**Capitolo 3**

**Software Linguistic Antipatterns e Code Smells in IaC**

## Introduzione

Studi recenti e passati hanno mostrato che utilizzare un lessico poco chiaro all’interno di codice sorgente influisce negativamente sulla comprensibilità, manutenibilità e, in generale, sulla qualità del codice software prodotto. Oltre ad uno scarso utilizzo di termini adatti e documentazione, una descrizione di un artefatto software può fuorviare i manutentori dalla sua reale funzione. Di conseguenza, gli sviluppatori impiegheranno più tempo e sforzi per comprendere porzioni di codice. Ciò potrebbe anche portare, come detto in precedenza, ad incomprensioni circa l’effettivo compito svolto da una parte di codice. Questo capitolo va a definire il concetto di *Linguistic Antipattern* applicato ai software.

Mentre gli antipattern di design rappresentano apparenti soluzioni ricorrenti, gli antipattern linguistici non rappresentano altro che scelte di denominazione e descrizione imprecise o sbagliate.[[1]](#footnote-1)

Nel capitolo seguente fornisce una categorizzazione degli antipattern linguistici, spiegando il tipo di incomprensione che l’antipattern in questione può causare. Infine, si applica il concetto di antipattern linguistico al contesto dell’Infrastructure Code.

## Software Linguistic Antipatterns: definizione e classificazione

Un antipattern linguistico è una cattiva pratica che riguarda le entità dei software, la loro nomenclatura e la loro documentazione. Una famiglia di antipattern linguistici su tutte è quella relativa alle incongruenze tra la denominazione, la documentazione e l’implementazione di un’entità software. In questo paragrafo sono presentate sei categorie di antipattern linguistici che riguardano specificamente il design del software: tre di esse riguardano il comportamento – metodi – e tre sono relative allo stato – attributi. Nello specifico, sono fornite le seguenti informazioni:

* Nome dell’antipattern linguistico;
* Descrizione.[[2]](#footnote-2)

### Fa più di ciò che dichiara - *Does more than it says*

Come la denominazione suggerisce, ricadono in questa categoria gli antipattern linguistici che effettuano, ad esempio, più operazioni di ciò che indica il nome di un metodo o di una procedura.

Di questa categoria fanno parte i seguenti antipattern:

* *“Get” is more than an accessor:* specificamente, in Java, i metodi accessori (chiamati anche *getters*), forniscono l’accesso in lettura agli attributi di una classe. Molto spesso, però, il getter effettua altre operazioni oltre a restituire l’attributo richiesto. Ogni altra azione dovrebbe essere documentata correttamente e, possibilmente, il naming del metodo dovrebbe modificato in qualcosa di diverso da, ad esempio, *getSomething;*
* *“Set” method returns:* i modificatori, ovvero i setters, sono metodi che consentono l’assegnazione di un valore ad un attributo di una classe che, normalmente, dovrebbe non essere accessibile dall’esterno della classe. Per convenzione, i setters non dovrebbero restituire alcun valore. Quindi, ogni setter con un tipo di ritorno differente da *void*, dovrebbe essere correttamente documentato;
* *“is” returns more than a boolean:* quando un metodo ha il nome che inizia per “*is*”, ci si aspetta che tale metodo restituisca un valore di tipo *Boolean*, con due possibili valori: *true* e *false.* Quindi, avere un metodo di questo tipo che non restituisce un valore booleano potrebbe risultare fuorviante. In questo caso, potrebbe essere necessario rinominare il metodo o almeno documentare il fatto che restituisce un valore differente da un *boolean*.
* *Expecting but not getting a single instance:* quando il nome di un metodo indica che sarà restituito un singolo oggetto, dovrebbe essere coerente col suo tipo di ritorno. Se, invece, il tipo di ritorno è una collezione di oggetti e non un oggetto singolo, il metodo dovrà essere rinominato o ben documentato a riguardo.[[3]](#footnote-3)

### Dichiara più di ciò che fa – *Says more than it does*

In questa categoria ricadono gli antipattern i cui nomi illudono circa la loro mansione:

* *Not implemented condition:* si ha quando i commenti suggeriscono un comportamento condizionale, mentre il codice non lo implementa.
* *Validation method does not confirm:* si ottiene quando un metodo validatore non ritorna un valore per confermare la validazione.
* *“Get” method does not return:* mentre il nome del metodo getter suggerisce che si avrà qualche tipo di ritorno, il metodo stesso non restituisce alcun elemento.
* *Not answered question:* il nome del metodo è nella forma “*is*”, che suggerisce un tipo di ritorno *boolean*, mentre il metodo non restituisce alcun valore booleano.
* *Transform method does not return:* il nome del metodo suggerisce che esso trasforma un oggetto, ma in realtà non restituisce alcun valore.
* *Expecting but not getting a collection:* il nome del metodo suggerisce che esso restituirà una collezione, ma esso ritorna o un singolo oggetto o nulla.[[4]](#footnote-4)

### Fa l’opposto – *does the opposite*

Come il nome della categoria suggerisce, questa classe raggruppa gli antipattern che fanno l’opposto di ciò che un metodo dichiara.

