susseguite innumerevoli versioni. Ogni quattro versioni rilasciate, viene rilasciata una LTS (Long Time Support) il cui supporto dura molto di più rispetto alle versioni normali.

Ed ecco un “piccolo” elenco delle versioni di Ubuntu uscite negli anni con indicato la fine del supporto:

Ubuntu 4.10 (Warty Warthog): Aprile 2005 - Ottobre 2006

Ubuntu 5.10 (Breezy Badger): Ottobre 2005 - Aprile 2007

Ubuntu 6.06 LTS (Dapper Drake): Giugno 2006 - Luglio 2009

Ubuntu 6.10 (Edgy Eft): Ottobre 2006 - Aprile 2007

Ubuntu 7.04 (Feisty Fawn): Aprile 2007 - Ottobre 2008

Ubuntu 7.10 (Gutsy Gibbon): Ottobre 2007 - Aprile 2009

Ubuntu 8.04 LTS (Hardy Heron): Aprile 2008 - Aprile 2011

²Per chi si chiedesse perché non ho inserito nel confronto i sistemi MAC la mia risposta è “*AAAAAAAAAAAAAAAAHAHAH!!!!!!*”

Ubuntu 8.10 (Intrepid Ibex): Ottobre 2008 - Aprile 2010

Ubuntu 9.04 (Jaunty Jackalope): Aprile 2009 - Ottobre 2010

Ubuntu 9.10 (Karmic Koala): Ottobre 2009 - Aprile 2011

Ubuntu 10.04 LTS (Lucid Lynx): Aprile 2010 - Aprile 2015

Ubuntu 10.10 (Maverick Meerkat): Ottobre 2010 - Aprile 2012

Ubuntu 11.04 (Natty Narwhal): Aprile 2011 - Ottobre 2012

Ubuntu 11.10 (Oneiric Ocelot): Ottobre 2011 - Maggio 2013

Ubuntu 12.04 LTS (Precise Pangolin): Aprile 2012 - Aprile 2017

Ubuntu 12.10 (Quantal Quetzal): Ottobre 2012 - Maggio 2014

Ubuntu 13.04 (Raring Ringtail): Aprile 2013 - Gennaio 2014

Ubuntu 13.10 (Saucy Salamander): Ottobre 2013 - Luglio 2014

Ubuntu 14.04 LTS (Trusty Tahr): Ottobre 2014 - Aprile 2019

Ubuntu 14.10 (Utopic Unicorn): Aprile 2014 - Luglio 2015

Ubuntu 15.04 (Vivid Vervet): Aprile 2015 - Febbraio 2016

Ubuntu 15.10 (Wily Werewolf): Ottobre 2015 - Luglio 2016

Ubuntu 16.04 LTS (Xenial Xerus): Aprile 2016 - Aprile 2021

Ubuntu 16.10 (Yakkety Yak): Ottobre 2016 - Luglio 2017

Ubuntu 17.04 (Zesty Zapus): Aprile 2017 - Gennaio 2018

Ubuntu 17.10 (Artful Aardvark): Ottobre 2017 - Luglio 2018

Ubuntu 18.04 LTS (Bionic Beaver): Aprile 2018 - Aprile 2023

Ubuntu 18.10 (Cosmic Cuttlefish): Ottobre 2018 - Luglio 2019

2.1 Il terminale

Alcuni lettori inesperti potrebbero obiettare dicendo “*Si vabbè, ma Ubuntu non ha l’interfaccia semplice e intuitiva di Windows!*”. Ciò è estremamente falso! Per gli utenti ai quali piace farsi venire il tunnel carpale per l’uso intensivo del mouse, Ubuntu offre un’interfaccia a finestre in tutto e per tutto uguale a quella di Windows.

Infatti Ubuntu, ufficialmente dalla versione 11.04 (Natty Narval), possiede come interfaccia grafica Unity. Unity è una shell grafica sviluppata da Canonical Ltd. per l’ambiente desktop GNOME dei sistemi operativi Ubuntu. Unity ha debuttato nella edizione netbook Ubuntu 10.10. Tuttavia, per sfruttare a fondo le potenzialità di Ubuntu, l’interfaccia del terminale rimane la migliore! I motivi sono molteplici:

- I comandi sono standardizzati tramite le specifiche POSIX, quindi uno script scritto per un computer, probabilmente, funzionerà anche su altri;
- UNIX è stato creato per l’uso tramite terminale. Praticamente tutto è configurabile tramite linea di comando;
- È flessibile. I comandi possono essere inseriti in una pipeline, il loro output può essere catturato e sostituito come input di altri comandi;
- Le utilità di UNIX sono state progettate per fare ognuna una singola cosa e a farla nel miglior modo possibile;
- Possibilità di automatizzare delle operazioni. I comandi della bash permettono di semplificare e/o automatizzare script lunghi e complessi.

2.1.1 Il comando “sudo”

Il comando “**sudo**” esegue un comando con i privilegi di amministratore (root-user), cosa necessaria, ad esempio, se si vuole lavorare con directory o file ai quali il normale utente non ha accesso. Quando si usa **sudo** verrà richiesta la password di amministratore³.

2.1.2 Alcuni comandi bash

- Il carattere `~` (tilde) indica la directory “home”. Ad esempio, per l’utente “chewbecca” la home avrà il seguente percorso “`/home/chewbecca`”;
- **pwd**: mostra il percorso completo alla directory nella quale vi trovate. Letteralmente, **pwd** significa “Print Working Directory”.
- **ls**: permette di visualizzare a schermo la lista dei file presenti nella directory corrente. Usato, ad esempio, con l’opzione `-al` permette di visualizzare i file nascosti (il quale nome inizia con un punto `.`) e i permessi (es: `ls -al`).
- **cd**: letteralmente significa “change directory”. Esempi:
 - Per muoversi nella “root”: `cd /`
 - Per navigare alla “home”: `cd` o `cd`
 - Per salire di livello nella gerarchia del file-system: `cd ..`
 - Per navigare indietro nella navigazione: `cd -`
- **cp**: copia un file. Es: “`cp jango_fett boba_fett`” copierà il file “jango_fett” e lo rinominerà in “boba_fett”, creandone un perfetto clone. Se si desidera copiare una directory, occorre farlo con “`cp -R dir destinazione`” (copia ricorsiva);
- **mv**: sposta il file. Questo comando serve anche per rinominare i file. Es: “`mv darth ~/`” copierà il file darth nella home. Es: “`mv anakin darth_vader`” rinomina il file “anakin” in “darth_vader”
- **rm**: cancella un file. Es: “`rm jarjar_binx`” cancellerà definitivamente il file jarjar_binx. “`rm -R alderan/`” cancellerà definitivamente la directory “alderan”. **ATTENZIONE!** I file cancellati in questo modo non sono recuperabili!
- **rmdir**: cancella una directory vuota.
- **mkdir**: crea una nuova cartella. Es: `mkdir death_star` creerà la cartella “death_star”
- **touch**: crea un file vuoto. Es: “`touch r2d2.drd`” creerà il file “r2d2.drd”

Per lanciare un eseguibile occorre aggiungere “.” prima del nome dell’eseguibile. Il comando “`.ordine66`” eseguirà il programma “ordine66”. È importante sottolineare che un file può essere eseguito SOLO se è un eseguibile. Ad esempio, un programma scritto in C++, che solitamente avrà estensione “.cpp”, deve essere precedentemente compilato per essere eseguito. Per altri tipi di programmi scritti in linguaggi interpretati, come Python o Ruby, occorre aver installato l’interprete di tale linguaggio. Infine, il file può essere eseguito SOLO se ha i permessi per farlo.

Per cambiare i permessi di un file bisogna usare il comando “**chmod**”. Es: “`sudo chmod 777 darth_sidious.emp`” aggiungerà i permessi di lettura, scrittura ed esecuzione per ogni tipo di utente sul file “darth_sidious.emp”. In pratica, con l’opzione 777, il file “darth_sidious.emp” riceverà il potere assoluto...

La tabella 2 riassume i codici dei permessi.

| Permesso | Azione | Opzione di chmod |
|------------|--------------------|------------------|
| lettura | (lettura del file) | “r” or “4” |
| scrittura | (modifica) | “w” or “2” |
| esecuzione | (esecuzione) | “x” or “1” |

Tabella 2: Permessi dei file

I permessi di un file, come detto in precedenza, possono essere visualizzati con il comando “`ls -al file`” (vedi tabella 3)

³Se volete divertirvi date un’occhiata qui

| Utente | ls -al output |
|--------------|---------------|
| Proprietario | -rwx----- |
| Gruppo | -----rwx--- |
| Altri | -----rwx |

Tabella 3: Output del comando `ls -al`

2.1.3 Altri comandi utili per la bash

- **df**: mostra lo spazio disponibile per ogni partizione “montata” dal sistema operativo. Aggiungendo l’opzione **-h** renderà l’output del comando “leggibile” visualizzando le informazioni sulla memoria in Megabyte e Gigabyte. Es: `df -h`;
- **free**: mostra la quantità di memoria libera e occupata nel sistema. Aggiungendo l’opzione **-m** l’informazione sarà restituita in Megabyte. Es: `free -m`;
- **top**: letteralmente significa “table of processes”. Mostra la tabella dei vari processi in esecuzione nel sistema e il loro consumo di risorse come percentuale della CPU, occupazione di memoria RAM, ecc... Per terminare l’aggiornamento continuo della tabella occorre premere “q”;
- **uname -a**: visualizza tutte le informazioni del sistema come nome, nome e versione del kernel, ecc...
- **lsb_release -a**: mostra la versione del sistema operativo e il nome;
- **ifconfig**: mostra le informazioni di tutte le interfacce di rete presenti (scheda ethernet, wifi, ecc...) come indirizzo IP e MAC-address.
- **lsusb**: visualizza la lista dei dispositivi USB collegati.

2.1.4 Installare e aggiornare il software

- `sudo apt-get update`: aggiorna la lista dei sorgenti dei repository. Da eseguire periodicamente;
- `sudo apt-get upgrade`: aggiorna tutto il software installato;
- `sudo apt-get install <nome_pacchetto>`: installa il pacchetto selezionato;
- `sudo apt-get remove <nome_pacchetto>`: disinstalla il pacchetto selezionato.

Se siete dei feticisti dell’interfaccia grafica, potete utilizzare l’“Ubuntu Software Center” per scaricare e installare i nuovi programmi (vedi Figura 1)

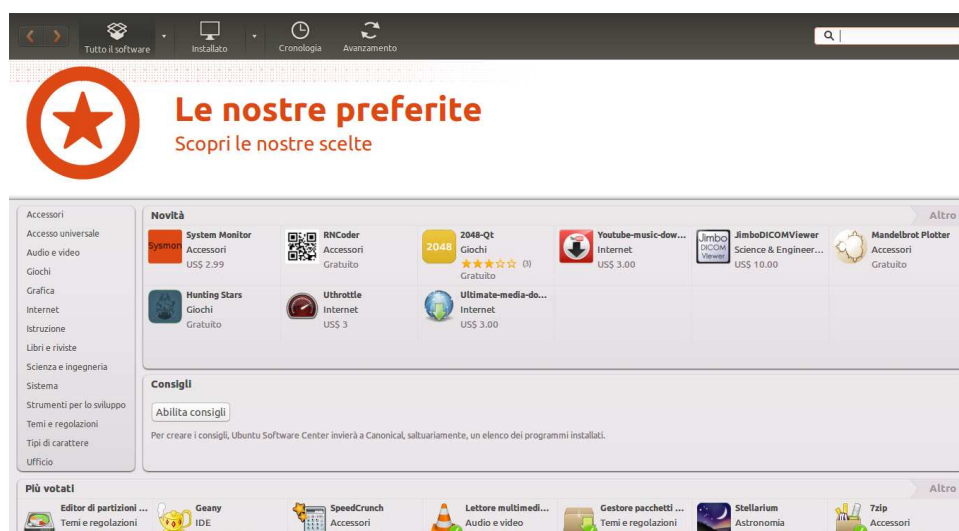


Figura 1: Schermata dell’“Ubuntu Software Center”

2.2 La root

Ubuntu, come ogni altro sistema operativo derivato da UNIX, organizza i file in una gerarchia chiamata “file-system”. Alla radice del file-system di Ubuntu si trova (guarda caso) la root directory. Nella root sono contenute queste cartelle:

- **/bin**: contiene i comandi base di Ubuntu come **ls**, **cd**, ecc...;
- **/boot**: contiene i file necessari per l’avvio del sistema, come il kernel Linux, la configurazione del boot-loader, ecc...;
- **/dev**: contiene i riferimenti alle periferiche hardware, rappresentate come file;
- **/etc**: contiene i file file di configurazione del sistema;
- **/home**: “*home sweet home*”;
- **/lib**: contiene le librerie dinamiche e i moduli del kernel;
- **/media**: è il punto di “mount” di tutti i dispositivi rimovibili (CD, DVD, chiavette USB, ecc...);
- **/mnt**: un altro punto di “mount” per i dispositivi in rete;
- **/opt**: contiene il software aggiuntivo installato NON dal package manager;
- **/proc**: filesystem virtuale che fornisce un meccanismo tramite il quale il kernel invia informazioni ai vari processi;
- **/root**: l’Head-Quarter del capo, ovvero la “home” del super-user.
- **/run**: è un filesystem temporaneo utilizzato in fase di avvio;
- **/sbin**: contiene i comandi che generalmente vengono utilizzati dal super-user;
- **/usr**: contiene la maggior parte delle utilities e delle applicazioni e replica al suo interno la struttura della root;
- **/var**: cartella dedicata ai dati variabili, come database, log files, ecc...;

2.3 Programmi utili...

- **gedit**: un editor di testi equivalente a Notepad++ di Windows;
- **nano**: editor di testo utilizzabile direttamente sul terminale. Utile se si deve modificare un programma su un altro computer con il quale ci si sta interfacciando con SSH...;
- **ssh**: Secure Shell (SSH) è un protocollo di rete crittografico per operare servizi di rete in maniera sicura su di una rete NON sicura. Permette anche il login da remoto;
- **git**: Version Control System. Di questo ve ne parlerò fino alla nausea più tardi...
- **evince**: visualizzatore di documenti come pdf, PostScript, DjVu, DVI, ecc...;
- **libreoffice**: l’equivalente di MS Office, ma per Ubuntu e gratis;
- **gimp**: (GNU Image Manipulation Program), programma di image editing di Ubuntu. È l’equivalente di Adobe Photoshop e Illustrator in un singolo programma;
- **firefox**: browser web;
- **thunderbird**: client di posta elettronica.

2.4 Bash aliases e altri trucchetti

- Siete stanchi di scrivere complessi e inutilmente lunghi comandi sul terminale? Allora create degli alias! Create un file “.bash_aliases” nella vostra home (se non esiste già). All'interno del file è possibile definire degli alias dei comandi. Es: `alias AGGIORNA='sudo apt-get update && sudo apt-get upgrade'`. Dopo aver riavviato il terminale, digitando il nuovo comando AGGIORNA sarà possibile aggiornare sia la lista dei sorgenti dei pacchetti che i pacchetti stessi.
- È possibile configurare il tasto “su” della tastiera↑ per eseguire la ricerca all'indietro di un comando nella “history” dei comandi. Per fare ciò basta aggiungere al file .bashrc le seguenti righe:

```
bind '"\e[A": history-search-backward'  
bind '"\e[B": history-search-forward'
```

3 ROS e l'Object Oriented Programming

3.1 Introduzione all' OOP

Questo tutorial non vuole sostituirsi in maniera più assoluta ad alcuna guida che è possibile trovare su internet (come quella di cplusplus.com, dove l'autore di questo tutorial ha effettivamente imparato a programmare) o a qualche fantastico libro [1,2,10–12](in particolare, [11] e [10] vanno letti con reverenziale attenzione essendo scritti dal sommo creatore del linguaggio C++, Bjarne Stroustrup, sempre sia lodato!). Questo tutorial ripercorrerà alcuni concetti utili del C++ imparati dall'autore negli anni di lavoro con ROS.

Partendo dal presupposto che TUTTI i lettori conoscano almeno le basi del C++ (tipi di variabili, puntatori, costrutti fondamentali come if/else, cicli for e while, switch, funzioni e strutture dati, ecc...) inizieremo a parlare di classi in C++.

Nella programmazione orientata agli oggetti una classe è un costrutto di un linguaggio di programmazione usato come modello per creare oggetti. Il modello comprende attributi e metodi che saranno condivisi da tutti gli oggetti creati (istanze) a partire dalla classe. Un "oggetto" è, di fatto, l'istanza di una classe.

Una classe è identificabile come un tipo di dato astratto che può rappresentare una persona, un luogo, oppure una cosa, ed è quindi l'astrazione di un concetto, implementata in un software. Fondamentalmente, essa definisce al proprio interno lo stato, i cui dati sono memorizzati nelle cosiddette variabili membro o attributi, e il comportamento dell'entità di cui è rappresentazione, descritto da blocchi di codice riutilizzabili chiamati metodi.

Una delle caratteristiche fondamentali dell'approccio orientato agli oggetti è la maggiore "fluidità" (rispetto agli approcci precedenti, come quello procedurale) con cui le fasi di analisi, progettazione e implementazione sfociano ciascuna nella successiva. Questa fluidità è dovuta al fatto che i linguaggi orientati agli oggetti forniscono una serie di strumenti sintattici e semantici che sono la diretta trasposizione degli strumenti concettuali di cui si è parlato per quanto riguarda le fasi di analisi e progettazione.

Un linguaggio orientato agli oggetti fornisce un costrutto di classe strutturalmente corrispondente al concetto astratto di classe menzionato sopra: una classe descrive un tipo di oggetti (una categoria di entità) in termini di un insieme di variabili interne o variabili d'istanza di cui tali oggetti sono dotati (attributi) e un insieme di procedure dette metodi che possono essere eseguite su di essi (operazioni). Una variabile interna di una classe che contenga un riferimento a un'istanza di un'altra classe può corrispondere a un'associazione; una variabile interna che contenga direttamente un'istanza vera e propria può considerarsi trasposizione implementativa del concetto di aggregazione; e infine l'ereditarietà corrisponde direttamente alla relazione ISA.

