1. Processi primari

1.1 Fornitura

In questa sezione vengono tratta le norme che i membri del gruppo 353 sono tenuti a rispettare al fine di proporsi e diventare fornitori nei confronti della Proponente Red Babel e dei Committenti Prof. Tullio Vardanega e Prof. Riccardo Cardin nell'ambito della progettazione, sviluppo e consegna del prodotto *Marvin*.

1.1.1 Studio di fattibilità

In seguito alla presentazione ufficiale dei **Capitolati d'appalto** avvenuta Venerdì 10 novembre 2017 alle ore 10.30 presso l'aula 1C150 di Torre Archimede è stata convocata una riunione interna al gruppo per discutere in merito alla varie proposte presentate.
Una volta stabilita la scelta del capitolato per il quale proporsi come fornitori, gli analisti hanno condotto un'ulteriore e approfondita attività di analisi dei rischi e delle opportunità culminata con la redazione del documento *Studio di fattibilità v 1.0.0*. Tale documento include le motivazioni che hanno portato il gruppo 353 a proporsi come fornitore per il prodotto indicato e riporta per ciascun capitolato:

- **Descrizione generale:** una sintesi del prodotto da sviluppare secondo quanto stabilito dal capitolato d'appalto;
- **Obbiettivo finale:** rappresenta il Dominio Applicativo, cioè l'ambito di utilizzo del prodotto da sviluppare;
- **Tecnologie richieste:** rappresenta il Dominio Tecnologico richiesto dal capitolato, raggruppando le tecnologie da impiegare nello sviluppo del progetto;
- **Valutazione finale:** racchiude le motivazioni, i rischi, le criticità evidenziate per le quali il capitolato in questione è stato respinto o accettato.

1.1.2 Rapporti di fornitura con la Proponente Red Babel

Durante l'intero progetto si intende instaurare con la Proponente Red Babel e con i referenti Alessandro Maccagnan e Milo Ertola un profondo e quanto più costante rapporto al fine di:

- determinare bisogni;
- stabilire scelte volte alla realizzazione e realizzazione del prodotto (vincoli sui requisiti);
- stabilire scelte volte alla definizione ed esecuzione di processi (vincoli di progetto);
- stimare i costi;
- accordarsi circa la qualifica di prodotto.

1.1.3 Documentazione fornita

Di seguito vengono elencati i documenti forniti alla Proponente Red Babel e ai Committenti Prof. Tullio Vardanega e Prof. Riccardo Cardin al fine di assicurare la massima trasparenza circa le attività di:

- **pianifica, consegna e completamento:** descritte all'interno del *Piano di progetto v 1.0.0*;
- **analisi:** analisi dei requisiti e dei casi d'uso del gruppo vengono descritte all'interno del documento *Analisi dei requisiti v 1.0.0*;
- **verifica e validazione:** descritte all'interno del *Piano di qualifica v 1.0.0*;
- **garanzia della qualità dei processi e di prodotto:** descritte e definite nel sopra citato *Piano di qualifica v 1.0.0*.

1.2 Sviluppo

1.2.1 Analisi dei Requisiti

L'obbiettivo dell'analisi dei requisiti è quello di individuare ed elencare tutti i requisiti$_G$ del capitolato. I requisiti possono essere estrapolati da più fonti:

- **Capitolato d'appalto**;

- **Verbali di riunioni esterne**;
- **Casi d'uso**.

Il risultato di quest'analisi sarà il documento *Analisi dei requisiti v 1.0.0* redatto dagli Analisti,dopo essere stato verificato, al fine di:

- descrivere l'obiettivo del lavoro del gruppo;
- fornire ai Progettisti riferimenti precisi ed affidabili;
- fissare le funzionalità e i requisiti concordati col cliente;
- fornire una base per raffinamenti successivi al fine di garantire un miglioramento continuo del prodotto e del processo di sviluppo;
- facilitare le revisioni del codice;
- fornire ai Verificatori riferimenti per l'attività di test circa i casi d'uso principali e alternativi;
- stimare i costi.