* *Method name and return type are opposite:* l’intenzione del metodo suggerita dal suo nome è in contraddizione con ciò che restituisce.
* *Method signature and comment are opposite:* la documentazione di un metodo è in contraddizione con la sua dichiarazione (nome e return type).

### Contiene più di ciò che dichiara – *contains more than it says*

* *Says one but contains many:* si ha quando il nome di un attributo suggerisce che di esso sia presente una singola istanza, quando, in realtà, il suo tipo indica che l’attributo è una collezione di oggetti.
* *Name suggests boolean but type does not:* il nome di un attributo suggerisce che il suo valore sarà un *boolean,* ma la dichiarazione del suo tipo è diversa da *Boolean.[[5]](#footnote-5)*

### Dichiara più di ciò che contiene – *Says more than it contains*

* *Says many but**contains one:* il nome di un attributo suggerisce che esso contenga una collezione di oggetti, ma il suo tipo indica che esso ne conterrà solo uno.[[6]](#footnote-6)

### Contiene l’opposto – *Contains the opposite*

* *Attribute name and type are opposite*: il nome di un attributo è in contraddizione col suo tipo.
* *Attribute signature and comment are opposite:* la documentazione dell’entità è in contraddizione con la sua dichiarazione.[[7]](#footnote-7)

## Code smells in Infrastructure as Code

L’espressione *Code Smell* è usata per indicare una serie di caratteristiche che il codice sorgente può avere e che sono generalmente riconosciute come probabili indicazioni di un difetto di programmazione. È un concetto strettamente collegato a quello di Linguistic Antipattern: entrambi non si riferiscono e non rivelano *bugs*, ma bensì debolezze di progettazione che riducono la qualità del codice, a prescindere dall’effettiva correttezza del suo funzionamento.

Nelle pratiche tradizionali di ingegneria del software, i *bad smells* sono classificati come *implementation (code) smells, design smells* e *architectural smells,* in base alla granularità dell’astrazione nel quale lo *smell* si presenta e colpisce.

Studi precedenti, hanno descritto la qualità del codice infrastrutturale in termine di *smelliness* e tendenza ai difetti dei componenti di *Chef* e *Puppet*. Da un punto di vista di *smelliness*, Schwarz et al., Spinellis et al. And Rahman et al. hanno applicato il *well-known concept* a IaC, identificando *code smells* che possono essere suddivisi in quattro gruppi:

* *Implementation Configuration,* come, ad esempio, espressioni complesse e statements superati;
* *Design Configuration,* come gerarchie spezzate e blocchi duplicati;
* *Security smells,* come impostazione utente come admin di default e *hard-coded secrets,* ovvero codici di accesso scritti in chiaro all’interno di files di configurazione;
* *General smells,* di cui risorse eccessivamente lunghe e/o con numerosi attributi ne fanno parte.[[8]](#footnote-8)

### Implementation Configuration Smells

Gli *Implementation Configuration Smells* sono problemi riguardanti la qualità del codice di configurazione, come ad esempio la *naming convention*, lo stile, la formattazione e l’indentazione. Di seguito sono listati alcuni degli smells che ricadono in questo gruppo, con una breve descrizione:

* *Inconsistent Naming Convention,* nel qualela scelta della *naming convention* devia da quella raccomandata dalla documentazione ufficiale del tool utilizzato;
* *Complex expression.* Un task contiene un’espressione o una serie di operazioni complesse e difficili da comprendere;
* *Duplicate Entry.* I parametri o le hash key duplicati/e presenti nel codice di configurazione;
* *Misplaced Attribute.* Il posizionamento errato degli attributi all’interno di una risorsa o di una classe (es. è consigliato specificare gli attributi obbligatori prima di quelli opzionali);
* *Improper Alignment.* Il codice non è correttamente allineato o sono stati utilizzati caratteri di tabulazione;
* *Invalid property value.* Utilizzo di un valore non valido di una proprietà o di un attributo;
* *Incomplete tasks.* Il codice di configurazione presenta tasks con tags “FIXME” e/o “TODO”, i quali indicano task incompleti;
* *Deprecated Statement Usage.* Il codice di configurazione usa uno degli statements deprecati;
* *Long Statement.* Il codice contiene statement eccessivamente lunghi;
* *Incomplete conditional.* Cioè quando il codice contiene un’istruzione condizionale non completa.[[9]](#footnote-9)

### Design Configuration Smells

I *Design Configuration Smells* rilevano problemi di qualità nel codice di progettazione di un modulo o di una struttura di un progetto di configurazione. È possibile reperire sulla documentazione ufficiale di qualsiasi tool IaC materiale da utilizzare per documentarsi sulle *best practices* per il design e la progettazione di files di configurazione, proprio per evitare problemi di natura *smelly*. Questo sottoparagrafo contiene una lista di alcuni dei *design configuration smells,* ognuno accompagnato da una breve descrizione.