Se da un punto di vista storico e tecnico la classe dei linguaggi orientati agli oggetti si può considerare come una evoluzione del record di linguaggi procedurali tipo C o Pascal, essa sottende un approccio completamente diverso alla programmazione, in cui i tipi di dati, corredati delle loro operazioni (metodi) diventano centrali. Gran parte delle novità significative di questo approccio sono legate ai concetti di ereditarietà, incapsulamento o information hiding e polimorfismo.

Dopo questa "sbrodolata" di concetti astratti (che ho copia-incollato da wikipedia, LOL!), cerchiamo di capire meglio cosa diavolo è una classe in C++.

3.2 Creare oggetti

Creiamo il nuovo file "starships/mercantile_corelliano_yt1300.h":

```
#include<iostream>
namespace starships {

// Mercantile Corelliano YT-1300
class MercantileCorellianoYT1300
{
public:
    MercantileCorellianoYT1300();
    ~MercantileCorellianoYT1300();

    // Capacita di carico della nave
    double capacita_carico;

protected:

    // Numero membri dell'equipaggio
    int numero_membri_equipaggio_;
```

```

        double viaggioLuce();
    }
} //namespace starships

```

Partiamo dal nome della classe dichiarata “**MercantileCoreellianoYT1300**”. Come potete vedere, la lettera iniziale è MAIUSCOLA, come le iniziali delle parole che ne compongono il nome. Secondo la guida allo stile di programmazione di Google la lettera iniziale del nome di una classe DEVE essere sempre maiuscola. Si può ricorrere al cosiddetto “*camel-case*” per rendere leggibile il nome senza dover inserire eccessivi “*underscore*”.

La classe è inserita all’interno dello “spazio dei nomi” **starships**. È importante utilizzare dei namespace per evitare di far “collidere” classi diverse ma con lo stesso nome. Alla chiusura del namespace è utile aggiungere un commento che ricordi il nome del namespace che si sta chiudendo.

All’interno della classe gli attributi devono essere sempre scritti in minuscolo. Gli attributi protetti o privati devono avere un “underscore” alla fine del nome. È sempre meglio dare nomi auto esplicativi alle variabili ed è sempre meglio aggiungere commenti per chiarirne l’utilizzo! Da notare che al momento mi sono limitato a mettere commenti molto brevi a solo scopo d’esempio. I commenti vanno scritti in modo diverso se volete che in base ad essi venga prodotta in modo automatico la documentazione del software, ma di ciò ne parleremo più avanti quando introdurrò Doxygen.

Gli attributi e i metodi di una classe possono essere dichiarati come

- **public**: gli attributi definiti in questo modo possono essere visti e modificati dall’esterno della classe. Quindi all’interno di un programma che istanzia la classe, posso leggere e modificare questi parametri. Stessa cosa per i metodi pubblici: possono essere invocati sia da dentro che da fuori della mia classe;
- **private**: gli attributi pubblici possono essere visti e modificati solo internamente alla mia classe. I metodi privati, invece, possono essere invocati SOLO all’interno della classe;
- **protected**: gli attributi possono essere visti e modificati SOLO all’interno della mia classe O delle classi figlie (vedi la prossima sezione sull’ereditarietà). Stessa cosa per i metodi: possono essere invocati solo internamente alla classe o alle classi figlie.

Ogni classe, come dovreste già sapere se avete studiato il linguaggio C++, possiede un metodo Costruttore e un metodo Distruttore. Il Costruttore viene invocato quando si vuole creare un’istanza della classe, ovvero un oggetto. All’interno di esso, solitamente, vengono inizializzati tutti gli attributi della classe. A volte, può capitare che alcuni attributi debbano essere inizializzati in un altro momento. Nel caso la propria classe abbia delle istanze di altre classi come attributi, può essere utile dichiarare tali oggetti come puntatori ad oggetti ed inizializzarli come puntatori a “NULL”. Nel caso nella mia classe non debba inizializzare nulla, posso anche NON dichiarare il costruttore della mia classe. In questo caso, il compilatore capirà che la mia classe non ha un costruttore e l’istanziamento della mia classe richiederà il costruttore di default, che è equivalente a

```

[...]  
class StormTrooper  
{  
public:  
    StormTrooper(){};  
[...]  
}

```

L’uso del distruttore, invece, è più complicato; il distruttore viene invocato, appunto, quando l’oggetto in questione deve essere distrutto, come ad esempio quando un programma volge al termine. Nel caso volessi distruggere un oggetto prima della fine del programma, si può ricorrere all’istruzione “**delete**” alla quale va passato il puntatore all’oggetto da distruggere:

```

[...]  
class TieFighter  
{  
public:  
    TieFighter(const std::string& nome) : nome_pilota_{nome}{};  
    ~TieFighter(){delete nome_pilota_;}  
  
private:  
    std::string* nome_pilota_;  
}  
  
int main (int argc, char** argv)

```

```

{
    TieFighter* pesky_tie_fighter = new TieFighter("StormTrooper12547");
    TieFighter other_tie_fighter("StormTrooper012844");
    [...]

    // I due Tie Fighter sono stati colpiti!
    delete pesky_tie_fighter;
    delete &other_tie_fighter;
    // Tie Fighter distrutti

    return 0;
}

```

Nell'esempio sopra, il distruttore viene utilizzato per de-allocare della memoria dinamica precedentemente allocata nella costruzione del puntatore ad una stringa, ma nel distruttore è possibile inserire diverse istruzioni per concludere la vita di un oggetto come ad esempio:

- Chiudere la comunicazione con un dispositivo esterno;
- Salvare la configurazione dell'oggetto in un file;
- Fermare dei motori;
- ecc...

3.3 L'ereditarietà

Nel caso volessi creare una classe “simile” alla classe `MercantileCorellianoYT1300` ma apportando qualche modifica, come ad esempio aggiungere qualche feature, posso percorrere due strade:

1. Fare il “copia-incolla” della classe e aggiungere gli attributi e i metodi che mi servono;
2. Sfruttare l'ereditarietà: creo una classe “figlia” di `MercantileCorellianoYT1300` che erediterà TUTTI gli attributi e i metodi della classe padre. Dopodiché potrò espandere la nuova classe.

```

#include <iostream>
#include "starships/mercantile_corelliano_yt1300.h"
namespace rebel_starships {
// Il leggendario Millenium Flacon
class MillenniumFalcon: public starships::MercantileCorellianoYT1300
{
public:
    MillenniumFalcon();
    ~MillenniumFalcon();

    // Velocita massima sub-luce
    double max_speed;

private:

    // Livello degli scudi del Falcon
    double livello_scudi_;

    double sparaTurboLaser(double);
    void sparaSiluriAConcussione(int);
}
} // rebel_starships

```

Facendo in questo modo, la nuova classe `MillenniumFalcon` erediterà TUTTE le funzioni e TUTTI gli attributi della classe `MercantileCorellianoYT1300`⁴ ai quali potrò aggiungere altri attributi e metodi. Se poi volessi creare dei comportamenti diversi per determinati metodi, posso ricorrere al polimorfismo e fare “l’overload” dei metodi che occorre modificare nella definizione della mia nuova classe “`rebel_starships::MillenniumFalcon`”.

⁴Citando i latini: “*Qualis pater, talis filius*”

Riassumiamo quello detto finora: se sono già in possesso di una classe che funziona e fa esattamente tutto quello per cui è stata progettata e, soprattutto, è già stata debuggata, È ASSOLUTAMENTE INUTILE RICREARE LA STESSA CLASSE! Come già detto in precedenza, il copia incolla è padre di bug. Se possibile, bisogna evitare ad ogni costo di ricreare classi funzionanti per il puro sollazzo del programmatore di riscoprire l'acqua calda. Inoltre, creare nuove classi inutili porta alla creazione di file inutili e quindi ad occupare spazio in memoria che, come voi lettori ben sapete, costa fior di quattrini!

Un'altra cosa molto importante quando si programma è mantenere l'ordine: è buona norma mantenere separate la dichiarazione delle classi (tipo quelle negli esempi sopra) e la loro definizione, ovvero dove viene codificata la loro implementazione.

Ad esempio, la cartella “include” dovrà contenere tutti i file *header* con la dichiarazione delle classi, mentre nella cartella “src” si troveranno i file *cpp* con l'implementazione. Nel caso sia necessario creare un eseguibile, esso sarà definito nel file *main.cpp* collocato nella cartella “src”.

Torniamo ora ai concetti base della programmazione ad oggetti, parlando delle classi astratte. Modifichiamo il primo esempio di classe “starships/MercantileCorellianoYT1300”:

```
#include<iostream>
namespace starships {

// Mercantile Corelliano YT-1300
class MercantileCorellianoYT1300
{
public:
    MercantileCorellianoYT1300();
    ~MercantileCorellianoYT1300();

    // Capacita di carico della nave
    double capacita_carico;

protected:

    // Numero membri dell'equipaggio
    int numero_membri_equipaggio_;

    double viaggioLuce();
    virtual double sparaTurboLaser(double);
    virtual void sparaSiluriAConcussione(int);
}
} //namespace starships
```

La parola *virtual* significa che ogni classe figlia dovrà implementare la propria versione del metodo. Rifacendosi al secondo esempio di classe, “rebel_starships/MillenniumFalcon” dovrà re-implementare le classi definite come *virtual*.

Se invece un metodo di una classe è definito come segue

```
[...]
virtual double viaggioLuce() = 0;
```

siamo di fronte ad una classe puramente astratta! Ogni classe figlia di questa classe, dovrà implementare la sua versione del metodo astratto, mentre la classe genitrice NON avrà la definizione del metodo dichiarato astratto. Nel caso si dimenticasse di implementare un metodo astratto ereditato, il compilatore o il linker genereranno un errore.

Infine, l'ultimo concetto particolarmente utile, sono gli attributi definiti “*static*”. Il valore di questi attributi sarà lo stesso per ogni istanziazione della classe. Esempio:

```
[...]
namespace republic_army{

class CloneTrooper
{
public:
    CloneTrooper(const std::string name) : clone_trooper_name_(name) {army_size_++;};
    ~CloneTrooper(){army_size_--;};
}
```

```

    static int army_size;

private:
    std::string clone_trooper_name_;
}

int republic_army::CloneTrooper::army_size = 0;

int main (int argc, char** argv)
{
    [...]
    republic_army::CloneTrooper soldier1("07425"), soldier2("354785");
    std::cout << "Army size: " << republic_army::CloneTrooper::army_size;
}

```

Nell'esempio qui sopra, l'attributo "army_size" sarà LA STESSA variabile per tutti gli oggetti creati. Quindi, l'output del programma sarà "Army size: 2". Ricordatevi che gli attributi "static" devono essere sempre inizializzati. Questo lo potrete fare nel file di implementazione della classe.

Una classe può avere anche metodi dichiarati come "static". Queste funzioni saranno comuni a TUTTI gli oggetti istanziati dalla stessa classe e possono essere utilizzate come se fossero delle funzioni-non-membro, ma alle quali è possibile accedere come se fossero delle funzioni-membro (perdonate il gioco di parole). Questi metodi statici, per ovvie ragioni, non avranno accesso agli attributi della classe, tranne che a quelli di tipo static. Esempio:

```

class JediKnight
{
public:
    static void useTheForce();
}

int main (int argc, char** argv)
{
    [...]
    JediKnight luke_skywalker();
    // Luke usa la forza
    luke_skywalker.useTheForce();
    // Uno Jedi usa la forza
    JediKnight::useTheForce();
}

```

3.4 Re-introduzione a ROS

Caro lettore, giunti a questo punto è il momento che tu faccia una pausa. Poniti queste domande:

1. Ho letto e compreso tutto ciò che il Tutorial Ufficiale di ROS ha da trasmettermi?
2. Ho letto e compreso tutto ciò che il Tutorial sui Frame ha da trasmettermi?
3. So cos'è un "launch-file"?
4. Ho risposto onestamente alle prime due domande?

Nel caso abbiate risposto "NO" ad almeno una delle quattro precedenti domande, correte ai ripari e andate a ripassare!

Nel caso siate già ferrati negli argomenti trattati nei tutorial, potete continuare. Se tuttavia avete ancora lacune per quanto riguarda i frame, le trasformazioni nello spazio e l'infame quaternione unitario, potete leggere l'Appendice A in fondo a questo documento.

3.5 Creare oggetti in un nodo ROS

Ora, dopo pagine e pagine di nozioni di base, arriva il bello...

Prima di specificare come creare una classe funzionante nel sistema ROS, cerchiamo di capire come strutturare al meglio un pacchetto e un nodo ROS per renderlo usabile e riusabile al meglio.

Il pacchetto può essere costruito con il comando apposito:

```
catkin_create_pkg starships_package roscpp tf sensor_msgs nav_msgs
```

Nel comando precedente, `starsihips_package` è il nome del pacchetto, mentre i nomi seguenti sono le dipendenze:

- `roscpp`: libreria base di ROS per la programmazione in C++;
- `tf`: libreria per l'uso del pacchetto `tf` per la gestione dei frame;
- `sensor_msgs`: libreria con le strutture base per la gestione dei messaggi dei sensori;
- `nav_msgs`: libreria con le strutture base per la gestione dei messaggi di navigazione, come ad esempio l'odometria.

Il pacchetto così creato avrà i seguenti elementi:

- Un file `CMakeLists.txt` nel quale specificare cosa compilare, cosa generare e come linkare il progetto;
- Un file `package.xml` con le dipendenze;
- Una cartella `/src` nella quale andranno TUTTI i file sorgente `.cpp`;
- Una cartella `/include/starships_package` dove andranno gli header file `.h`.

Per i file `.launch`, è opportuno creare manualmente una cartella “launch” che li contenga. Lo stesso vale per i file di configurazione o di parametri, che andranno in una cartella “config” o “param”. I file di descrizione dei messaggi ad-hoc o dei servizi andranno, rispettivamente, nelle cartelle “msg” e “srv”.

Ad esempio, creiamo il file `millennium_falcon.h` nella cartella degli header e il file `millennium_falcon.cpp` nella cartella dei sorgenti. Nel primo verrà dichiarata la classe “`MillenniumFalcon`”, mentre nel secondo verrà implementata.

`millennium_falcon.h`:

```
#include <ros/ros.h>
#include <sensor_msgs/Laser.h>

namespace starhips {
class MillenniumFalcon : public MercantileCorellianoYT1300
{
public:
    MillenniumFalcon()
    ~MillenniumFalcon();

    void spinner();

private:
    // Nodehandle
    ros::NodeHandle nh_;
    // Subscriber per i laser
    ros::Subscriber laser_subscriber_;
    // Membri dell'equipaggio
    short unsigned int membri_equipaggio_;
    // Taglia messa sulla nave da parte dell'Impero
    double taglia_sulla_nave_;

    void danniRicevutiDaLaserCB(const sensor_msgs::LaserConstPtr&);

    /*Tanti altri metodi e attributi privati...*/
};
} // starships

millennium_falcon.cpp
```

```
#include "starships/millennium_falcon.h"

namespace starships{

MillenniumFalcon::MillenniumFalcon()
{
    // Definizione del subscriber
    laser_subscriber = nh_.subscribe("incoming_laser", 5,
        MillenniumFalcon::danniRicevutiDaLaserCB, this);
}

/*...altri metodi*/

void MillenniumFalcon::spinner()
{
    ros::spinOnce();
}

}
```

Da notare BENE, nel costruttore, come definisco l'oggetto "laser_subscriber" per far sì che al topic desiderato venga collegata la Callback giusta!!!! La sintassi è la seguente:

```
<nome_subscriber> = <nodeHandle>.subscribe("_topic", <dimensione_buffer>, &<nome_callback>,
this);
```

Il file main che rappresenta il nodo avrà la seguente forma:

```
#include "starships/millennium_falcon.h"

int main(int argc, char** argv)
{
    ros::init(argc, argv, "millennium_node");

    starships::MillenniumFalcon* mf = new starships::MillenniumFalcon();

    while (ros::ok())
        mf->spinner();

    delete mf;

    return 0;
}
```

In questo modo il "main" è ridotto ai minimi termini e tutto il lavoro avviene all'interno dell'oggetto che, all'occasione, potrà essere riutilizzato in un altro nodo assieme ad altri oggetti.

Per quanto riguarda i servizi, la sintassi è molto simile. Una volta dichiarato il servizio nella dichiarazione della classe come attributo (pubblico o privato, importa solo se la classe verrà successivamente estesa) basta iniziarlo nella definizione della classe, collegando il "ServiceServer" alla relativa callback:

```
[...]
// Nella dichiarazione della classe
ros::ServiceServer viaggio_velocita_luce_;
[...]
// Nella definizione della classe
viaggio_velocita_luce_ = nh_.advertiseService("iper_travel",
    &MillenniumFalcon::speedTravel, this);
```

Nel caso volessi utilizzare il contenuto di un topic all'interno di un altro metodo che non sia la callback relativa al topic stesso, basta salvare l'intero topic (o il singolo campo del topic) in un attributo della classe. In questo modo, tale attributo avrà visibilità all'interno di ogni metodo della classe.