Ogni requisito dovrà essere meno ambiguo possibile e rispettare le seguenti norme.

Requisiti: Ogni requisito è identificato da un codice, costruito come descritto di seguito:

R[Importanza][Classificazione][Identificativo]

- la prima lettera (R) è l'abbreviazione di requisito;
- il secondo valore indica l'**importanza**. Assume il valore:
  - **Zero (0):** indica un requisito obbligatorio, il cui soddisfacimento dovrà necessariamente avvenire per garantire le funzioni base del sistema;
  - **Uno (1):** se si tratta di un requisito desiderabile, cioè un requisito il cui soddisfacimento può dare maggiore completezza al sistema ma il non soddisfarlo non pregiudica alcuna funzione di base;
  - **Due (2):** indica un requisito opzionale;
- la terza lettera indica la **classificazione**. Assume valore F se si tratta di un Requisito funzionale, Q se di qualità, P se prestazionale e V se di vincolo;
- l'**identificativo** indica invece un numero progressivo.

Casi d'uso: gli Analisti hanno anche il compito di identificare i casi d'uso, elencandoli con un grado di precisione che va dal generico verso il dettaglio. Ogni caso d'uso è descritto dalla seguente struttura:

- **Codice identificativo e nome:** ogni caso d'uso è identificato da una serie di cifre separate dal punto. L'ultima cifra indica il numero di figlio, la penultima cifra indica il numero del padre e UC è l'abbreviazione di Use Case(casi d'uso in inglese). Queste cifre sono seguite dopo il trattino(-) dal nome del caso d'uso:

  UC[Codice padre].[Codice figlio]-Nome

- **Attori:** indica gli attori principali (ad esempio l'utente generico) e secondari (ad esempio ufficio universitario) del caso d'uso;
- **Scopo e descrizione:** riporta una breve descrizione del caso d'uso;
- **Scenario principale:** rappresenta il flusso degli eventi come lista numerata, specificando per ciascun evento: titolo, descrizione, attori coinvolti e casi d'uso generati;
- **Precondizione:** specifica le condizioni che sono identificate come vere prima del verificarsi degli eventi del caso d'uso;
- **Postcondizione:** specifica le condizioni che sono identificate come vere dopo il verificarsi degli eventi del caso d'uso;
- **Inclusioni (se presenti):** usate per non descrivere più volte lo stesso flusso di eventi, inserendo il comportamento comune in un caso d'uso a parte;
- **Estensioni (se presenti):** descrivono i casi d'uso che non fanno parte del flusso principale degli eventi, allo stesso modo di quanto descritto in "Scenario principale".

Tracciamento: il gruppo 353 ha deciso di utilizzare il software **Trender** per il tracciamento dei requisiti.

1.2.2 Progettazione

L'attività di Progettazione consiste nel descrivere una soluzione del problema che sia soddisfacente per tutti gli stakeholders. Essa serve a garantire che il prodotto sviluppato soddisfi le proprietà e i bisogni specificati nell'attività di analisi. La progettazione permette di:

- Garantire la qualità di prodotto sviluppato, perseguendo la *correttezza per costruzione*;
- Organizzare e ripartire compiti implementativi, riducendo la complessità del problema originale fino alle singole componenti facilitandone la codifica da parte dei singoli Programmatori;
- Ottimizzare l'uso di risorse.

È compito dei Progettisti svolgere tale attività, definendo l'architettura logica del prodotto identificando componenti chiare, riusabili e coese rimanendo nei costi fissati. L'architettura definita dovrà:

- soddisfare i requisiti definiti nel documento *Analisi dei requisiti*e adattarsi facilmente nel caso essi evolvano o se ne aggiungano di nuovi;
- essere comprensibile, modulare e robusta riuscendo a gestire situazioni erronee improvvise;
- essere affidabile, cioè svolgere ai suoi compiti quando viene usata;
- essere sicura rispetto ad intrusioni e malfunzionamenti;
- essere disponibile, riducendo i tempi di manutenzione;
- avere componenti semplici, coese nel raggiungere gli obiettivi, incapsulate e con scarse dipendenze tra loro.