* *Multifaceted Abstraction.* Ogni astrazione (i.e. una risorsa, una classe o un modulo) dovrebbe essere progettata per specificare le proprietà di un singolo pezzo di software. In altre parole, ogni astrazione dovrebbe seguire *il principio di singola responsabilità.* Un’astrazione soffre di un’astrazione multiforme quando gli elementi dell’astrazione non sono coesi;
* *Unnecessary Abstraction.* Una classe o un modulo deve contenere dichiarazioni o statements che specificano le proprietà del sistema desiderato. Una classe vuota o un modulo mostra la presenza di tale smell e dev’essere necessariamente rimossa/o;
* *Missing Abstraction.* Le dichiarazioni delle risorse e degli statements sono facili da fare e riutilizzare quando sono incapsulate in un’astrazione, come la classe o il ‘define’. Un modulo soffre di tale *smell* quando le risorse e gli elementi del linguaggio sono dichiarati e usati senza essere incapsulati in un’astrazione;
* *Insufficient Modularization.* Un’astrazione soffre di tale *smell* quando essa è larga e complessa e, di conseguenza, potrebbe essere *modularizzata*.
* *Duplicate Block.* Un blocco duplicato di statements indica che, probabilmente, non è stata applicata un’astrazione adatta. [[10]](#footnote-10)

### Security Smells

I *Security Smells* sono pattern di codice ricorrente che rappresentano indicatori di debolezze relative alla sicurezza che richiedono una particolare attenzione.

Gli *smells* elencati di seguito riguardano i tools *Ansible* e *Chef.*

* *Admin by default.* Questo *smell* è un modello ricorrente per specificare gli utenti predefiniti come admin. Tale smell può violare il *principio* *del privilegio minimo*, che raccomanda di progettare e implementare un sistema in modo tale che, per impostazione predefinita, venga fornito a qualsiasi entità la quantità minima di accesso necessaria.
* *Empty password.* Questo *smell* è un modello ricorrente di utilizzo di una stringa di lunghezza zero per una password. Una password vuota è indice di una password indubbiamente debole.
* *Hard-coded secret.* Questo smell riguarda la rivelazione di informazioni sensibili, come l’username e la password di qualche servizio, all’interno di scripts IaC. Tali scripts offrono l’opportunità di specificare intere configurazioni di sistema, come la configurazione delle credenziali di accesso, l’impostazione delle chiavi SSH per gli utenti e la specifica dei files di autenticazione.
* *Missing Default in Case Statement.* Tale *smell* è il pattern ricorrente riguardo la non gestione di tutte le combinazioni di input quando si implementa una logica condizionale. A causa di questo pattern, un utente malintenzionato può indovinare un valore (che non è gestito dalle istruzioni condizionali del caso) e causare un errore. Questo errore potrebbe fornire all’attaccante informazioni per il sistema in termini di stack traces o un system error.
* *No integrity check.* Questo pattern riguarda il download di contenuti da internet e il non controllo del materiale scaricato utilizzando il checksum o le firme gpg. Non controllando l’integrità, il developer presume che il contenuto sia sicuro e non sia stato compromesso da un potenziale aggressore. Il controllo d’integrità fornisce un ulteriore livello di sicurezza per garantire che il contenuto scaricato sia intatto.
* *Suspicious comment.* Questo *smell* riguarda l’inserimento di informazioni circa la presenza di difetti, funzionalità mancanti o debolezze del sistema all’interno dei commenti del codice di configurazione. Parole chiave come “TODO” e “FIXME” nei commenti vengono utilizzate per specificare un caso limite o un problema.
* *Use of weak cryptography algorithms.* Tale *smell* riguarda l’utilizzo di un algoritmo di crittografia debole, vale a dire MD5 e SHA-1. È risaputo che MD5 soffra di problemi di sicurezza, come dimostrato dal malware Flame nel 2012. Anche SHA-1 è suscettibile ad attacchi di collisione, quindi, l’utilizzo di algoritmi deboli per l’hashing potrebbe portare ad una violazione e a problemi di sicurezza.

1. Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antonio, Yann-Gael Gueheneuc, *A New Family of Software Anti-Patterns: Linguistic Anti-Patterns*, 2013 [↑](#footnote-ref-1)
2. Op. cit. pag. 2. [↑](#footnote-ref-2)
3. Op. cit. pag. 2. [↑](#footnote-ref-3)
4. Op. cit. pag. 2. [↑](#footnote-ref-4)
5. Op. cit. pag. 2. [↑](#footnote-ref-5)
6. *ibidem* [↑](#footnote-ref-6)
7. Op. cit. pag. 2. [↑](#footnote-ref-7)
8. N. Borovits, I. Kumara, P. Krishnan, S. Dalla Palma, D. Di Nucci, F. Palomba, D. A. Tamburri, W-J. van den Heuvel, *DeepIaC: Deep Learning-Based Linguistic Anti-pattern Detection in IaC,* 2020 [↑](#footnote-ref-8)
9. T. Sharma, M. Fragkoulis, D. Spinellis, *Does Your Configuration Code Smell?,* 2016 [↑](#footnote-ref-9)
10. Op. cit. p. 11 [↑](#footnote-ref-10)