3.5.1 Esportare classi da un pacchetto ROS

A questo punto uno potrebbe benissimo dire "Ok, incorporare publisher e subscriber in una classe è utile: il codice risulta essere molto più ordinato e comprensibile. Ma come faccio a renderlo utilizzabile in un altro

pacchetto senza dover richiamare i sorgenti in fase di compilazione?”. Il trucco è molto semplice. Anziché creare un eseguibile, ovvero un nodo ROS, basta esportare la classe come libreria. Per far ciò, basta modificare il file “CMakeLists.txt” nel seguente modo:

```
cmake_minimum_required(VERSION 2.8.3)
project(millennium_falcon)

add_compile_options(-std=c++11)

find_package(catkin REQUIRED COMPONENTS
roscpp
std_msgs
)

catkin_package(
INCLUDE_DIRS include
LIBRARIES millennium_falcon
CATKIN_DEPENDS roscpp std_msgs
)

include_directories(include ${catkin_INCLUDE_DIRS})

add_library(${PROJECT_NAME} src/millennium_falcon.cpp)
target_link_libraries(${PROJECT_NAME} ${CATKIN_LIBRARIES})
```

Se in un nuovo pacchetto volessimo riutilizzare la classe definita nel pacchetto “millennium_falcon” basta aggiungere tale nome quando si va a creare il nuovo pacchetto ROS, es:

```
catkin_create_package limpero_colpisce_ancora roscpp tf millennium_falcon
```

3.5.2 Il server dei parametri di ROS: chi è questo sconosciuto?

Come dovreste già sapere a questo punto grazie ai tutorial ufficiali⁵, i “*launch-file*” di ROS sono FONDAMENTALI quando si gestiscono progetti con tanti nodi. Infatti, i “*launch-file*” permettono di lanciare in esecuzione molteplici nodi in una sola volta, ma non solo: i *launch-file* permettono anche di caricare dei parametri nel “parameter-server” di ROS!!!! Questi parametri potranno poi essere caricati in automatico nei vari nodi in esecuzione. Il vantaggio principale di caricare dei parametri direttamente dal server dei parametri di ROS è quello di NON dover ricompilare il codice ogni volta che si desidera modificare di un millesimo una costante. Ve lo spiego con un piccolo esempio:

```
#include <tutti_gli_header_che_servono>

#ifndef MEMBRI_EQUIPAGGIO
#define MEMBRI_EQUIPAGGIO 2
#endif

class MillenniumFalcon : public MercantileCorellianoYT1300
{
public:
    MillenniumFalcon()
    {
        // Codice...
        membri_equipaggio_ = MEMBRI_EQUIPAGGIO;
        // Altro codice ...
    }
    /*Altri attributi e metodi...*/
private:
    short unsigned int membri_equipaggio_;
    /*Tanti altri metodi e attributi privati...*/
}
```

⁵Non mi stancherò mai di ripeterlo: fate questo dannato tutorial!!!

Nell'esempio sopra, l'attributo privato "membri equipaggio_" viene definito nel costruttore tramite una MACRO di C++. Questo sarebbe utile SE E SOLO SE il numero di membri dell'equipaggio fosse sempre lo stesso. Nel caso qualcun altro salisse a bordo, dovrei aprire il codice sorgente, modificare il valore della MACRO e ricompilare il codice. Considerando il possibile andirivieni di passeggeri, ciò non è per nulla comodo! Ecco una possibile miglitoria del codice per caricare i parametri:

```
#include <tutti_gli_header_che_servono>

class MillenniumFalcon : public MercantileCoreellianoYT1300
{
public:
    MillenniumFalcon()
    {
        ros::NodeHandle private_nh("~");

        private_nh.param("membri equipaggio_", membri equipaggio_, 2);
        nh_.param("taglia imperiale", taglia_sulla_nave_, 0.0);
    }
    /*Altri attributi e metodi...*/
private:
    ros::NodeHandle nh_;
    short unsigned int membri equipaggio_;
    double taglia_sulla_nave_;

    /*Tanti altri metodi e attributi privati...*/
}
```

Nel caso il nodo risultante dalla classe fosse lanciato con "roslaunch", negli attributi saranno caricati i valori di default (membri equipaggio_ = 2, taglia_sulla_nave_ = 0.0). Per caricare dei parametri diversi, occorre caricarli sul server PRIMA di lanciare il nodo con il comando "rosparam set <nome_parametro> <valore>" o in contemporanea al lancio del nodo tramite "roslaunch" con un launch-file simile a questo:

```
<launch>
  <node package="starships_package" type="millennium_falcon" name="solo_ship">
    <param name="membri equipaggio_" type="int" value="3"/>
    <param name="taglia imperiale" type="double" value="300000"/>
  </node>
</launch>
```

In questo modo, il costruttore stesso interrogherà il server dei parametri per caricare i valori inseriti nel launch-file.

Nel caso i parametri da inserire siano troppi (ovvero > 5), è possibile utilizzare caricare da launch-file un file di parametri definito in YAML, come nell'esempio seguente:

```
parsec: 12 # esempio di parametro int
light_speed: 299792458 # esempio di parametro double
name: "darth_vader" # esempio di stringa
droid_names: {a: "R2D2", b: "C3P0"} # esempio di lista di stringhe
```

Per caricare un file "YAML" da launch-file non si usa l'istruzione "param" come nel caso di singoli parametri, ma si usa:

```
<launch>
  <node pkg="millennium_falcon" type="millennium_falcon" name="solo_ship">
    <rosparam command="load"
      file="$(find_millennium_falcon_pkg)/param/millennium_data.yaml"/>
  </node>
</launch>
```

Per accedere ai parametri definiti in liste fate riferimento, ad esempio, all'esempio seguente:

```
...
ros::NodeHandle pnh("~");
if (pnh.getParam("lista"))
{
    XmlRpc::XmlRpcValue list;
```

```

    pnh.getParam("nome_lista", list);
    for (int ii = 0, ii < list.size; ii++)
        parametro[ii] = list[ii];
}
else
{
    ROS_FATAL("Errore!");
    ros::shutdown();
    return;
}

```

Ricordatevi BENE di una cosa di fondamentale importanza:

LE UNITÀ DI MISURA DI ROS SONO QUELLE DEL SISTEMA INTERNAZIONALE!!!⁶

Le unità di misura, per chi avesse bisogno di una ripassatina da Fisica I, sono le seguenti:

- Lunghezza: metri, $[m]$
- Massa: chilogrammi, $[kg]$
- Tempo: secondi, $[s]$
- Corrente: ampere, $[A]$
- Angoli: radianti, $[rad]$
- Frequenza: hertz, $[hz]$
- Forza: newton, $[N]$
- Potenza: watt, $[W]$
- Tensione elettrica: volt, $[V]$
- Campo magnetico: tesla, $[T]$
- Temperatura: celsius (anche se non è l'unità ufficiale del SI, ROS stabilisce che sia questa la sua unità di misura ufficiale per la temperatura...), $[^{\circ}C]$

La chiralità in ROS, segue sempre la regola della mano destra. Solitamente, gli assi dei frame in ROS hanno la seguente configurazione: asse x che punta in avanti nella direzione di avanzamento del robot, asse y che punta a sinistra del robot e asse z che punta verso l'alto.

3.5.3 Ascoltare più voci e usare TF

Probabilmente, se è da un po' di tempo che state programmando in ROS, vi sarà capitato di dover sviluppare un programma che deve accedere a due o più topic in contemporanea. Per far sì che un unico metodo o funzione possa accedere ai contenuti di più topic le vie sono molteplici:

- Salvare il contenuto di un topic in una variabile condivisa (attributo o, peggio, variabile globale). Facile da implementare ma non garantisce la sincronicità;
- Usare i "message_filter" di ROS, cioè sfruttare le librerie Boost, per linkare i topic ed eseguire una callback in base ad una policy di sincronizzazione. Abbastanza complicato ma la sincronicità dei topic viene garantita se è stata scelta la giusta policy di sincronizzazione;
- Nel caso dovessi utilizzare il dato di un sensore e una posizione di un robot, posso sfruttare TF.

⁶dovreste essere degli ingegneri!! Agite e pensate come se lo foste per davvero!

Utilizzare TF per aver accesso alla posizione di un oggetto è estremamente semplice e a costo zero. Occorre solamente includere la libreria TF fra le dipendenze del pacchetto che contiene il nodo in esame e sfruttare un paio di accortezze durante la programmazione della classe. Facciamo un esempio (in questo esempio inserirò la definizione della classe all'interno della dichiarazione della stessa per semplificarci la vita. Voi NON FATELO!):

```
#include <tf/transform_listener.h>
#include <ros/ros.h>

namespace basi_ribelli{

classe BaseHoth
{
public:
    [...]

private:
    // TF listener
    tf::TransformListener listener_;

    // Metodo che restituisce la posizione (x,y) dell'i-esimo ATAT
    std::pair<double, double> leggiPosizioneATAT(int i)
    {
        std::pair<double, double> posizione;
        tf::StampedTransform transform;
        char atat_name[20];
        strcpy(atat_name, "atat_link_");
        strcat(atat_name, std::to_string(i).c_str());
        try
        {
            listener_.waitForTransform("map", atat_name, ros::Time(0),
                                     ros::Duration(0.2));
            listener_.lookupTransform("map", atat_name, ros::Time(0),
                                     transform);
            posizione.first = transform.getOrigin().getX();
            posizione.second = transform.getOrigin().getY();
        }
        catch (tf::TransformException &ex)
        {
            ROS_ERROR("L' ATAT_%d non esiste ancora!", i);
            posizione.first = INF;
            posizione.second = INF;
        }

        return(posizione);
    }
};
}
```

In questo modo, la classe `BaseHoth` ha accesso alla posizione di un ATAT senza dover per forza sottoscriverne al topic.

Come potete vedere dall'esempio precedente, l'uso di TF facilita ENORMEMENTE la programmazione di applicazioni "general-purpose". In un sistema ROS, ogni sensore, robot o oggetto di interesse dovrebbe essere associato al proprio frame. In questo modo, nel caso aveste bisogno di accedere a qualche posizione, basta creare un "tf::TransformListener" per accedere alla posizione desiderata senza doversi complicare la vita sottoscrivendo a N topic e creando altrettante callback⁷.

Nel caso, invece, vogliate usare per forza i `message_filter` di ROS, potete fare riferimento al seguente esempio; anche in questo caso inserisco dichiarazione e definizione nello stesso file:

```
#include <ros/ros.h>
#include <ros/callback_queue.h>
```

⁷Come si fa a fare il procedimento inverso, ovvero a pubblicare un frame? Se non lo sapete siete dei somari, poiché è scritto nel <http://wiki.ros.org/tf/Tutorials> che voi dovrete aver studiato in maniera approfondita.

```

#include <message_filters/subscriber.h>
#include <message_filters/synchronizer.h>
#include <message_filters/time_synchronizer.h>
#include <message_filters/sync_policies/approximate_time.h>
#include <nav_msgs/Odometry.h>
#include <geometry_msgs/Twist.h>

namespace starships {

class MillenniumFalcon
{
public:
    MillenniumFalcon()
    {
        [...]

        tie_fighter_pose_sub_.subscribe(nh_, "tie_fighter/pose", 1);
        tie_fighter_speed_sub_.subscribe(nh_, "tie_fighter/speed", 1);

        sync_.reset(new Sincronizzatore(Regola_di_sincronizzazione(10),
            tie_fighter_pose_sub_, tie_fighter_speed_sub_));
        sync_->registerCallback(boost::bind(&MillenniumFalcon::aimAndShot,
            this, _1, _2));
        nh_.setCallbackQueue(&coda_delle_callback_);
    };

    ~MillenniumFalcon(){};

private:
    ros::NodeHandle nh_;

    ros::CallbackQueue coda_delle_callback_;

    typedef sync_policies::ApproximateTime<nav_msgs::Odometry,
        geometry_msgs::Twist> Regola_di_sincronizzazione;

    typedef message_filters::Synchronizer<Regola_di_sincronizzazione>
        Sincronizzatore;

    boost::shared_ptr<Sincronizzatore> sync_;

    message_filters::Subscriber<nav_msgs::Odometry> tie_fighter_pose_sub_;

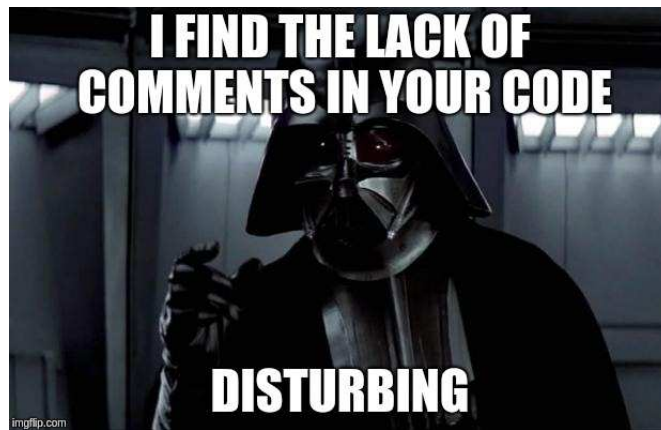
    message_filters::Subscriber<geometry_msgs::Twist> tie_fighter_speed_sub_;

    void aimAndShot(const nav_msgs::OdometryConstPtr& pose,
        const geometry_msgs::TwistConstPtr& speed){[...]};
};
} // starships

```

Nell'esempio, la classe “MillenniumFalcon” sottoscrive sia alla posa che alla velocità con una regola di sincronizzazione approssimata “sync_policies::ApproximateTime”. Nel costruttore della classe, la regola di approssimazione viene settata a 10 *ms*, e viene settata come callback il metodo “aimAndShot”. Questo significa che una volta ricevuto un topic il sincronizzatore lo manterrà per almeno 10 *ms* in attesa del secondo topic, dopodiché lo scarcerà.

4 Commentare il codice per evitare maledizioni



Cercherò di essere molto breve in questa sezione: una volta che tu, geniale programmatore, avrai completato di scrivere il tuo codice che funzionerà efficientemente in maniera impeccabile, eseguendo perfettamente quello per cui è stato progettato, ti verrà da dire: “*Fantastico! Ora il mio lavoro è finito! Posso tornare a procrastinare!*”. ERRORE!!!! Dopo aver concluso le fasi di “coding” e “testing” del software, DOVETE INSERIRE I COMMENTI!! Sappiate che all’inferno esiste un girone apposta per coloro che lasciano il codice senza commenti!

Detto ciò, passiamo alle cose serie. Come si fa a commentare il codice? In C++ per commentare una riga di codice basta aggiungere il doppio slash “//” all’inizio della riga. In alternativa, per creare un blocco di commenti, si può usare “/*” per aprire il blocco e “*/” per chiuderlo.

Il problema, però, non è come fare ad aggiungere commenti, ma bensì “come commentare il codice in modo che un altro programmatore capisca come riusare il mio codice o, se necessario, modificarlo”. Questo non significa che dovrete commentare ogni singola riga del codice, in quanto ciò sarebbe più dannoso che utile. Per commentare in maniera corretta il codice occorre usare la testa e usare lo stesso impegno che si usa quando si scrive il codice stesso. Ecco alcune dritte su cosa commentare:

- Ogni variabile dovrebbe avere un commento che ne spiega lo scopo, a meno che ciò non sia di spiegazione immediata, come ad esempio i contatori all’interno dei cicli;
- Ogni attributo di una classe deve avere una spiegazione sulla propria natura e sul proprio utilizzo;
- Ogni funzione o metodo deve avere una spiegazione dettagliata sul funzionamento, su cosa sono le variabili di input e su cosa restituisce;
- Ogni MACRO deve avere un commento che ne descriva lo scopo;
- Ogni oggetto (inteso come classe, struct, union, dati di tipo enum, typedef, ...) deve avere una descrizione sulla sua natura e/o utilizzo;
- Ogni macro-blocco di codice la cui interpretazione non è immediata deve avere una descrizione di cosa esegue.

La scrittura corretta dei commenti è utile non solo per gli altri che leggeranno il vostro codice, ma è utile a chi scrive il codice per tenere traccia di ciò che fa nel caso dovesse riprendere in mano del codice scritto nel passato e anche per scrivere la documentazione del codice stesso!

Esistono infatti dei tool appositi per generare in automatico la documentazione del codice a partire dai commenti. Uno di questi è Doxygen. Se scrivete i vostri commenti in stile Doxygen, le cui linee guida le trovate qui, potete sfruttare ROSDOC per generare la documentazione. Ad esempio, create una cartella “/doc” all’interno del pacchetto. Lanciando il comando dalla cartella “/src” del proprio ROS-workspace:

```
rostdoc_lite millennium_falcon -o millennium_falcon/doc
```

verrà creato in automatico un documento ipertestuale in formato “HTML” con tutta la documentazione del codice scritto. Per visualizzarlo, basta aprire il file “index.html” generato con un browser web.

Propongo un esempio di codice commentato in stile Doxygen:

```
#include <...>

/// Namespace including all the starships
namespace starships {

/**
 * \brief The MillenniumFalcon class
 * This class implements the marvellous, hyper-fats and amazing
 * Han Solo's ship
 */
class MillenniumFalcon {
public:
    /**
     * \brief Constructor
     */
    MillenniumFalcon()
    {...};

    /**
     * \brief Destructor
     */
    virtual ~MillenniumFalcon()
    {...};

    /// Actual speed of the starship
    double actual_speed;

private:

    /// Number of passengers
    int number_of_passengers_;

    /**
     * \brief Activates light speed
     * This method computes the route for a light speed travel into
     * space, given the coordinates of the destination planet
     * @param coordinates Destination coordinates
     * @return True if it was possible to activate the light speed.
     * Otherwise, it returns false
     */
    bool activateLightSpeed(std::vector<double> coordinates)
    {
        [...]
    }
};

} // starships
```

Per creare un commento compatibile con Doxygen basta aprire i commenti su una linea con 3 “*slash*” anziché 2, oppure, per i blocchi di commenti utilizzare “*/***” per aprire e “**/*” per chiudere. Vi invito nuovamente a leggere il tutorial di Doxygen disponibile a questo indirizzo.