Al fine di rendere più chiare le scelte progettuali adottate e ridurre le possibili ambiguità, sarà necessario fare largo uso di vari tipi di diagrammi **UML 2.0** e i **design pattern** (adattati per i linguaggi di programmazione elencati sopra).

1.2.3 Codifica

In questa sotto-sezione vengono elencate le norme alle quali i Programmatori devono attenersi durante l'attività di programmazione e implementazione.
All'inizio verranno elencate delle norme generali a cui i Programmatori devono attenersi con qualsiasi linguaggio di programmazione utilizzato, mentre di seguito verranno elencate delle norme specifiche per i linguaggi Javascript, React, Solidity e Scss.
Ogni norma è rappresentata da un paragrafo. Ciascuna ha un titolo, una breve descrizione e, se necessario, un esempio che illustra i modi accettati o meno. Alcune di esse includono inoltre una lista di possibili eccezioni d'uso. L'uso di norme e convenzioni è fondamentale per permettere la generazione di codice leggibile e uniforme, agevolare le fasi di manutenzione, verifica e validazione e migliorare la qualità del prodotto.

Convenzioni per i nomi:

- nomi da evitare perché facilmente confondibili con i numeri `1` e `0`:
  - `l` (lettera minuscola elle);
  - `O` (lettera maiuscola o);
  - `I` (lettera maiuscola i).
- tutti i nomi devono essere **unici** ed **esplicativi** al fine di evitare al più possibile ambiguità e comprensione.

Per i vari linguaggi verranno successivamente descritte altre norme per nomi di variabili, funzioni e altro facendo riferimento ai seguenti stili:

- `lowercase`;
- `lower_case_with_underscores`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (o `CapWords`)
- `mixedCase`
- `Capitalized_Words_With_Underscores`
- `lower-case-with-dashes`

Convenzioni per la documentazione:

- tutti i nomi e i commenti al codice per la documentazione vanno scritti in **inglese**;
- è possibile utilizzare in un commento la keyword **TODO** per indicare codice temporaneo e soluzioni a breve termine o evidentemente migliorabili;
- ogni file contenente codice deve avere la seguente **intestazione** contenuta in un commento e posta all'inizio del file stesso:

```
File : nome file
Version : versione file
Type : tipo file
Date : data di creazione
Author : nome autore/i
E- mail : email autore/i

License : tipo licenza

Advice : lista avvertenze e limitazioni

Changelog :
Autore || Data || breve descrizione delle modifiche
```

- La **versione** del codice viene inserita all'interno dell'intestazione del file e rispetta il seguente formalismo:

  X.Y

  - **X**: è l'indice di versione principale, un incremento di tale indice rappresenta un avanzamento della versione stabile, che porta il valore dell'indice Y ad essere azzerato;
  - **Y**: è l'indice di modifica parziale, un incremento di tale indice rappresenta una verifica o una modifica rilevante, come per esempio la rimozione o l'aggiunta di una istruzione.

  La versione *1.0* deve rappresentare la prima versione del file completo e stabile, cioè quando le sue funzionalità obbligatorie sono state definite e si considerano funzionanti. Solo dalla versione *1.0* è possibile testare il file, con degli appositi test definiti, per verificarne l'effettivo funzionamento.

1.2.3.1 JavaScript

In questa sotto-sezione vengono elencate le norme tratte dal **Airbnb JavaScript Style Guide**[1].

Indentazione 1: vanno utilizzati due (2) spazi per ogni livello di indentazione.

[1] https://github.com/airbnb/javascript

SI

```
function() {
..let name;
}
```

NO

```
function() {
....let name;
}
```

Indentazione 2: bisogna mettere uno (1) spazio prima della graffa principale.

SI

```
function() {
..let name;
}
```

NO

```
function(){
..let name;
}
```

Indentazione 3: va inserito uno (1) spazio prima della parentesi di apertura degli statement di controllo (`if`, `while` etc.).
Non va inserito alcuno spazio prima della lista degli argomenti nelle chiamate di funzioni e nelle dichiarazioni.