PS: oltre ai commenti, per rendere più leggibile il codice, il programma va anche **INDENTATO!!!**

5 Git e Version Control

Un version control system, abbreviato in VCS, è un programma che permette di tener traccia di tutte le modifiche e le evoluzioni effettuate nel corso della stesura di un codice o di un qualsiasi progetto su supporto digitale. Un software VCS permette di mantenere una copia del proprio codice sorgente, sia in locale sia in remoto, senza incorrere in un eccessivo dispendio di energie e senza deconcentrarsi eccessivamente dalla stesura del proprio testo. In linea di principio un VCS permette di tenere sotto controllo qualsiasi documento, siano essi foto, documenti realizzati con programma di videoscrittura, fogli di calcolo, ecc. . . ma un VCS dà il meglio di sé con i file di testo semplice, non formattato, poiché permette di vedere tutte le modifiche apportate di volta in volta al file. Proprio per questo l'uso principale dei VCS è quello di controllare lo sviluppo dei software.

L'insieme dei file e cartelle facenti parte di un progetto controllato da un VCS, comprendente tutta la cronologia delle modifiche, si chiama repository. L'operazione di registrazione di una modifica di uno o più file all'interno repository si chiama commit.

I VCS si dividono in due grandi classi: sistemi centralizzati e sistemi distribuiti. I VCS più vecchi sono di tipo centralizzato (centralized version control system, CVCS), cioè il repository principale si trova su un server remoto centrale a cui tutti gli utenti che vogliono utilizzarlo devono fare riferimento. Con il passare del tempo ci si è resi conto che questo meccanismo pone pesanti limiti, perché ogni operazione legata al VCS (controllare la cronologia, registrare una modifica, ecc. . .) deve necessariamente essere fatta in un momento in cui si dispone di una connessione alla rete in cui si trova il server centrale. Inoltre questo comporta dei tempi relativamente lunghi per alcune operazioni semplici e frequenti a causa al fatto che è necessario inviare attraverso la rete la richiesta di una certa operazione al server, il quale elabora la richiesta e invia al richiedente la risposta nuovamente attraverso la rete. Quando negli anni passati le connessioni a Internet non raggiungevano le velocità attuali era spesso tedioso aspettare tanto tempo per ottenere l'output di queste operazioni. Dunque in un CVCS si è completamente dipendenti dal server centrale che contiene il repository del progetto, con l'ulteriore conseguenza che se il server remoto viene spento dal suo gestore o compromesso da malintenzionati tutto il lavoro può essere perduto. Per questi e altri motivi si sono diffusi in seguito i sistemi distribuiti (distributed version control system, DVCS), nei quali ogni utente del progetto ha una copia dell'intero repository sul proprio sistema e, se lo si desidera, è possibile inviare una copia del proprio repository su un server remoto per metterlo in condivisione con altre persone. Un DVCS può quindi riprodurre il flusso di lavoro di un CVCS ma senza avere i suoi aspetti negativi legati alle limitazioni di dover lavorare necessariamente con l'unico server centrale.

Fra i più famosi CVCS ricordiamo Concurrent Versions System (abbreviato in CVS) e Subversion (SVN), i DVCS più noti invece sono Bazaar (bzd), Git e Mercurial (hg).

Si comincerà col descrivere il modello dati di Git, cioè il repository. Da qui si descriveranno le varie operazioni che Git fornisce per l'elaborazione del repository, cominciando dalla più semplice (aggiunta di dati ad un repository) passando progressivamente verso le operazioni più complesse di diramazione (branching) e fusione (merging). Dopodiché si discuterà dell'uso di Git in un contesto collaborativo e, infine, si analizzerà la funzione di rebase di Git, che fornisce un'alternativa al merging, esponendone i pro e i contro.

5.0.1 Repository

Lo scopo di Git (come ogni sistema di controllo versione) è la gestione di un progetto, o un'insieme di file, che cambiano nel tempo. Git memorizza queste informazioni in una struttura dati chiamata repository.

Un repository Git contiene, insieme ad altre cose, i seguenti elementi:

- Un insieme di oggetti commit;
- Un insieme di riferimenti a oggetti commit, chiamati intestazioni (o head)

Il repository Git viene memorizzato nella stessa directory del progetto, in una sottodirectory di nome `“.git”`. Esiste **una sola** directory `.git`, nella directory radice del progetto.

5.0.2 Commit

Un **oggetto “commit”** contiene tre cose:

- Un insieme di **file** che riflette lo stato di un progetto in un dato punto nel tempo;
- Uno o più riferimenti agli **oggetti commit genitori**;
- Un nome **“SHA1”**, cioè una stringa di 40 caratteri alfanumerici, che identifica univocamente l'oggetto commit. Il nome viene creato componendo un hash (una sorta di checksum) di aspetti rilevanti del commit, per cui commit identici avranno lo stesso nome.

Gli oggetti commit genitori sono quei commit che sono stati modificati per produrre lo stato successivo del progetto. In generale un oggetto commit avrà un commit genitore, dato che generalmente si prende il progetto in un dato stato, si fanno alcuni cambiamenti, e si salva il nuovo stato del progetto.

Un progetto possiede sempre un oggetto commit senza genitori. Questo è il primo commit eseguito sul repository del progetto.

È possibile immaginare un repository come un grafo aciclico diretto di oggetti commit, con puntatori ai commit genitori che puntano sempre indietro nel tempo, fino al primo commit. Partendo da qualsiasi commit, si può camminare lungo l'albero, scorrendo i commit genitori, per vedere la cronistoria dei cambiamenti che portano a quel particolare commit.

L'idea che sta dietro a Git è che il controllo versione consiste nella elaborazione di questo grafo di commit. Ogniqualevolta si voglia eseguire una qualche operazione per interrogare o modificare il repository, si può pensare l'operazione in questi termini: "in che modo voglio interrogare o modificare il grafo dei commit?".

5.0.3 Intestazioni

Una **intestazione** (o **head**) è semplicemente un riferimento ad un oggetto commit. Ogni intestazione possiede un nome. Come impostazione predefinita, c'è una intestazione in ogni repository chiamata master. Un repository può contenere qualsiasi numero di intestazioni. In ogni istante, c'è una intestazione selezionata come "intestazione corrente". Questa intestazione ha come alias HEAD, sempre scritto in maiuscolo. Si noti questa differenza: una "head" (minuscolo) si riferisce ad una qualsiasi delle intestazioni con nome presenti nel repository; "HEAD" (maiuscolo) si riferisce esclusivamente alla intestazione correntemente attiva. Questa distinzione viene menzionata spesso nella documentazione di Git. In questo testo si userà anche la convenzione di impostare in nomi delle intestazioni, inclusa *HEAD*, in corsivo.

5.0.4 Il primo repository

Per creare un repository, create una directory per il progetto, se questa non esiste già, entratevi, ed eseguite il comando `git init`. La directory non deve necessariamente essere vuota.

Esempio:

- `mkdir <nome_cartella>`
- `cd <nome_cartella>`
- `git init`

Ciò creerà una nuova directory `.git` nella directory di progetto `<nome_cartella>`

Per creare un commit, bisogna fare due cose:

1. Dire a Git quali file includere nel commit, con il comando `git add` (aggiungi). Se un file non è cambiato dal commit precedente (il commit "genitore"), Git lo includerà automaticamente nel commit che si sta effettuando. Perciò sarà necessario aggiungere solo file che sono stati aggiunti o modificati. Notare che il comando aggiunge ricorsivamente le directory, per cui il comando `git add .` o il comando `git add -A` aggiungeranno ogni cosa cambiata. Per evitare che Git aggiunga file indesiderati è possibile aggiungere alla cartella del nostro progetto un file `.gitignore`, ma di questo parleremo nella prossima sottosezione.
2. Chiamare `git commit` per creare l'oggetto commit. Il nuovo oggetto commit avrà HEAD come suo genitore. Il comando necessita anche di un messaggio di commit, che può essere aggiunto successivamente al comando, quando Git aprirà in automatico un editor di testo o scrivendolo esplicitamente con il comando `git commit -m "messaggio"`

Come scorciatoia è possibile utilizzare il comando `"git commit -a"`, che aggiungerà automaticamente TUTTI i file modificati ma non quelli nuovi.

Da notare che se si modifica un file ma non lo si aggiunge, Git includerà tutte le versioni precedenti (prima della modifica) al commit. Il file modificato rimarrà al suo posto.

Consideriamo il seguente **esempio**: dopo tre commit il repository avrà questo aspetto:

```
---> tempo --->
(A) <-- (B) <-- (C)
      |
      master
      |
      HEAD
```

dove (A), (B) e (C) sono rispettivamente il primo, il secondo e il terzo commit in ordine temporale.

Altri comandi utili sono:

- **git log**: mostra il registro di tutti i commit cominciando da **HEAD** sino al commit iniziale;
- **git status**: mostra quali file sono cambiati tra lo stato corrente del progetto e **HEAD**. I file vengono divisi in tre categorie:
 - nuovi file che non sono stati aggiunti
 - vecchi file che sono stati modificati ma non sono stati aggiunti
 - file che sono stati aggiunti (vecchi e nuovi)
- **git diff**: mostra le differenze tra **HEAD** e lo stato corrente del progetto. Aggiungendo l'opzione **--cached** vengono confrontati i file aggiunti con **HEAD**, altrimenti vengono confrontati i file non ancora aggiunti;
- **git mv**: marca i file da spostare o rinominare (simile all'istruzione bash di Linux **"mv"**);
- **git rm**: marca i file per la rimozione (simile all'istruzione bash di Linux **"rm"**).

Una tipica sequenza di lavoro potrebbe essere questa:

1. Coding & Debugging
2. **git status** per controllare i file modificati
3. **git diff [file]** per mostrare le modifiche effettuate
4. **git commit -a -m [messaggio]** per fare il commit

Per fare riferimento ad un commit specifico esistono diversi modi:

- Dal nome "SHA1" ottenibile tramite il comando **"git log"**;
- Dai primi caratteri del nome "SHA1";
- Per intestazione. Per esempio, *HEAD* si riferisce all'oggetto commit riferito da *HEAD*. Si può usare anche il nome, come ad esempio *"MASTER"*;
- Relativamente ad un commit. Ponendo il carattere **"^"** (accento circonflesso) dopo il nome del commit si può recuperare il genitore di quel commit. Ad esempio, **"HEAD^"** è il genitore dell'intestazione del commit corrente.

5.0.5 .gitignore

Il file **.gitignore** è un semplice file di testo nel quale vengono elencati i file dei quali non si vuole tenere traccia nella storia del repository. **.gitignore** interpreta in maniera del tutto automatica il nome dei file e le "wildcard" (indicate con il carattere speciale asterisco **"*"**). Ad esempio, se noi aggiungiamo ***.exe** al file **.gitignore**, Git, in automatico, ignorerà tutti i file con estensione **".exe"** che si trovano nella cartella del nostro progetto. Nel caso volessimo dire a Git di ignorare una cartella contenuta nella directory del progetto, basta aggiungere al file **.gitignore** la stringa **/nome_cartella**.

5.0.6 Branching

Git permette di spostare l'evoluzione di un progetto (e di un repository) dal ramo principale (main branch o master) su altri rami **"branch"** per, ad esempio, sviluppare particolari feature isolate l'una dall'altra o per creare versioni ad-hoc di certi software con varianti particolari richieste da diversi committenti. Il comando per creare un nuovo branch di nome **"feature_x"** è:

```
git checkout -b feature_x
```

In questo modo viene creato il nuovo branch e l'utente può da subito iniziare a lavorare sulla nuova feature, dato che il precedente comando sposta già il riferimento di git al nuovo branch.

Dopo aver finito di implementare una nuova caratteristica in una diramazione del progetto (branch), si potrebbe desiderare di riportare tale caratteristica nel ramo principale (main) del progetto, in modo tale che tutti possano usarla. Ciò può essere eseguito con i comandi **git merge** o **git pull**. La sintassi è la seguente:

- **git merge [intestazione]**
- **git pull . [intestazione]**

Il risultato è identico (malgrado il comando con `merge` sembri, per ora, la versione più semplice, la ragione dell'esistenza della versione con `pull` sarà evidente quando si parlerà dell'uso con più sviluppatori in contemporanea).

Questi comandi eseguono le seguenti operazioni. Chiamiamo l'intestazione corrente *current*, e quella da fondere *merge*.

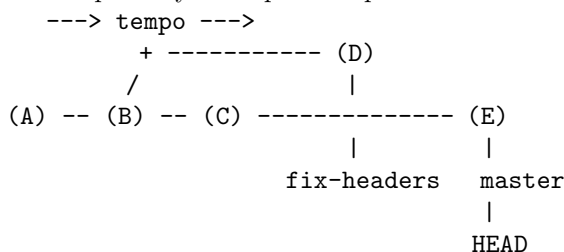
1. Identificare l'antenato comune di *current* e *merge*. Chiamiamolo *commit-antenato*.
2. Partendo dalle cose più semplici, se *commit-antenato* è uguale a *merge*, allora non serve fare nulla. Se *commit-antenato* è uguale a *current*, allora è necessario eseguire un **fast forward merge**.
3. Altrimenti, determinare i cambiamenti tra *commit-antenato* e *merge*.
4. Cercare di fondere tali cambiamenti nei file in *current*.
5. Se non ci sono conflitti, creare un nuovo commit, con due genitori, *current* e *merge*. Impostare *current* (e HEAD) in modo da puntare a questo nuovo commit, e aggiornare i file di lavoro del progetto in modo appropriato.
6. Se c'era un conflitto, inserire gli appropriati marcatori di conflitto e informare l'utente. Non verrà creato nessun commit.

IMPORTANTE

Quando si esegue una fusione, Git può confondersi molto, se ci sono cambiamenti di cui non sono stati fatti i commit dei file. Perciò è meglio assicurarsi di fare il commit di ogni cambiamento effettuato **prima** di effettuare una fusione.

Prendendo come riferimento l'esempio nella sezione “**Il mio primo repository**”, lo estendiamo portando fuori l'intestazione *master* nuovamente e finire di modificare i file del progetto. Poi reimportiamo tali cambiamenti che si erano effettuati nelle intestazioni (fix-headers).

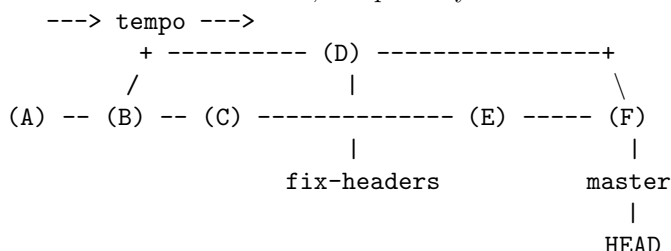
Il repository avrà questo aspetto:



dove (E) è il commit che riflette la versione completa con le ultime modifiche ai file. A questo punto si esegue:

```
git merge fix-headers
```

Se non ci sono conflitti, il repository ottenuto sarà:



Il commit merge è (F), che ha come genitori (D) e (E). Dato che (B) è l'antenato comune tra (D) e (E), i file in (F) dovrebbero contenere i cambiamenti tra (B) e (D), cioè le correzioni delle intestazioni, incorporate nei file da (E).

Nota sulla terminologia: quando si dice “fondere (merge) intestazione A in intestazione B,” si intende che intestazione B è la intestazione corrente, e che si sta traendo i cambiamenti dalla intestazione A in essa. Intestazione B viene aggiornata; nulla viene effettuato su intestazione A (se si sostituisce la parola “fusione” con la parola “estrazione” si rende più il senso dell'operazione).

5.0.7 Risoluzione dei conflitti

Un conflitto si verifica se il commit da fondere ha un cambiamento in esso in una determinata posizione, e il commit corrente possiede un cambiamento nella stessa posizione. Git non ha modo di sapere quale dei due cambiamenti possa avere la precedenza.

Per risolvere il commit, modificare i file per riparare i cambiamenti in conflitto. Poi eseguire `git add` per aggiungere i file con la soluzione, ed eseguire `git commit` per fare il commit della fusione riparata. Git è in grado di ricordare di essere stato interrotto in mezzo ad una fusione, perciò imposta correttamente i genitori del commit.

Nota sul branching: Le più frequenti ragioni per fondere due “diramazioni” (branch) dello sviluppo sono due. La prima, come spiegato poc’anzi, è per estrarre i cambiamenti da una diramazione, creata per esempio per sviluppare e testare delle nuove caratteristiche, e trasportarli nel ramo principale. Il secondo uso è per trasportare il ramo principale nel ramo di test che si sta sviluppando. Ciò mantiene il ramo di test aggiornato con gli ultimi aggiornamenti di riparazione bug e di aggiunta di nuove caratteristiche, aggiunti al ramo principale. Facendo ciò con cadenza regolare, si riduce il rischio di creare un conflitto, che solitamente emerge quando si voglia poi fondere le nuove caratteristiche in test, dentro il ramo principale. Uno svantaggio di queste procedure è che il ramo di test finirà con l’avere molti commit di fusione. Una soluzione alternativa di questo problema è il rebasing, malgrado non sia scevro da altri problemi a sua volta.

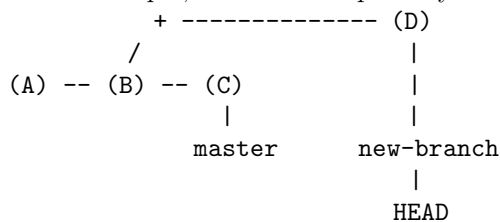
5.0.8 Rebasing

In alternativa al “merge” di due diversi branch, che come abbiamo visto può presentare parecchie difficoltà e rischia di rendere difficilmente leggibile la “history” del repository, Git offre una via alternativa per eseguire la fusione di due rami: il **rebase**. La sintassi è la seguente:

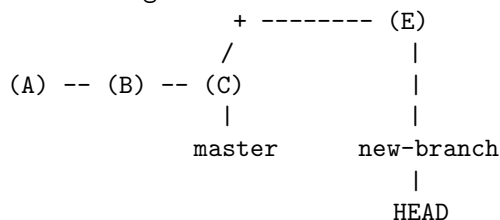
```
git rebase [nome-branch OR commit-ID]
```

Questo comando prende una serie di commit (di solito da un branch) e li riproduce in cima ad un altro commit (normalmente l’ultimo commit in un altro branch), ricalcolando gli ID dei commit.

Ad esempio, se il nostro repository ha la seguente struttura



il comando `git rebase master` lo trasformerà nella seguente



dove (E) è un nuovo commit che incorpora tutte le modifiche intercorse fra il commit (B) e (D) ma riorganizzati per comprendere anche i cambiamenti in (C).

5.1 Collaborare con Git

5.1.1 Al lavoro con altri repository

Una caratteristica chiave di Git è che il repository è memorizzato a fianco alle copie di lavoro dei file sotto controllo. Il vantaggio di ciò consiste nel fatto che il repository memorizza l’intera cronologia del progetto, e che Git può funzionare senza bisogno di collegarsi ad un server esterno. Ciò significa che, se si vuole gestire il repository, è necessario avere anche l’accesso ai file di lavoro. Di conseguenza, due sviluppatori che usano Git non possono, come impostazione predefinita, condividere lo stesso repository. Per condividere il lavoro tra più sviluppatori, Git usa un modello distribuito di controllo versione. Esso assume che non ci sia un repository centrale. È possibile, naturalmente, usare un repository come “centrale”, ma è necessario comprendere il modello distribuito su cui è basato.

5.1.2 Sistema di controllo versione distribuito

Poniamo che si voglia lavorare assieme ad un collega su uno stesso documento. Il collega ha già effettuato del lavoro su di esso. Ci sono tre compiti su cui bisogna riflettere perché ciò sia possibile:

1. Come ottenere dal collega una copia aggiornata del lavoro;
2. Come inserire i cambiamenti effettuati dal collega nel proprio repository;
3. Come aggiornare il collega sui propri cambiamenti apportati al software.

Git è fornito di un insieme di protocolli di trasporto per la condivisione delle informazioni sul repository, come SSH e HTTP. Il metodo più semplice (che si può usare come test) è comunque accedere simultaneamente ad entrambi i repository dallo stesso filesystem.

Ogni protocollo viene identificato da una “specifica-remota”. Per i repository presenti sullo stesso filesystem, la specifica-remota è semplicemente il percorso necessario per raggiungere l’altro repository.

5.1.3 Copia del repository

Per fare una copia del repository del collega per usarlo personalmente, usare il comando

```
git clone [specifica-remota]
```

La *[specifica-remota]* identifica la posizione del repository del collega, che può anche essere un’altra cartella nel filesystem. Qualora il repository del collega fosse accessibile tramite un protocollo di rete come ssh, Git si riferisce ad esso tramite un URL del tipo “ssh://server/repository”.

Per esempio, se il repository del collega è posizionato in *ssh://jdserver/jdrepo*, il comando sarà:

```
git clone ssh://jdserver/jdrepo
```

Ciò effettuerà le seguenti operazioni:

1. Creazione di una directory *jdrepo* e inizializzazione di un repository in essa;
2. Copia di tutti gli oggetti commit e riferimenti alle intestazioni prelevandoli dal progetto originale e importandoli nel novo repository locale;
3. Aggiunta di un riferimento a repository remoto di nome *origin* nel nuovo repository e associazione di *origin* con *ssh://jdserver/jdrepo*;
4. Aggiunta di intestazioni remote di nome *origin/[nome-intestazione]* che corrispondono alle intestazioni nel repository remoto;
5. Impostazione di una intestazione nel repository per tenere traccia della corrispondente intestazione *origin/[nome-intestazione-corrente]*, cioè quella che era attualmente attiva nel repository clonato.

Un riferimento repository remoto è un nome che Git usa per riferirsi al repository remoto. Generalmente esso sarà *origin*. Tra l’altro, Git associa internamente la specifica-remota con il riferimento repository remoto, in modo tale da non aver mai bisogno di riferirsi nuovamente al repository originale.

Una diramazione che traccia una diramazione remota, mantiene un riferimento interno alla diramazione remota. Questa semplificazione, come descritto sotto, in molte situazioni permette di evitare di battere il nome del repository remoto.

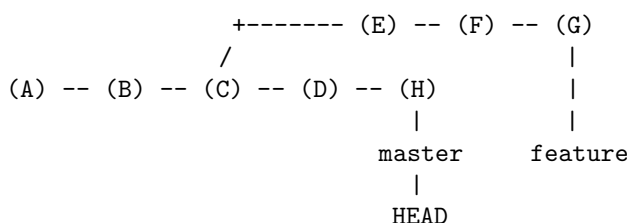
Il fatto importante da notare è che ora si possiede una copia completa dell’intero repository del collega. Quando si dirama, commit, fonde, o si opera in altro modo sul repository, si opera solo sul proprio repository. Git interagisce con il repository remoto solamente quando glielo si chiede esplicitamente.

5.1.4 Ricezione di cambiamenti dal repository remoto

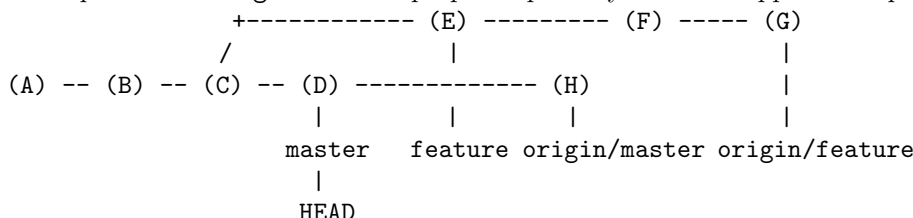
Dopo aver clonato il repository, il collega aggiunge dei commit al suo repository. Come ottenere le copie di questi cambiamenti?

Il comando `git fetch [riferimento-repository-remoto]` recupera i nuovi oggetti commit del repository del collega e crea e/o aggiorna le intestazioni remote di conseguenza. Come impostazione predefinita, il *[riferimento-repository-remoto]* è *origin*.

Poniamo ad esempio che il repository del collega appaia con questa struttura:



Dopo il comando `git fetch` il proprio repository dovrebbe apparire in questo modo:



Si noti che le proprie intestazioni non sono state modificate. L'unica differenza è che le intestazioni remote, quelle che cominciano con "origin/", sono state aggiornate insieme all'aggiunta dei nuovi oggetti commit.

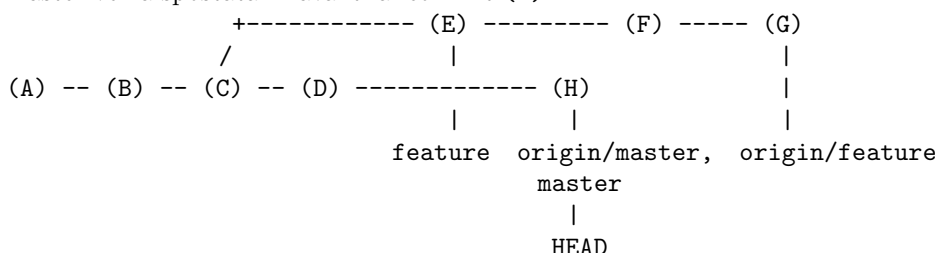
Ora si vuole aggiornare le proprie intestazioni *master* e *feature* per riflettere i cambiamenti apportati dal collega. Questo si fa tramite una fusione, di solito materialmente con un comando `git pull`. La sintassi generica è:

```
git pull [riferimento-repository-remoto] [nome-intestazione-remota]
```

Ciò fonderà l'intestazione di nome *[riferimento-repository-remoto]/[nome-intestazione-remota]* in *HEAD*.

Git possiede due caratteristiche che rendono l'operazione più semplice. La prima consiste nel fatto che, se una intestazione viene impostata tracciante, un semplice `git pull` senza argomenti fonderà la corretta intestazione remota. La seconda è che `git pull` eseguirà automaticamente un `fetch`, perciò sarà difficilmente necessario eseguire singolarmente `git fetch`.

Perciò, riferendosi all'ultimo esempio, effettuando un `git pull` sul proprio repository, la propria intestazione *master* verrà spostata in avanti al commit (H):



5.1.5 Invio di cambiamenti al repository remoto

Ponendo che si voglia modificare il proprio repository, si vorrà successivamente spedire questi cambiamenti nel repository del collega. Il comando `git push`, che esegue ovviamente l'operazione opposta di `git pull`, spedisce i dati al server remoto. La sua sintassi completa è la seguente:

```
git push [riferimento-repository-remoto] [nome-intestazione-remota]
```

Quando il comando viene invocato, Git richiede che il repository remoto faccia le operazioni seguenti:

1. Aggiunta di nuovi oggetti commit al repository remoto
2. Impostazione di *[nome-intestazione-remota]* in maniera da puntare allo stesso commit che punta sul repository inviante

Sul repository inviante, Git aggiorna anche il riferimento alla corrispondente intestazione remota.

Se non vengono aggiunti argomenti al comando `git push`, esso invierà tutte le diramazioni nel repository impostate come traccianti.

5.1.6 Tag

Sostanzialmente in Git con il termine "taggare" intendiamo un'operazione con la quale creare dei collegamenti a determinati stati di avanzamento di un progetto, si tratta solitamente di passaggi cruciali per la sua evoluzione come per esempio il completamento di una prima stabile, la classica versione "1.0" di un'applicazione, l'aggiornamento di un ramo di sviluppo ("1.5.0", "1.5.1" ..), o di una successiva major release ("2.0", "3.0" ..).

I tag possono essere associati ad un numero variabile di informazioni, quelli più articolati rappresentano delle vere e proprie "annotazioni"; l'esempio seguente mostra la creazione di un tag tramite un'istruzione formata dal comando `git tag` seguita dal nome del tag introdotto dall'opzione `-a` e da un messaggio, a commento

dell'operazione, definito tramite l'opzione -m:

```
git tag -a 4.4.1.1 -m 'Aggiornamento a versione 4.1.1'
```

Per visualizzare l'elenco dei tag basta eseguire il comando `git tag`. In questo modo verranno visualizzati TUTTI i tag creati. Per visualizzare solo le versioni con un determinato numero (nell'esempio proposto qui di seguito 4) eseguire il comando:

```
git tag -l 4*
```

Nel caso si volessero visualizzare altre informazioni a corredo di un tag, bisogna ricorrere all'opzione `show`, ad esempio:

```
git show 4.4.1.1
```

Le procedure per la condivisione di risorse sui server remoti non prevedono anche il trasferimento automatico dei tag. A ciò si può ovviare specificando il nome del tag che si vuole condividere sul server remoto direttamente durante l'operazione di `push`:

```
git push origin 4.4.1.1
```

Le procedure di condivisione dei tag possono essere anche cumulative, passando l'opzione `--tags` al comando `git push origin` si avrà infatti la possibilità di condividere in remoto tutti i tag di un progetto fino ad ora disponibili soltanto per chi li ha definiti, senza la necessità di specificarne i singoli nomi:

```
git push origin --tags
```

Così come possono essere creati, i tag possono essere anche cancellati, per far questo di dovrà passare al comando `git tag` l'opzione `-d` seguita dal nome del tag da rimuovere:

```
git tag -d 4.4.1.1
```

Nel caso in cui un tag sia stato condiviso su server remoto si dovrà procedere anche alla rimozione da quest'ultimo operando tramite il comando `git push origin` seguito dalla stringa `:refs/tags/nome-del-tag`, ad esempio:

```
git push origin :refs/tags/4.4.1.1
```

5.2 Bitbucket

Bitbucket (<https://bitbucket.org>) è un servizio di hosting web-based per progetti che usano i sistemi di controllo versione Mercurial (sin dal lancio) o Git (dall'ottobre 2011). Bitbucket offre sia piani commerciali che account gratuiti. Esso offre account gratuiti su un numero illimitato di repository private (che possono avere fino a cinque utenti nel caso di account gratuiti) dal settembre 2010, ma invitando tre utenti a unirsi a Bitbucket, altri tre utenti possono essere aggiunti, per un totale di otto utenti. Bitbucket è scritto in Python usando il Framework per applicazioni web Django. Bitbucket è un prodotto offerto da Atlassian.

Per poter utilizzare i servizi offerti da Bitbucket è necessario registrarsi.

5.2.1 Creare un repository remoto su Bitbucket

Una volta registrati è possibile creare un nuovo repository cliccando sull'icona "+" posta sulla sinistra (vedi fig. 2) e selezionare "Repository".

Comparirà una nuova finestra (vedi Fig. 3) dove sarà possibile inserire il nome del repository (che dovrà essere uguale al repository che si possiede localmente sul proprio computer), il sistema di version control (dato che questo tutorial è su Git, selezionate "Git"), il livello di accesso (pubblico o privato), il sistema di gestione del progetto (ad esempio l'issue tracking di cui parleremo dopo in maniera approfondita) e il linguaggio del software (C, C++, C#, L^AT_EX, Java, ecc ...).

Una volta creato il repository remoto, occorre collegarlo al repository locale presente sul proprio computer. Per fare ciò, aprire un terminale, navigare fino alla cartella del proprio repository (inizializzato precedentemente tramite il comando `git init`) e digitare i comandi:

```
git remote set-url origin [riferimento-repository-remoto]
```

```
git push -u origin master
```

NB: i precedenti comandi sono suggeriti direttamente da Bitbucket, quindi non è necessario ricordarli!

5.2.2 README.md

È buona norma creare sempre un file *README.md* nella root del proprio repository per descriverne il contenuto, elencarne le eventuali dipendenze, spiegarne l'installazione, ecc...

Il file è in formato Markdown language(.md): Markdown è un linguaggio di markup con una sintassi del testo semplice progettata in modo che possa essere convertita in HTML e in molti altri formati usando un tool omonimo. Una guida pratica al Markdown language può essere trovata qui.

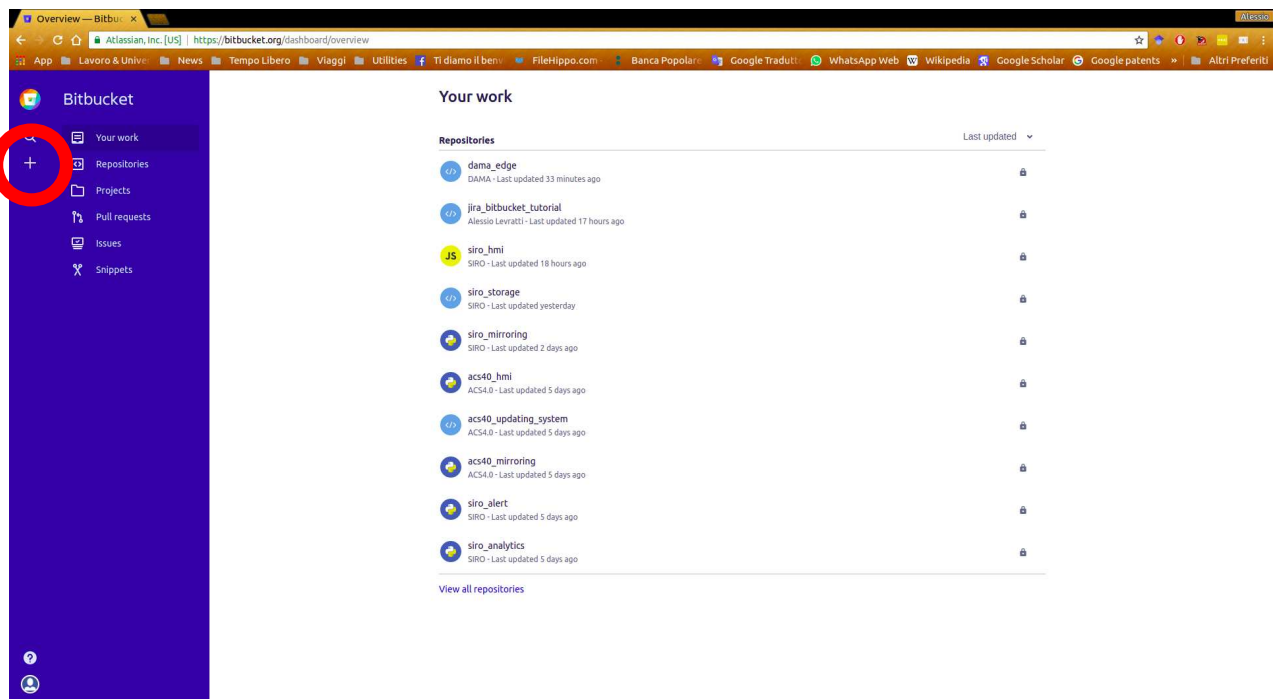


Figura 2: La pagina principale di Bitbucket

Create a new repository

Import repository

Owner

Chaos84

Repository name*

Access level

☒ This is a private repository

Include a README?

No

Version control system

☒ Git

☐ Mercurial

Advanced settings

Description

Forking

Allow only private forks

Project management

☐ Issue tracking

☒ Wiki

Language

Select language...

Integrations

☐ Enable Hipchat notifications

Create repository

Cancel

Figura 3: Finestra di creazione nuovo repository

5.2.3 Issue tracking

Il sistema di “*Issue tracking*” è un sistema che serve per tracciare i bug e le feature da implementare in un software. Bitbucket implementano un pratico ed efficace sistema di issue tracking.