SI

```
function fight() {
  console.log('353');
}
```

NO

```
function fight () {
  console.log ('353');
}
```

Spazi: è vietato l'utilizzo di tabulazioni, che devono essere necessariamente sostituite da spazi. Al fine di assicurare il rispetto di questa regola si consiglia di configurare adeguatamente il proprio editor o IDE.

Linee vuote: bisogna lasciare una riga vuota dopo blocchi e prima di un nuovo statement.

SI

```
if (foo) {
  return bar;
}

return baz;

//oppure

var obj = {
  foo: function() {
  },

  bar: function() {
  }
};
```

NO

```
if (foo) {
  return bar;
 }
return baz;

//oppure

var obj = {
  foo: function() {
  },
  bar: function() {
  }
};
return obj;
```

Parentesizzazione 1: i blocchi di codice multi-riga vanno racchiusi tra parentesi graffe. I blocchi di codice con una sola riga possono essere scritti senza parentesi, però vanno scritti sulla stessa riga.

SI

```
if (test) return false;

if (test) {
  return false;
}

function() {
  return false;
}
```

NO

```
if (test)
  return false;

function() { return false; }
```

Parentesizzazione 2: le parantesi graffe iniziano nella stessa riga del codice, non in quella sottostante.

SI

```
if (test) {
  \\..
  if(test2) {
  \\..
    if(test3) {
          \\..
    }
  }
}
```

NO

```
if (test)
{
  if(test2)
  {
    if(test3)
    {
      ...
    }
  }
}
```

Parentesizzazione 3: in caso di blocchi multi-riga con `if` e `else`, bisogna mettere `else` nella stessa riga della parentesi graffa che chiude il blocco `if`.

SI

```
if (test) {
  thing1();
  thing2();
} else {
  thing3();
}
```

NO

```
if (test) {
  thing1();
  thing2();
}
  else {
  thing3();
}
```

Commenti: utilizzare `//` per commenti da una riga. Inserire i commenti di questo tipo su una nuova riga sopra il soggetto del commento. Inoltre bisogna lasciare una riga vuota prima del commento.

SI

```
// comment
var active = true;

function getType() {
  console.log('353');

  // comment
  var type = this._type;

  return type;
}
```

NO

```
var active = true; // comment

function getType() {
  console.log('353');
  // comment
  var type = this._type;

  return type;
}
```

Variabili 1: una variabile deve sempre essere dichiarata usando `var` o `let`. Altrimenti tutte le variabili dichiarate saranno variabili globali. Bisogna sempre evitare di inquinare lo spazio di nomi globale.

SI

```
var superPower = new SuperPower();
```

NO

```
superPower = new SuperPower();
```

Variabili 2: utilizzare una sola dichiarazione `var` o `let` per ogni variabile. È più facile aggiungere dichiarazioni di variabili in questa maniera, e non c'è il rischio di scambiare un `";"` con una `","`. Inoltre, le variabili non assegnate vanno sempre dichiarate per ultime.

SI

```
var objects = getItems();
var cond = true;
var stranger = 'things';
var notAssign;
```

NO

```
var objects = getItems(),
        cond = true,
        notAssign,
        stranger = 'things';
```

Variabili 3: le variabili vanno assegnate o dichiarate solo quando si ha la necessità di usarle. In modo da evitare operazioni inutili.

SI

```
function cName(hasName) {
  if (hasName === 'test') {
    return false;
  }

  const name = getName();

  if (name === 'test') {
    this.setName('');
    return false;
  }

  return name;
}
```

NO

```
function cName(hasName) {
  const name = getName();

  if (hasName === 'test') {
    return false;
  }

  if (name === 'test') {
    this.setName('');
    return false;
  }

  return name;
}
```

Nomi 1: vietato utilizzare nomi delle variabili o delle funzioni con una singola lettera.

SI

```
function query() {
  // ...
}
```

NO

```
function q() {
  // ...
}
```

Nomi 2: utilizzare lo stile `mixedCase` per i nomi delle variabili, funzioni e delle istanze. Inoltre è vietato usare gli underscore nei nomi.