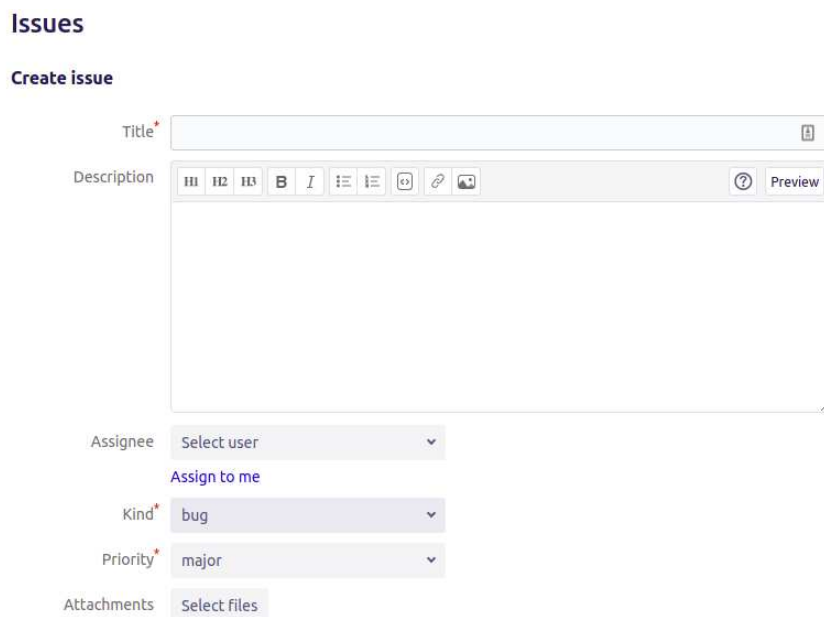
Un issue può essere le seguenti cose:

- Una nuova funzionalità da implementare in un software;
- Un bug da risolvere
- Una richiesta di modifica da parte di un utente interno o esterno all’organizzazione che sta sviluppando il software
- Una richiesta di modifica da parte di altri sviluppatori

Il sistema di issue tracking è estremamente utile per tenere traccia del lavoro svolto e da svolgere. Prima di iniziare a scrivere il codice è utile iniziare il proprio lavoro scomponendo il problema da affrontare in una serie di sotto-problemi e task e da questi identificare gli issue sui quali lavorare.

Bitbucket fornisce un semplice ed intuitivo sistema di issue tracking (vedi Fig. 4) che permette di suddividere gli issue in base alla loro criticità:

- bug: classico “baco” nel software o issue di facile risoluzione;
- enhancement: miglioria da apportare nel software per migliorarne l’efficienza o l’usabilità;
- proposal: proposta di aggiunta di una funzionalità;
- task: richiesta di una feature maggiore da implementare.



The screenshot shows the 'Create issue' interface in Bitbucket. At the top, there's a 'Title' field with a red asterisk indicating it's required. Below it is a 'Description' field with a rich text editor toolbar containing icons for bold, italic, list, link, and image. To the right of the description field is a 'Preview' button. Below the description field are three dropdown menus: 'Assignee' with a 'Select user' option, 'Kind' set to 'bug', and 'Priority' set to 'major'. At the bottom, there's an 'Attachments' section with a 'Select files' button.

Figura 4: L’issue tracking di Bitbucket

Dopo aver creato la propria lista degli issue, è di fondamentale importanza tenerne conto durante lo sviluppo del codice aggiornando la lista mano a mano che nuovi issue vengono identificati e che altri vengono risolti. Per farlo, è sufficiente segnare l’issue come “done” e riportarne l’etichetta nei commit che si fanno nel repository, ad esempio eseguendo il commit con:

```
git commit -a -m ‘Fixed issue#11: FTP communication problem’
```

ed eseguendo il push delle proprie modifiche sul repository remoto.

Consiglio importante: al fine di rendere la “history” del repository leggibile è necessario limitare il numero di push sul repository remoto. Se localmente abbiamo eseguito numerosi commit, è possibile eseguirne lo “squash” per “schiacciarli” insieme in un unico commit, come ad esempio:

```
git reset --soft HEAD~3
```

```
git commit -m "New message for the combined commit"
```

In questo caso l’utente ha “schiacciato” in un unico commit gli ultimi tre commit.

5.2.4 Lavorare in team su di un repository Bitbucket

Una volta creato il repository, per far sì che il proprio team di lavoro, possa contribuire alla realizzazione del software, occorre invitare i diversi sviluppatori a lavorarci. I modi per invitare più collaboratori sono principalmente due:

- Invitare manualmente gli sviluppatori uno ad uno: andare nelle impostazioni di accesso utente del repository (Nel menù a sinistra del repository Settings→User and group access, vedi Fig: 5) e settare gli accessi in lettura e scrittura.

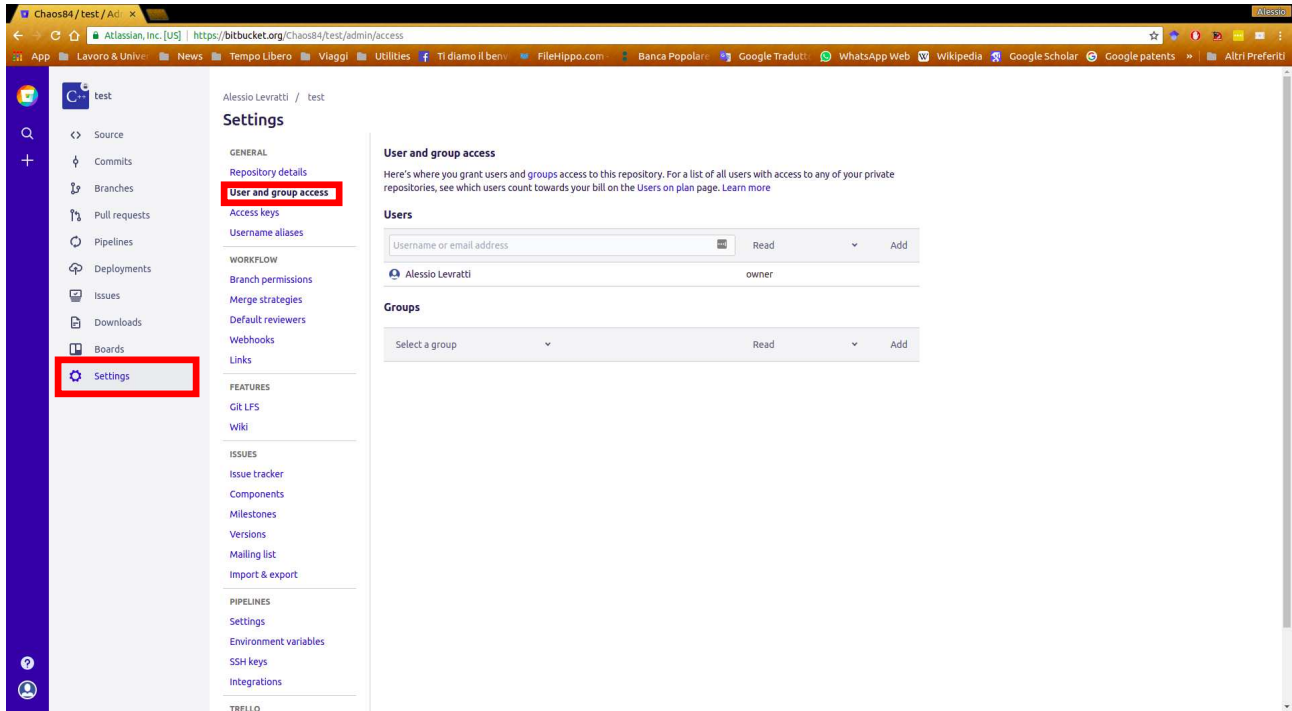


Figura 5: Accesso utenti al repository

- Creare un team di sviluppo, creare un progetto e collegarlo al team, e collocare il repository all'interno del progetto. In questo modo, tutti gli utenti presenti nel team possono interagire con il repository.

Una volta creato il team di sviluppo è possibile assegnare gli issue ai vari collaboratori.

6 Concludendo, dulcis in fundo...

Caro lettore, in conclusione a questo tutorial, spero che tu abbia trovato qualcosa di utile nelle sue pagine. So bene che mancano tantissime cose:

- Una vera e propria guida riassuntiva sulla programmazione in C++;
- Un'introduzione a ROS semplice ed intuitiva che sia diversa dal classico tutorial presente sul sito ufficiale;
- Una sezione nel quale viene introdotto il lavoro di squadra nell'ambito della programmazione in ROS;
- Una vera e propria introduzione alla modellazione del Software tramite l'UML (coming soon...);
- Una spiegazione dettagliata su come, dove e quando creare dei messaggi e dei servizi ROS ad-hoc;
- Ecc. . .

Questa è solo una prima versione di questo tutorial e spero di riuscire a trovare il tempo, fra tutti i miei impegni, di portare avanti questo progetto al fine di agevolare al meglio la fase di “learning” che ogni tesista deve affrontare quando entra a far parte del gruppo ARSControl e per omogeneizzare l'approccio alla programmazione ROS di questo gruppo di lavoro.

Detto questo, spero ti sia divertito leggendo questo tutorial e che tu abbia imparato qualche nuova nozione che tu possa utilizzare durante il tuo lavoro in questo o altri gruppi.

Alessio Levratti

A Appendice A: trasformazioni nello spazio

Come già dovrete sapere, ROS utilizza i “*frame*” per rappresentare la posa di un oggetto nello spazio.

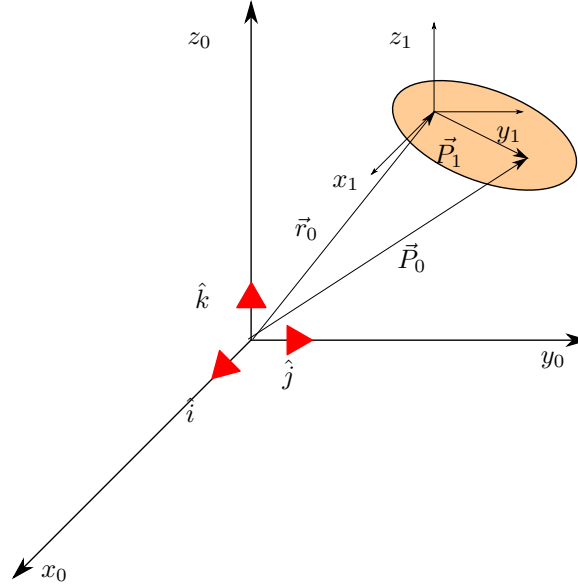


Figura 6: Sistema di coordinate tridimensionale con un corpo.

Tutti quelli che hanno seguito il corso rivolto ai meccatronici di Cocconcelli, sanno che esistono diversi modi per parametrizzare la posizione e l'orientamento di un corpo. Facendo riferimento alla Figura 6, la terna di coordinate $\{x, y, z\}_0$ rappresenta il sistema di riferimento fisso, mentre la terna di coordinate $\{x, y, z\}_1$ rappresenta il sistema di riferimento inerziale solidale al corpo P . I vettori \vec{P}_0 e \vec{P}_1 rappresentano entrambi la posizione di un punto del corpo P in due sistemi di riferimento differenti. Dato il vettore \vec{r}_0 che descrive la posizione del sistema di riferimento 1 rispetto al sistema inerziale 0, la relazione che lega i due vettori è data in 1:

$$\vec{P}_0 = \mathbf{R}_1^0 \cdot \vec{P}_1 + \vec{r}_0 \quad (1)$$

dove \mathbf{R}_1^0 è una **matrice di rotazione**.

In letteratura esistono svariati metodi per parametrizzare l'orientamento di un corpo. In questo tutorial ripercorreremo i vari step per passare dalle matrici di rotazione alla notazione asse/angolo, per giungere infine al metodo del quaternion unitario, che ROS utilizza per rappresentare l'orientamento di un corpo nello spazio⁸.

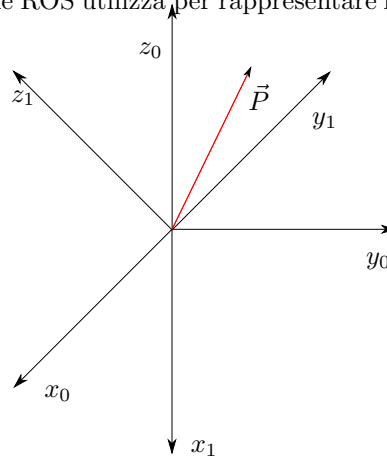


Figura 7: Rappresentazione del vettore \vec{P} in due sistemi di riferimento ruotati uno rispetto l'altro

⁸Nelle pagine seguenti eviterò di dilungarmi troppo sulla teoria e passerò velocemente da una parte all'altra senza fornire troppi dettagli. Perdonatemi, ma non ne ho voglia... Se volete ulteriori dettagli cercateli su internet

Facendo riferimento alla Figura 7, il vettore \vec{P} può essere rappresentato in entrambi i sistemi di riferimento con le equazioni 2, dove $\hat{i}_0, \hat{j}_0, \hat{k}_0$ e $\hat{i}_1, \hat{j}_1, \hat{k}_1$ sono i versori per i sistemi di riferimento $\{x, y, z\}_0$ e $\{x, y, z\}_1$, rispettivamente:

$$\begin{aligned}\vec{P} &= x_0 \cdot \hat{i}_0 + y_0 \cdot \hat{j}_0 + z_0 \cdot \hat{k}_0 \\ \vec{P} &= x_1 \cdot \hat{i}_1 + y_1 \cdot \hat{j}_1 + z_1 \cdot \hat{k}_1\end{aligned}\quad (2)$$

Ponendo uguali le due equazioni in 2 si ottiene:

$$\begin{cases} x_0 = x_1 \cdot \hat{i}_1 \cdot \hat{i}_0 + y_1 \cdot \hat{j}_1 \cdot \hat{i}_0 + z_1 \cdot \hat{k}_1 \cdot \hat{i}_0 \\ y_0 = x_1 \cdot \hat{i}_1 \cdot \hat{j}_0 + y_1 \cdot \hat{j}_1 \cdot \hat{j}_0 + z_1 \cdot \hat{k}_1 \cdot \hat{j}_0 \\ z_0 = x_1 \cdot \hat{i}_1 \cdot \hat{k}_0 + y_1 \cdot \hat{j}_1 \cdot \hat{k}_0 + z_1 \cdot \hat{k}_1 \cdot \hat{k}_0 \end{cases}\quad (3)$$

Ponendo il sistema in 2 in forma matriciale otteniamo:

$$\begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} = \begin{bmatrix} \hat{i}_1 \cdot \hat{i}_0 & \hat{j}_1 \cdot \hat{i}_0 & \hat{k}_1 \cdot \hat{i}_0 \\ \hat{i}_1 \cdot \hat{j}_0 & \hat{j}_1 \cdot \hat{j}_0 & \hat{k}_1 \cdot \hat{j}_0 \\ \hat{i}_1 \cdot \hat{k}_0 & \hat{j}_1 \cdot \hat{k}_0 & \hat{k}_1 \cdot \hat{k}_0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix}\quad (4)$$

che non è altro che $\vec{P}_0 = \mathbf{R}_1^0 \cdot \vec{P}_1$

La matrice di rotazione \mathbf{R}_1^0 gode delle seguenti proprietà:

$\forall \mathbf{R}$ agenti in \mathbb{R}^n :

- $\mathbf{R}^T = \mathbf{R}^{-1}$
- $\det \mathbf{R} = \pm 1$
- $\mathbf{R}^T \cdot \mathbf{R} = \mathbf{I}$

In un sistema di riferimento tridimensionale, è possibile definire le matrici di rotazione elementari per le rotazioni attorno ai tre assi di riferimento:

$$\mathbf{R}_x(\varphi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \varphi & -\sin \varphi \\ 0 & \sin \varphi & \cos \varphi \end{bmatrix} \quad \mathbf{R}_y(\vartheta) = \begin{bmatrix} \cos \vartheta & 0 & \sin \vartheta \\ 0 & 1 & 0 \\ -\sin \vartheta & 0 & \cos \vartheta \end{bmatrix} \quad \mathbf{R}_z(\psi) = \begin{bmatrix} \cos \psi & -\sin \psi & 0 \\ \sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{bmatrix}\quad (5)$$

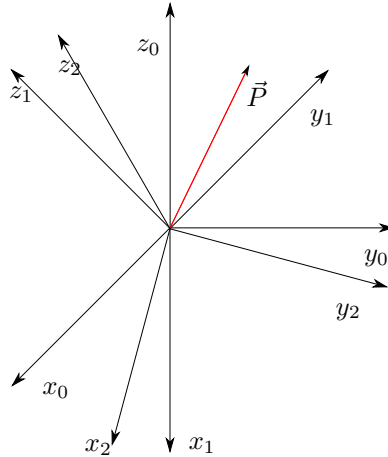


Figura 8: Concatenazione di più rotazioni.

È possibile concatenare matrici di rotazione (vedi figura 8) per ottenere una matrice di rotazione generale che rappresenta la rotazione di un qualsiasi vettore nello spazio.

$$\begin{aligned}\vec{P}_0 &= \mathbf{R}_1^0 \cdot \vec{P}_1 \\ \vec{P}_0 &= \mathbf{R}_1^0 \cdot \mathbf{R}_2^1 \cdot \vec{P}_2 = \mathbf{R}_2^0 \cdot \vec{P}_2\end{aligned}\quad (6)$$

È però di fondamentale importanza notare che se la rotazione è in terna fissa, le rotazioni successive vanno moltiplicate a sinistra, mentre se ci si trova in una terna corrente, è necessario concatenare la matrici a destra (vedi Figura 9)

■

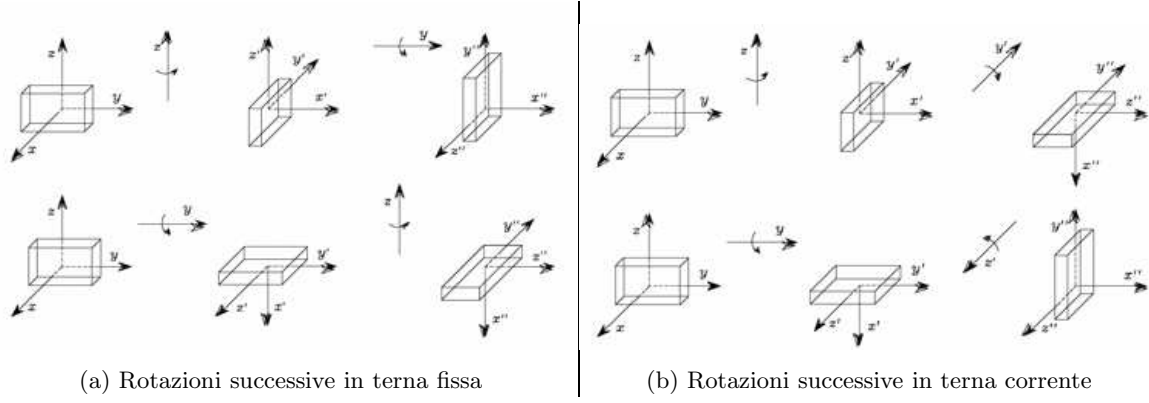


Figura 9: Rotazioni successive

A.0.1 Esempio: rotazione $z - y - z$ in terna corrente

$$\mathbf{R}_{TOT} = \{r_{i,j}\} = \mathbf{R}_z(\psi) \cdot \mathbf{R}_y(\vartheta) \cdot \mathbf{R}_z(\varphi) = \{r_{i,j}\}$$

$$\mathbf{R}_{TOT} = \begin{bmatrix} -\cos \psi \cos \vartheta \cos \varphi - \sin \psi \sin \vartheta & -\cos \psi \cos \vartheta \sin \varphi - \sin \psi \cos \varphi & \cos \psi \sin \vartheta \\ \sin \psi \cos \vartheta \cos \varphi + \cos \psi \sin \vartheta & -\sin \psi \cos \vartheta \sin \varphi + \cos \psi \cos \varphi & \sin \psi \sin \vartheta \\ -\sin \vartheta \cos \varphi & -\sin \vartheta \sin \varphi & \cos \vartheta \end{bmatrix}$$

Note:

- $(\psi, \vartheta, \varphi) \rightarrow \mathbf{R}_{TOT}$ è facile da ottenere la matrice di rotazione complessiva a partire dalle rotazioni elementari;
- Il procedimento inverso, ovvero $\mathbf{R}_{TOT} \rightarrow (\psi, \vartheta, \varphi)$, non è banale!!!!