SI

```
const thisIsMyObject = {};
var myVar = "Hello";
function thisIsMyFunction() {}
```

NO

```
const OBJEcttssss = {};
const this_is_my_object = {};
var _myVar_= "Hello";
function c() {}
```

Nomi 3: utilizzare lo stile `CapWords` per i nomi delle classi.

SI

```
class User {
  constructor(options) {
    this.name = options.name;
  }
}
```

NO

```
class user {
  constructor(options) {
    this.name = options.name;
  }
}
```

1.2.3.2 React

React deve seguire tutte le norme viste sopra per il linguaggio di programmazione Javascript. In seguito sono elencate delle norme aggiuntive di JSX tratte da **Airbnb React/JSX Style Guide**[2], da utilizzare solo quando si codifica con React/JSX.

Indentazione 1: va inserito sempre uno (1) spazio nei tag di chiusura.

SI

```
<Foo />
```

NO

```
<Foo/>
```

Proprietà 1: utilizzare sempre lo stile `mixedCase` per i nomi delle proprietà di un tag.

SI

```
<Foo
  userName="hello"
  phoneNumber={12345678}
/>
```

NO

```
<Foo
  UserName="hello"
  phone_number={12345678}
/>
```

Proprietà 2: omettere il valore di una proprietà quando il valore è implicitamente vero (`true`).

SI

```
<Foo
  hidden
/>
```

NO

```
<Foo
  hidden={true}
/>
```

[2] https://github.com/airbnb/javascript/tree/master/react

I tag 1: chiudere tutti i tag che non hanno i tag figli o altro contenuto senza il tag di chiusura.

SI

```
<Foo variant="stuff" />
```

NO

```
<Foo variant="stuff"></Foo>
```

I tag 2: è vietato chiudere un tag con più di una proprietà sulla stessa riga.

SI

```
<Foo
  bar="bar"
  baz="baz"
/>
```

NO

```
<Foo
  bar="bar"
  baz="baz" />
```

r

1.2.3.3 Solidity

In questa sotto-sezione vengono elencate le norme tratte dalla **documentazione officiale di Solidity**[3].

Identazione: vanno utilizzati quattro (4) spazi per ogni livello di indentazione.

Spazi: è vietato l'utilizzo di tabulazioni, che devono essere necessariamente sostituite da spazi. Al fine di assicurare il rispetto di questa regola si consiglia di configurare adeguatamente il proprio editor o IDE.

Linee vuote 1: devono essere lasciate due (2) linee vuote tra la dichiarazione di più smart contract.

[3] http://solidity.readthedocs.io/en/develop/style-guide.html

SI

```
contract A {
    ...
}


contract B {
    ...
}


contract C {
    ...
}
```

NO

```
contract A {
    ...
}
contract B {
    ...
}

contract C {
    ...
}
```

Linee vuote 2: le linee vuote possono essere omesse tra le dichiarazioni di funzioni in linea.

SI

```
contract A {
    function spam();
    function ham();
    }


contract B is A {
    function spam() {
        ...
    }

    function ham() {
        ...
    }
}
```

NO

```
contract A {
    function spam() {
        ...
    }
    function ham() {
        ...
    }
}
```

Codifica dei codici sorgenti: i codici sorgente devono essere codificati in UTF-8.

Imports: tutte le dichiarazioni di `import` vanno fatte all'inizio del file.

SI

```
import "owned";


contract A {
    ...
}


contract B is owned {
    ...
}
```

NO

```
contract A {
    ...
}


import "owned";


contract B is owned {
    ...
}
```

Ordine delle funzioni: all'interno di un contratto l'ordine è fondamentale per identificare meglio quali funzioni è possibile chiamare e per trovare più velocemente le definizioni del costruttore e di eventuali funzioni di fallback.
Le funzioni devono essere raggruppate secondo la loro visibilità e ordinate nel seguente modo:

1. costruttore
2. funzioni di fallback (se esistono)
3. esterne
4. pubbliche
5. interne
6. private

In ogni gruppo le funzioni costanti vanno dichiarate per ultime.