Infatti: $\vartheta_{1,2} = \pm \arccos(r_{3,3})$.

Considerando $\sin \vartheta > 0$.

$\psi = \arctan 2(r_{1,3}, r_{2,3})$, $\varphi = \arctan 2(-r_{3,1}, r_{3,2})$

Se $\vartheta = 0$ è impossibile distinguere la prima e la terza rotazione!

■

A.0.2 Esempio: rotazione $x - y - z$ in terna fissa

$$\mathbf{R}_{TOT} = \{r_{i,j}\} = \mathbf{R}_z(\phi) \cdot \mathbf{R}_y(\chi) \cdot \mathbf{R}_x(\vartheta) = \{r_{i,j}\}$$

$$\mathbf{R}_{TOT} = \begin{bmatrix} \cos \phi \cos \chi & -\sin \phi \cos \vartheta + \cos \phi \sin \chi \sin \vartheta & \sin \phi \sin \vartheta + \cos \phi \sin \chi \cos \vartheta \\ \sin \phi \cos \chi & \cos \phi \cos \vartheta + \sin \phi \sin \chi \sin \vartheta & -\sin \vartheta \cos \phi + \sin \phi \sin \chi \cos \vartheta \\ -\sin \chi & \sin \vartheta \cos \chi & \cos \vartheta \cos \chi \end{bmatrix}$$

$$\chi_1 = \arcsin(-r_{3,1}), \chi_2 = \pi - \arcsin(r_{3,1})$$

Considerando $\cos \chi \neq 0$.

$$\vartheta = \arctan 2(r_{3,3}, r_{3,2}), \phi = \arctan 2(r_{1,1}, r_{2,1})$$

Se $\cos \chi = 0 \rightarrow \chi = \frac{\pi}{2}$, significa trasformare l'asse x nell'asse z . Questa corrisponde ad una doppia rotazione attorno all'asse z .

■

Ora che abbiamo introdotto le matrici di rotazione, introduciamo la parametrizzazione dell'orientamento tramite la notazione **asse/angolo**.

Secondo la parametrizzazione asse/angolo è sempre possibile rappresentare una sequenza di rotazioni di un oggetto con una singola rotazione attorno ad un asse. Questa notazione rappresenta una rotazione arbitraria mediante quattro parametri $(u_x, u_y, u_z, \vartheta)$ tali che $u_x^2 + u_y^2 + u_z^2 = 1$, vedi Figura 10.

$$\sin \alpha = \frac{u_y}{\sqrt{u_x^2 + u_z^2}}, \cos \alpha = \frac{u_x}{\sqrt{u_x^2 + u_z^2}}, \cos \beta = u_z, \sin \beta = \sqrt{u_x^2 + u_y^2}$$

La matrice per una rotazione di un angolo ϑ attorno ad un asse generico, individuato tramite le rotazioni α e β rispettivamente attorno agli assi z e y , diventa quindi:

$$\mathbf{R}_{TOT} = \mathbf{R}_z(\alpha) \cdot \mathbf{R}_y(\beta) \cdot \mathbf{R}_z(\vartheta) \cdot \mathbf{R}_y(-\beta) \cdot \mathbf{R}_z(-\alpha) = \{r_{i,j}\} =$$

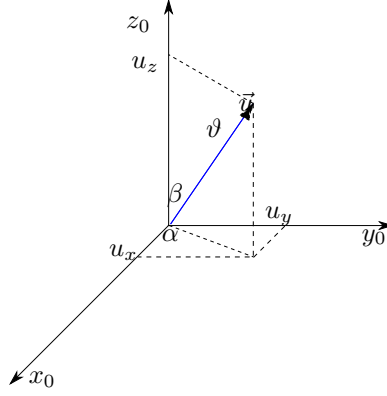


Figura 10: Parametrizzazione asse/angolo

$$\begin{bmatrix} u_x^2(1 - \cos \vartheta) + \cos \vartheta & u_x u_y(1 - \cos \vartheta) - u_z \sin \vartheta & u_x u_z(1 - \cos \vartheta) + u_y \sin \vartheta \\ u_x u_y(1 - \cos \vartheta) - u_z \sin \vartheta & u_y^2(1 - \cos \vartheta) + \cos \vartheta & u_y u_z(1 - \cos \vartheta) - u_x \sin \vartheta \\ u_x u_z(1 - \cos \vartheta) + u_y \sin \vartheta & u_y u_z(1 - \cos \vartheta) - u_x \sin \vartheta & u_z^2(1 - \cos \vartheta) + \cos \vartheta \end{bmatrix}$$

Il procedimento inverso ci permette di trovare i componenti dell'asse attorno al quale avviene la rotazione e l'angolo del quale l'oggetto è ruotato. Partendo dal calcolo della traccia della matrice è possibile calcolare l'angolo di rotazione⁹:

$$\begin{aligned} \text{Tr } \mathbf{R}_{TOT} &= 1 + 2 \cos \vartheta \longrightarrow \cos \vartheta = \frac{r_{1,1} + r_{2,2} + r_{3,3} - 1}{2} \\ \vartheta &= \pm \arccos \left(\frac{r_{1,1} + r_{2,2} + r_{3,3} - 1}{2} \right) \end{aligned}$$

I componenti dell'asse di rotazione possono essere quindi trovati con:

$$u_x = \frac{r_{3,2} - r_{2,3}}{2 \sin \vartheta}, u_z = \frac{r_{2,1} - r_{1,2}}{2 \sin \vartheta} \iff \sin \vartheta \neq 0, \text{ altrimenti}$$

Il terzo componente può essere trovato sfruttando il fatto che $u_x^2 + u_y^2 + u_z^2 = 1$. Da notare che nel caso in cui $\sin \vartheta = 0$, la matrice di rotazione risulterebbe una matrice identità che equivale ad una rotazione nulla.

■

Giunti a questo risultato, è il momento di definire il QUATERNIONE UNITARIO: i quaternioni sono un'estensione del campo dei numeri complessi \mathbb{C} allo spazio tridimensionale.

Partendo dalla notazione asse/angolo, definiamo i seguenti $\eta = \cos \frac{\vartheta}{2}$, $\vec{\varepsilon} = (\varepsilon_x, \varepsilon_y, \varepsilon_z)$. Ma $\vec{\varepsilon} = \vec{u} \sin \frac{\vartheta}{2} = (u_x \sin \frac{\vartheta}{2}, u_y \sin \frac{\vartheta}{2}, u_z \sin \frac{\vartheta}{2})$, con il vincolo $\eta^2 + \varepsilon_x^2 + \varepsilon_y^2 + \varepsilon_z^2 = 1$ (che spiega il fatto che il quaternione sia unitario...).

La matrice di rotazione risultante sarà quindi

$$\mathbf{R} = \begin{bmatrix} 2(\eta^2 + \varepsilon_x^2) - 1 & 2(\varepsilon_x \varepsilon_y - \eta \varepsilon_z) & 2(\varepsilon_x \varepsilon_z + \eta \varepsilon_y) \\ 2(\varepsilon_x \varepsilon_y + \eta \varepsilon_z) & 2(\eta^2 + \varepsilon_y^2) - 1 & 2(\varepsilon_y \varepsilon_z - \eta \varepsilon_x) \\ 2(\varepsilon_x \varepsilon_z - \eta \varepsilon_y) & 2(\varepsilon_y \varepsilon_z + \eta \varepsilon_x) & 2(\eta^2 + \varepsilon_z^2) - 1 \end{bmatrix}$$

Il procedimento inverso, per trovare i componenti del quaternione a partire dalla matrice di rotazione, è il seguente:

Definiamo $\eta = \pm \frac{\sqrt{\text{Tr } \mathbf{R}}}{2}$. Per limitare le soluzioni possibili, definiamo $\frac{\vartheta}{2} \in [-\frac{\pi}{2}, \frac{\pi}{2}]$, ovvero $\vartheta \in [-\pi, \pi]$. Nonostante ci sia una singolarità per $\eta = 0$, è comunque possibile determinare $\vec{\varepsilon}$:

$$\varepsilon_x = \pm \sqrt{\frac{r_{1,1} + 1}{2}} \quad \varepsilon_y = \pm \sqrt{\frac{r_{2,2} + 1}{2}} \quad \varepsilon_z = \pm \sqrt{\frac{r_{3,3} + 1}{2}}$$

■

⁹Ricordo ai lettori che l'operatore "Tr" rappresenta la traccia della matrice, ovvero la somma degli elementi posti sulla diagonale principale della matrice stessa.

B Appendice B: Glossario dei termini

Agile Lo sviluppo software Agile fa riferimento a un gruppo di metodologie basate sullo sviluppo iterativo, in cui i requisiti e le soluzioni si evolvono attraverso la collaborazione tra team interfunzionali. I metodi agili o i processi Agile generalmente promuovono un flusso disciplinato di gestione del progetto che incoraggia frequenti ispezioni e adattamenti del prodotto. Si basa su una filosofia di leadership che incoraggia il lavoro di squadra, l'auto-organizzazione e la responsabilità del team e un approccio di business che allinea lo sviluppo con le esigenze dei clienti e gli obiettivi aziendali. La metodologia Agile si può riferire a qualsiasi processo di sviluppo che sia allineato con i concetti del Manifesto Agile. Scrum quindi è un sottoinsieme di Agile. È un framework semplice e leggero per la gestione di progetti complessi. Per leggero si intende che il sovraccarico di lavoro viene mantenuto al minimo possibile per ogni fase di sviluppo, per massimizzare la quantità di tempo produttivo e la qualità del lavoro stesso. Scrum aumenta significativamente la produttività all'interno dell'azienda e riduce i tempi di lavoro per ottenere dei benefici rispetto ai classici processi "a cascata". I processi Scrum consentono alle organizzazioni di adattarsi facilmente al mercato in rapida evoluzione e di produrre un prodotto che soddisfi gli obiettivi e i requisiti sia del cliente che del business.

Boost Le Librerie C++ Boost sono una collezione di librerie open source che estendono le funzionalità del C++. Molte di esse sono licenziate sotto la Boost Software License in modo da poter essere utilizzate sia in progetti open source che closed source. Alcuni dei fondatori di Boost fanno parte del comitato standard C++ (ISO/IEC 14882) e diverse librerie Boost sono state accettate per l'incorporazione sia in C++ Technical Report 1, sia in C++0x. Per assicurare efficienza e flessibilità, Boost fa un estensivo utilizzo della programmazione basata su template, e quindi sulla programmazione generica e metaprogrammazione..

compilatore Un compilatore è un programma informatico che traduce una serie di istruzioni scritte in un determinato linguaggio di programmazione (codice sorgente) in istruzioni di un altro linguaggio (codice oggetto). Il processo di traduzione si chiama compilazione mentre l'attività inversa - ovvero passare dal codice oggetto al codice sorgente - è chiamata decompilazione ed è effettuata per mezzo di un decompilatore. Se tutti i compilatori aderissero esattamente alla specifica del linguaggio, lo stesso programma potrebbe essere compilato senza modifiche da ciascun compilatore, producendo risultati semanticamente uguali, ovvero programmi che producono lo stesso risultato se sottoposti agli stessi dati di ingresso. Nella realtà, molti compilatori implementano il linguaggio in modo incompleto o aggiungono estensioni proprietarie, creando quindi dei dialetti del linguaggio principale..

Doxygen Doxygen è una applicazione per la generazione automatica della documentazione a partire dal codice sorgente di un generico software. È un progetto open source disponibile sotto licenza GPL, scritto per la maggior parte da Dimitri van Heesch a partire dal 1997. Doxygen è un sistema multiplatforma (Windows, Mac OS, Linux, ecc.) ed opera con i linguaggi C++, C, Java, Objective C, Python, IDL (versioni CORBA e Microsoft), Fortran, PHP, C#, e D. Nell'ambito del C++, è compatibile con le estensioni Qt. È il sistema di documentazione di gran lunga più utilizzato nei grandi progetti open source in C++. Due esempi per tutti, sono l'adozione di doxygen da parte di ACE e KDE. In Java invece, la posizione leader viene meno, in virtù della presenza del concorrente Javadoc. Il sistema estrae la documentazione dai commenti inseriti nel codice sorgente e dalla dichiarazione delle strutture dati.

ereditarietà In informatica l'ereditarietà è uno dei concetti fondamentali nel paradigma di programmazione a oggetti. Essa consiste in una relazione che il linguaggio di programmazione, o il programmatore stesso, stabilisce tra due classi. Se la classe B eredita dalla classe A, si dice che B è una sottoclasse di A e che A è una superclasse di B. Denominazioni alternative equivalenti, sono classe madre o classe base per A e classe figlia o classe derivata per B. A seconda del linguaggio di programmazione, l'ereditarietà può essere ereditarietà singola o semplice (ogni classe può avere al più una superclasse diretta) o multipla (ogni classe può avere più superclassi dirette)..

Git Git è un software di controllo versione distribuito utilizzabile da interfaccia a riga di comando, creato da Linus Torvalds nel 2005. La sua progettazione si ispirò a strumenti (allora proprietari) analoghi come BitKeeper e Monotone. Git (che nello slang americano significa idiota) nacque per essere un semplice strumento per facilitare lo sviluppo del kernel Linux ed è diventato uno degli strumenti di controllo versione più diffusi.

GNOME GNOME (GNU Network Object Model Environment) è un ambiente desktop creato inizialmente dai programmatori messicani Miguel de Icaza e Federico Mena nell'agosto del 1997, con l'obiettivo di fornire sia un ambiente desktop che un ambiente di sviluppo libero per i sistemi operativi basati su GNU/Linux. Grazie a questo e ai risultati raggiunti, è presto stato riconosciuto come il desktop environment ufficiale del progetto GNU.

GNU GNU (acronimo ricorsivo di "GNU's Not Unix") è un sistema operativo Unix-like, ideato nel 1984 da Richard Stallman e promosso dalla Free Software Foundation, allo scopo di ottenere un sistema operativo completo utilizzando esclusivamente software libero. Dato che GNU Hurd, il kernel ufficiale del progetto, non è considerato pronto per la distribuzione, GNU viene in genere utilizzato congiuntamente ad altri kernel tra cui Linux, Linux-libre, XNU o quello utilizzato da FreeBSD. La parola GNU si pronuncia "gh-nù" e non "gniù" per non confonderlo con l'omonima specie animale o con l'aggettivo inglese new.

incapsulamento In informatica, nell'ambito della programmazione, si definisce incapsulamento (o encapsulation) la tecnica di nascondere il funzionamento interno, deciso in fase di progetto, di una parte di un programma, in modo da proteggere le altre parti del programma dai cambiamenti che si produrrebbero in esse nel caso che questo funzionamento fosse difettoso, oppure si decidesse di implementarlo in modo diverso. Per avere una protezione completa è necessario disporre di una robusta interfaccia che protegga il resto del programma dalla modifica delle funzionalità soggette a più frequenti cambiamenti.

ISA le relazioni ISA (relazione "è un") specificano che una classe deve considerarsi una sottocategoria o sottoclasse di un'altra (detta superclasse); per esempio, la classe conto corrente potrebbe essere descritta come sottocategoria della classe servizio bancario. Il nome della relazione segue dal fatto che si può sensatamente dire, per esempio, che un conto corrente è un ("is a", ISA) servizio bancario. Si noti che questa rappresenta una relazione esclusivamente fra classi e non fra istanze. La relazione ISA ha un ruolo importante nella razionalizzazione di un modello. Un aspetto fondamentale è che si può assumere che una sottoclasse sia sicuramente dotata di tutti gli attributi, le operazioni e le associazioni della superclasse; eventualmente può averne altre, legate alle particolarità specifiche della sottocategoria di oggetti che descrive. Per esempio, se un servizio bancario ha un costo annuo ed è intestato a un cliente, tutto ciò resta vero per un conto corrente; quest'ultimo ha inoltre un saldo, attributo che non ha necessariamente senso per tutti i servizi bancari (e che quindi non comparirà nella classe più generica servizio bancario).