SI

```
contract A {
    function A() {
        ...
    }

    function() {
        ...
    }

    // External functions
    // ...

    // External functions
    // that are constant
    // ...

    // Public functions
    // ...

    // Internal functions
    // ...

    // Private functions
    // ...
}
```

NO

```
contract A {

    // External functions
    // ...

    // Private functions
    // ...

    // Public functions
    // ...

    function A() {
        ...
    }

    function() {
        ...
    }

    // Internal functions
    // ...
}
```

Spazi bianchi nelle espressioni: non vanno messi spazi bianchi nelle seguenti situazioni:

- immediatamente dopo l'apertura o la chiusura di una parentesi tonda, quadrata o graffa con un'espressione di una singola riga

SI

```
spam(ham[1], Coin({name: "ham"}));
```

NO

```
spam( ham[ 1 ], Coin( { name: "ham" } ) );
```

ECCEZIONE

```
function singleLine() { spam(); }
```

- immediatamente prima di una virgola o di un punto e virgola

SI

```
function spam(uint i, Coin coin);
```

NO

```
function spam(uint i , Coin coin) ;
```

- prima o dopo un'assegnazione o con altri operatori per permettere l'allineamento con altre istruzioni

SI

```
x = 1;
y = 2;
long_variable = 3;
```

NO

```
x             = 1;
y             = 2;
long_variable = 3;
```

- nella dichiarazione di funzioni fallback

SI

```
function() {
    ...
}
```

NO

```
function () {
    ...
}
```

Strutture di controllo: le parentesi graffe che denotano l'apertura di un contratto, una libreria o una funzione devono:

- essere aperte nella stessa riga della dichiarazione e precedute da uno spazio bianco;
- chiuse nello stesso livello di indentazione della dichiarazione iniziale.

SI

```
contract Coin {
    struct Bank {
        address owner;
        uint balance;
    }
}
```

NO

```
contract Coin
{
    struct Bank {
        address owner;
        uint balance;
    }
}
```

Le stesse regole valgono anche per strutture di controllo come `if`, `else`, `while` e `for`.
Inoltre, deve essere messo un singolo spazio tra la dichiarazione della struttura di controllo `if`, `while` e `for` e la condizione e anche tra quest'ultima e l'apertura della parentesi graffa.

SI

```
if (...) {
    ...
}

for (...) {
    ...
}
```

NO

```
if (...)
{
    ...
}

while(...){
}

for (...) {
    ...;}
```

Per le strutture di controllo che nel corpo hanno una singola istruzione è possibile omettere le parentesi graffe solo se l'istruzione è contenuta in una singola linea.

SI

```
if (x < 10)
    x += 1;
```

NO

```
if (x < 10)
    someArray.push(Coin({
        name: 'spam',
        value: 42
    }));
```

È consentita una sola eccezione: nei blocchi di `if` che contengono clausole di `else` o `else if`, l'`else` deve stare nella stessa linea della chiusura del blocco `if`.

SI

```
if (x < 3) {
    x += 1;
} else if (x > 7) {
    x -= 1;
} else {
    x = 5;
}


if (x < 3)
    x += 1;
else
    x -= 1;
```

NO

```
if (x < 3) {
    x += 1;
}
else {
    x -= 1;
}
```

Dichiarazione di funzioni: per dichiarazioni brevi, la dichiarazione della funzione va lasciato uno spazio bianco prima dell'apertura della graffa sulla stessa linea. La graffa che chiude il corpo della funzione va chiusa allo stesso livello di indentazione della dichiarazione iniziale.

SI

```
function increment(uint x) returns (uint) {
    return x + 1;
}

function increment(uint x) public onlyowner returns (uint) {
    return x + 1;
}
```

NO

```
function increment(uint x) returns (uint)
{
    return x + 1;
}

function increment(uint x) returns (uint){
    return x + 1;
}

function increment(uint x) returns (uint) {
    return x + 1;
}

function increment(uint x) returns (uint) {
    return x + 1;}
```

La visibilità di una funzione va dichiarata prima di altri modificatori.

SI

```
function kill() public onlyowner {
    selfdestruct(owner);
}
```

NO

```
function kill() onlyowner public {
    selfdestruct(owner);
}
```

Per dichiarazioni lunghe, ogni argomento va dichiarato in una linea allo stesso livello di indentazione del corpo della funzione. La parentesi di chiusura della dichiarazione degli argomenti va collocata allo stesso livello di indentazione della dichiarazione della funzione.

SI

```
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f
) {
    doSomething();
}
```

NO

```
function thisFunctionHasLotsOfArguments(address a, address b,
    address c, address d, address e, address f) {
    doSomething();
}

function thisFunctionHasLotsOfArguments(address a,
                                        address b,
                                        address c,
                                        address d,
                                        address e,
                                        address f) {
    doSomething();
}

function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) {
    doSomething();
}
```

Se una lunga dichiarazione ha molti modificatori, ciascuno di essi deve essere collocato in una linea.

SI

```
function thisFunctionNameIsReallyLong(address x, address y)
    public
    onlyowner
    priced
    returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(
    address x,
    address y,
    address z
)
    public
    onlyowner
    priced
    returns (address)
{
    doSomething();
}
```

NO

```
function thisFunctionNameIsReallyLong(address x, address y)
                                      public
                                      onlyowner
                                      priced
                                      returns (address) {
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y)
    public onlyowner priced returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y)
    public
    onlyowner
    priced
    returns (address) {
    doSomething();
}
```

Per costruttori o per contratti estesi che richiedono degli argomenti, i costruttori di base vanno dichiarati uno per linea.

SI

```
contract A is B, C {
    function A(uint param1, uint param2, uint param3, uint param4)
        B(param1)
        C(param2, param3)
    {
        // do something with param4
    }
}
```

NO

```
contract A is B, C {
    function A(uint param1, uint param2, uint param3, uint param4)
    B(param1)
    C(param2, param3)
    {
        // do something with param4
    }
}

contract A is B, C {
    function A(uint param1, uint param2, uint param3, uint param4)
        B(param1)
        C(param2, param3) {
       // do something with param4
    }
}
```

È possibile dichiarare funzioni in una sola linea solo se possiedono una singola istruzione.

```
function shortFunction() { doSomething(); }
```

Dichiarazione di variabili: nella dichiarazione di array non devono essere messi spazi tra il tipo e le parentesi quadrate.

SI

```
uint[] x;
```

NO

```
uint [] x;
```

Le stringhe vanno dichiarate solo utilizzando i doppi apici.

SI

```
str = "foo";
str = "Hamlet says, 'To be or not to be...'";
```

NO

```
str = 'bar';
str = '"Be yourself; everyone else is already taken." -O.W.';
```

Operatori: lasciare uno spazio tra un operatore e una variabile.

SI

```
x = 3;
x = 100 / 10;
x += 3 + 4;
x |= y && z;
```

NO

```
x=3;
x = 100/10;
x += 3+4;
x |= y&&z;
```

Per una maggiore leggibilità e comprensione del codice la norma ha come eccezione gli operatori con maggiore priorità in istruzioni composte da più operazioni.

SI

```
x = 2**3 + 5;
x = 2*y + 3*z;
x = (a+b) * (a-b);
```

NO

```
x = 2** 3 + 5;
x = y+z;
x +=1;
```

Convenzioni per i nomi:

- contratti e librerie: seguono il `CapWords` style.
  es. `SimpleToken`, `SmartBank`, `CertificateHashRepository`, `Player`;
- eventi: seguono il `CapWords` style.
  es. `Deposit`, `Transfer`, `Approval`, `BeforeTransfer`, `AfterTransfer`;
- funzioni: seguono il `mixedCase` style.
  es. `getBalance`, `transfer`, `verifyOwner`, `addMember`, `changeOwner`;
- argomenti di funzioni: seguono il `mixedCase` style.
  es. `initialSupply`, `account`, `recipientAddress`, `senderAddress`, `newOwner`;
- variabili locali: seguono il `mixedCase` style.
  es. `totalSupply`, `remainingSupply`, `balanceOf`, `creatorAddress`, `isPreSale`;
- costanti: seguono il `CAPITAL_CASE_WITH_UNDERSCORE` style.
  es. `MAX_BLOCKS`, `TOKEN_NAME`, `TOKE_TICKET`;
- modificatori: seguono il `mixedCase` style.
  es. `onlyBy`, `onlyAfter`, `onlyDuringThePreSale`;