Issue tracking Sistema che tiene traccia delle segnalazioni di bug all'interno dei software, in modo che questi errori siano mantenuti sotto controllo, con una descrizione della riproducibilità e dei dettagli ad essi correlati, dunque più facilmente risolvibili. Solitamente un issue tracking system è strutturato per rappresentare lo stato dell'issue, quale ad esempio "nuovo" se nessuno ne ha ancora preso in carico la verifica o "risolto" se al contrario è stato rimosso e se ne vuole comunque tenere traccia per consultazioni future qualora l'imperfezione dovesse ripresentarsi, e supporta una gerarchia di utenza per cui solo un amministratore della piattaforma può variare lo stato di ogni elemento ed operare particolari azioni.

kernel Il kernel, in informatica, costituisce il nucleo fondamentale di un sistema operativo ovvero il software avente il compito di fornire ai processi in esecuzione sull'elaboratore un accesso sicuro e controllato all'hardware. Dato che possono esserne eseguiti simultaneamente più di uno, il kernel ha anche la responsabilità di assegnare una porzione di tempo-macchina (scheduling) e di accesso all'hardware a ciascun programma (multitasking). Naturalmente, un kernel non è strettamente necessario per far funzionare un elaboratore: i programmi possono essere infatti direttamente caricati ed eseguiti sulla macchina, a patto che i loro sviluppatori ritengano necessario fare a meno del supporto del sistema operativo: questa era la modalità di funzionamento tipica dei primi elaboratori, che venivano resettati prima di eseguire un nuovo programma. In un secondo tempo, alcuni programmi accessori come i program loader e i debugger venivano lanciati da una ROM o fatti risiedere in memoria durante le transizioni dell'elaboratore da un'applicazione all'altra: essi formarono la base di fatto per la creazione dei primi sistemi operativi.

linker In informatica il linking (letteralmente "collegamento") [1] è il procedimento di integrazione dei vari moduli a cui un programma fa riferimento (i quali possono essere sottoprogrammi o librerie), per creare una singola unità eseguibile. Il linker (o link editor) è un programma che effettua il collegamento tra il programma oggetto, cioè la traduzione del codice sorgente in linguaggio macchina, e le librerie del linguaggio necessarie per l'esecuzione del programma. Nei mainframe IBM come gli OS/360 questo programma è chiamato linkage editor. Nelle varianti di Unix è spesso usato anche il termine loader come sinonimo di linker. I codici oggetto sono parti di programma contenenti sia codice macchina che informazioni per il linker. Questa informazione è costituita principalmente nella forma di definizioni di simboli, che sono di due tipi:

- I simboli definiti o esportati sono simboli che rappresentano quelle funzioni o quelle variabili che sono presenti nel modulo (quindi nel relativo codice oggetto), e che vengono rese disponibili per essere utilizzate da parte di altri moduli;
- I simboli non definiti né importati rappresentano funzioni che vengono chiamate, o variabili che vengono referenziate, nell'oggetto presente, e tuttavia non sono definite internamente, bensì sono definite in altri moduli.

Il lavoro del linker consiste nel risolvere i collegamenti ai simboli non definiti, trovando quale altro modulo li definisce; e quindi nel rimpiazzare ciascun segnaposto con l'effettivo indirizzo del simbolo. Un insieme di moduli software pronti per essere utilizzati da altri programmi viene chiamato "libreria". Una libreria è quindi un insieme di moduli (in forma di codici-oggetto) raccolti in un unico file contenitore. I linker possono utilizzare i moduli software contenuti nelle librerie. Alcuni linker non includono l'intera libreria nel programma di output, ma solo le parti che si rendono necessarie perché referenziate da altro codice oggetto o altre librerie. Esistono librerie per svariati scopi (esempi: interazione con l'hardware, calcolo matematico); tipicamente, le librerie più comuni sono utilizzate automaticamente in modo implicito, mentre quelle più specializzate devono essere indicate esplicitamente dal programmatore. Il linker si occupa anche di distribuire il software (codice oggetto) nello spazio di indirizzi del programma. Questo può richiedere lo spostamento di codice da un certo indirizzo base ad un altro; questa è una operazione importante perché spostare un segmento di codice da un indirizzo ad un altro richiede di ricalcolare e modificare tutti gli indirizzi che puntano a quel codice. Poiché normalmente il compilatore non sa dove (in quale indirizzo di memoria) risiederà il codice relativo a un particolare modulo, il compilatore ipotizza un indirizzo fisso (ad esempio zero), che dovrà poi essere modificato nel momento in cui tutti i moduli vengono combinati in un codice eseguibile unico. Spostare un segmento di codice da un indirizzo ad un altro viene chiamato "rilocazione" e può richiedere il ricalcolo degli indirizzi utilizzati nelle istruzioni di salto assoluto o nelle istruzioni di caricamento o salvataggio di dato in memoria. Il programma eseguibile prodotto dal linker potrebbe richiedere un'ulteriore rilocazione quando il programma viene caricato in memoria (appena prima della sua esecuzione). Questa operazione non è necessaria nei computer che dispongono di un sistema di memoria virtuale; in questi computer ogni programma ha a disposizione un proprio spazio di indirizzamento privato. Nei sistemi a memoria virtuale tutti i programmi utilizzano lo stesso indirizzo base senza che ci siano conflitti tra programmi diversi. I sistemi operativi moderni prevedono il dynamic linking, ossia la risoluzione di simboli non definiti rimandata fino a che il programma non è in esecuzione. Questo significa che l'eseguibile contiene simboli non definiti e una lista degli oggetti o librerie che ne possono fornire la definizione. Lanciando il programma per l'esecuzione vengono caricati questi oggetti / librerie e si stabilisce il collegamento finale. Questa modalità di esecuzione offre due vantaggi:

- Le librerie molto usate (come ad esempio quelle standard di sistema) possono essere memorizzate in una sola locazione, e non duplicate ad ogni richiesta;
- Se viene aggiornata e rimpiazzata una libreria di funzioni, tutti i programmi che la usano dinamicamente beneficiano dell'aggiornamento appena vengono rieseguiti. Al contrario, i programmi che includono tali funzioni con collegamento statico devono essere sottoposti ad un nuovo passaggio di linkage editor / loader

Linux Linux, è una famiglia di sistemi operativi di tipo Unix-like, pubblicati sotto varie possibili distribuzioni, aventi la caratteristica comune di utilizzare come nucleo il kernel Linux. Oggi molte società importanti nel campo dell'informatica come Google, IBM, Oracle Corporation, Hewlett-Packard, Red Hat, Canonical, Novell e Valve sviluppano e pubblicano sistemi Linux.

Object Oriented Programming In informatica la programmazione orientata agli oggetti (OOP, Object Oriented Programming) è un paradigma di programmazione che permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi. È particolarmente adatta nei contesti in cui si possono definire delle relazioni di interdipendenza tra i concetti da modellare (contenimento, uso, specializzazione). Un ambito che più di altri riesce a sfruttare i vantaggi della programmazione ad oggetti è quello delle interfacce grafiche. Tra gli altri vantaggi della programmazione orientata agli oggetti:

- fornisce un supporto naturale alla modellazione software degli oggetti del mondo reale o del modello astratto da riprodurre
- permette una più facile gestione e manutenzione di progetti di grandi dimensioni
- permette una più facile gestione e manutenzione di progetti di grandi dimensioni

- l'organizzazione del codice sotto forma di classi favorisce la modularità e il riuso di codice

open-source In informatica il termine inglese open source (che significa sorgente aperta) viene utilizzato per riferirsi ad un software di cui gli autori (più precisamente, i detentori dei diritti) rendono pubblico il codice sorgente, favorendone il libero studio e permettendo a programmatori indipendenti di apportarvi modifiche ed estensioni. Questa possibilità è regolata tramite l'applicazione di apposite licenze d'uso. Il fenomeno ha tratto grande beneficio da Internet, perché esso permette a programmatori distanti di coordinarsi e lavorare allo stesso progetto. Alla filosofia del movimento open source si ispira il movimento open content (contenuti aperti): in questo caso, ad essere liberamente disponibile non è il codice sorgente di un software, ma contenuti editoriali quali testi, immagini, video e musica. Wikipedia è un chiaro esempio dei frutti di questo movimento. Attualmente, l'open source tende ad assumere rilievo filosofico, consistendo in una nuova concezione della vita, aperta e refrattaria ad ogni oscurantismo, che l'open source si propone di superare mediante la condivisione della conoscenza. Open source e software libero, seppure siano sovente utilizzati come sinonimi, hanno definizioni differenti: l'Open Source Initiative ha definito il termine "open source" per descrivere soprattutto libertà sul codice sorgente di un'opera. Il concetto di software libero descrive più generalmente le libertà applicate ad un'opera, ed è prerequisito che il suo codice sia consultabile e modificabile, rientrando generalmente nella definizione di open source.

polimorfismo In informatica, il termine polimorfismo viene usato in senso generico per riferirsi a espressioni che possono rappresentare valori di diversi tipi (dette espressioni polimorfiche). In un linguaggio non tipizzato, tutte le espressioni sono intrinsecamente polimorfiche. Il termine viene associato a due significati specifici:

- nel contesto della programmazione orientata agli oggetti, si riferisce al fatto che una espressione il cui tipo sia descritto da una classe A può assumere valori di un qualunque tipo descritto da una classe B sottoclasse di A (polimorfismo per inclusione);
- nel contesto della programmazione generica, si riferisce al fatto che il codice del programma può ricevere un tipo come parametro invece che conoscerlo a priori (polimorfismo parametrico)

posa Il termine "**posizione**" indica, da un punto di vista matematico, la locazione relativa di un oggetto all'interno di un sistema di riferimento. Il termine "**orientamento**" indica il posizionamento di un oggetto in un sistema di coordinate rotazionali, rispetto ad un punto fisso e ad un punto di riferimento. Il termine "**posa**" indica entrambe la posizione e l'orientamento del corpo in uno spazio tridimensionale.

quaternioni In matematica, i quaternioni sono entità introdotte da William Rowan Hamilton nel 1843 come estensioni dei numeri complessi. Un quaternioni è un oggetto formale del tipo $a + b\vec{i} + c\vec{j} + d\vec{k}$, dove $\{a, b, c, d\} \in \mathbb{R}$ e \vec{i}, \vec{j} e \vec{k} sono dei simboli che si comportano in modo simile all'unità immaginaria dei numeri complessi. I quaternioni formano un corpo: soddisfano quindi tutte le proprietà usuali dei campi, quali i numeri reali o complessi, tranne la proprietà commutativa del prodotto. Le estensioni dei quaternioni, quali gli ottetti e i sedenioni, non hanno neppure la proprietà associativa. I quaternioni contengono i numeri complessi $a + bi$ e formano anche uno spazio vettoriale reale di dimensione 4 (analogamente ai complessi, che sono uno spazio a 2 dimensioni, cioè un piano). Le due proprietà di corpo e di spazio vettoriale conferiscono ai quaternioni una struttura di algebra di divisione non commutativa. I quaternioni trovano un'importante applicazione nella modellizzazione delle rotazioni dello spazio: per questo motivo questi sono ampiamente usati nella fisica teorica (nella teoria della relatività e nella meccanica quantistica) e in settori più applicati, come la computer grafica 3D e la robotica (per individuare la posizione spaziale dei bracci meccanici a più snodi)..

Repository Un repository (letteralmente deposito o ripostiglio) è un ambiente di un sistema informativo (ad es. di tipo ERP), in cui vengono gestiti i metadati, attraverso tabelle relazionali; l'insieme di tabelle, regole e motori di calcolo tramite cui si gestiscono i metadati prende il nome di metabase.

Scrum Scrum è un framework che consente alle persone di risolvere problemi complessi in maniera adattiva e, al tempo stesso, di creare e rilasciare prodotti in modo efficace e creativo dal più alto valore possibile per l'utente finale.

Sprint Lo sprint è una delle strategie di lavoro definite dallo Scrum. Il progetto viene portato avanti e sviluppato tramite degli Sprint di qualche settimana ciascuno, nei quali si deve portare a termine un determinato numero di compiti.

Ubuntu Ubuntu (pronunciato [u:'bu:ntu]) è un sistema operativo nato nel 2004, focalizzato sulla facilità di utilizzo. È prevalentemente composto da software libero proveniente dal ramo unstable di Debian GNU/Linux, ma contiene anche software proprietario, ed è distribuito liberamente con licenza GNU GPL. È orientato all'utilizzo sui computer desktop, ma presenta delle varianti per server, tablet, smartphone e dispositivi IoT, ponendo grande attenzione al supporto hardware. Il nome Ubuntu è un termine in un dialetto nguni-bantu traducibile come "umanità verso gli altri". È un riferimento ad una filosofia di origine sudafricana che teorizza un legame universale di scambio che unisce l'intera umanità (letteralmente, "dell'Essere Umano"). L'obiettivo è portare questa idea nel mondo del software, dando un grande peso alla comunità di utenti partecipanti nello sviluppo del sistema operativo.

UML In ingegneria del software, UML (Unified Modeling Language, "linguaggio di modellizzazione unificato") è un linguaggio di modellazione e specifica basato sul paradigma orientato agli oggetti. Il nucleo del linguaggio fu definito nel 1996 da Grady Booch, Jim Rumbaugh e Ivar Jacobson (detti "i tre amigos") sotto l'egida dell'Object Management Group, consorzio che tuttora gestisce lo standard UML. Il linguaggio nacque con l'intento di unificare approcci precedenti (dovuti ai tre padri di UML e altri), raccogliendo le migliori prassi nel settore e definendo così uno standard industriale unificato. UML svolge un'importantissima funzione di "lingua franca" nella comunità della progettazione e programmazione a oggetti. Gran parte della letteratura di settore usa UML per descrivere soluzioni analitiche e progettuali in modo sintetico e comprensibile a un vasto pubblico. La versione 2.0 è stata consolidata nel 2004 e ufficializzata da OMG nel 2005. UML 2.0 riorganizza molti degli elementi della versione precedente (1.5) in un quadro di riferimento ampliato e introduce molti nuovi strumenti, inclusi alcuni nuovi tipi di diagrammi. Sebbene OMG indichi UML 2.0 come la versione "corrente" del linguaggio, la transizione è di fatto ancora in corso; le stesse specifiche pubblicate da OMG sono ancora non completamente aggiornate e il supporto dei tool a UML 2.0 è, nella maggior parte dei casi, appena abbozzato. L'ultima versione è la 2.5, finalizzata nel 2013.

Unity Utilizza le librerie nux (un toolkit basato su OpenGL) e in Ubuntu è implementata con diverse applicazioni di GNOME, utilizzando anche molte dipendenze di tale ambiente desktop. Concepita inizialmente per l'utilizzo sui netbook (Ubuntu Netbook Remix), si è successivamente notato che funzionalità ritenute inizialmente utili solo per i piccoli schermi fossero adatte anche all'utilizzo in ambito desktop. Dalla versione di Ubuntu 11.04 le due distribuzioni (quella desktop e quella per netbook) sono state così unificate. Utilizza Compiz come gestore di finestre. La maggiore velocità di Compiz è stato uno dei motivi principali che ha spinto Canonical a scegliere Unity rispetto a GNOME 3 (che utilizza il compositing Mutter).

Version Control System Il controllo versione (versioning), in informatica, è la gestione di versioni multiple di un insieme di informazioni. Gli strumenti software per il controllo versione sono ritenuti molto spesso necessari per la maggior parte dei progetti di sviluppo software.

YAML YAML (pronunciato in rima con camel) è un formato per la serializzazione di dati utilizzabile da esseri umani. Il linguaggio sfrutta concetti di altri linguaggi come il C, il Perl e il Python e idee dal formato XML e dal formato per la posta elettronica (RFC2822). Proposto da Clark Evans nel 2001, è stato sviluppato da quest'ultimo e Brian Ingerson. Il nome definisce l'acronimo ricorsivo "YAML Ain't a Markup Language". Nella prima fase di sviluppo l'acronimo veniva definito come "Yet Another Markup Language", significato che è andato perso in favore di un nome che specificasse la natura orientata alla memorizzazione di dati del linguaggio, contrapposto all'utilizzo consoni dei linguaggi di markup..

Riferimenti bibliografici

- [1] ECKEL, B. *Thinking in C++ - Volume 1*, 2nd ed. Prentice Hall, 2000.
- [2] ECKEL, B. *Thinking in C++ - Volume 2*, 2nd ed. Prentice Hall, 2000.
- [3] FERNANDEZ, E., CRESPO, L. S., MAHTANI, A., AND MARTINEZ, A. *Learning ROS for Robotics Programming - Second Edition*, 2nd ed. Packt Publishing, 2015.
- [4] JOSEPH, L. *Mastering ROS for Robotics Programming*. Packt Publishing, December 2015.
- [5] LENTIN, J., AND CACACE, J. *Mastering ROS for Robotics Programming - Second Edition: Design, build, and simulate complex robots using the Robot Operating System*. February 2018.
- [6] LEVRATTI, A. Jira e Bitbucket: Manuale d'uso per ARSControl e IT-I. June 2018.
- [7] NEWMAN, W. *A Systematic Approach to Learning Robot Programming with ROS*. Chapman and Hall/CRC, 2017.
- [8] O'KANE, J. M. *A Gentle Introduction to ROS*. CreateSpace Independent Publishing Platform, November 2013.
- [9] QUIGLEY, M., GERKEY, B., AND SMART, W. D. *Programming Robots with ROS: a practical introduction to the Robot Operating System*. " O'Reilly Media, Inc.", 2015.
- [10] STROUSTRUP, B. *C++. Linguaggio, libreria standard, principi di programmazione*, 3rd ed. Addison-Wesley, 1997.
- [11] STROUSTRUP, B. *Programming Principles and Practice Using C++*, 2nd ed. Addison-Wesley, 2014.
- [12] YAROSHENKO, O. *The Beginner's Guide to C++*. Wrox Press, 1994.
- [13] YOONSEOK PYO, HANCHEOL CHO, L. J. D. L. *ROS Robot Programming (Second Edition)*. Ruby Paper, August 2017.