1.2.3.4 Scss

In questa sotto-sezione vengono elencate le norme tratte dalla **Airbnb CSS/Scss Styleguide**[4].

[4] https://github.com/airbnb/css

Formattazione:

- utilizzare due (2) spazi per ogni livello di indentazione;
- per i nomi delle classi utilizzare la `lower-case-with-dashes`;
- non usare selettori ID;
- in caso di selettori multipli per una regola, mettere ogni selettore in una linea singola;
- lasciare uno spazio prima dell'apertura della parentesi graffa di una regola;
- nelle proprietà lasciare uno spazio dopo e non prima i due punti;
- chiudere la parentesi graffa di una regola in una nuova linea;
- lasciare una linea vuota tra una dichiarazione e l'altra.
- dichiarare le classi raggruppate per tipologia e in ordine alfabetico.

SI

```
.avatar {
  border-radius: 50%;
  border: 2px solid white;
}

.one,
.per-line,
.selector
 {
  // ...
}
```

NO

```
.wrong {
  // ...}
.avatar{
  border-radius:50%;
  border:2px solid white; }
.no, .nope, .not_good {
  // ...
}
#lol-no {
  // ...
}
```

BEM: utilizzare la notazione BEM (Block-Element-Modifier) come convenzione per i nomi delle classi.

```
// ListingCard.jsx
function ListingCard() {
  return (
    <article class="listingcard listingcard-featured">

      <h1 class="listingcard-title">Adorable Mission</h1>

      <div class="listingcard-content">
        <p>Vestibulum id ligula porta euismod semper.</p>
      </div>

    </article>
  );
}
```

```
/* listingcard.css */
.listingcard { }
.listingcard-featured { }
.listingcard-title { }
.listingcard-content { }
```

In questo esempio:

- `.listingcard` è il 'blocco' e rappresenta il livello più alto del componente;
- `.listingcard-title` è un 'elemento' e rappresenta un discendente di `.listingcard`;
- `.listingcard-featured` è un 'modificatore' e rappresenta un diverso stato o una variante del blocco `.listingcard`;

Commenti:

- non usare commenti di blocco;
- fare commenti in linee a se stanti, non in linee contenenti anche codice.

Ordine di dichiarazione delle proprietà:

1. **proprietà standard:** vanno dichiarate per prime e in ordine alfabetico;

```
.btn-green {
  background: green;
  font-weight: bold;
  // ...
}
```

2. **dichiarazioni di `@include`:** vanno raggruppate sotto alla dichiarazione delle proprietà standard

```
.btn-green {
  background: green;
  font-weight: bold;
  @include transition(background 0.5s ease);
  // ...
}
```

3. **selettori nidificati:** vanno dichiarati dopo tutte le altre dichiarazioni e deve essere lasciata una linea vuota prima della loro dichiarazione

```
.btn {
  background: green;
  font-weight: bold;
  @include transition(background 0.5s ease);

  .icon {
    margin-right: 10px;
  }
}
```

Extend: la direttiva `@extend` non va utilizzata perché ha un comportamento poco intuitivo e pericoloso, specialmente quando usato in selettori nidificati.

Selettori annidati: non utilizzare più di tre livelli di annidamento

```
.page-container {
  .content {
    .profile {
      // STOP!
    }
  }
}
```

1.2.4 Procedure

1.2.4.1 Tracciamento componenti-requisiti

1.2.5 Strumenti

- Tracciamento requisiti
- Creazione diagrammi UML
- Tracciamento dei test
- Scrittura del codice
- Esecuzione dei test
- Continuos Integration
- Analisi statica del codice