Linköping University | Department of Computer and Information Science

Master thesis, 30 ECTS | Datateknik

2019 | LIU-IDA/LITH-EX-A--19/059--SE

Measuring Architectural Degeneration

– In Systems Written in the Interpreted Dynamically Typed Multi-Paradigm Language Python

Anton Mo Eriksson & Hampus Dunström

Supervisor : Anders Fröberg
Examiner : Erik Berglund

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

Architectural degeneration is an ever-present threat to software systems with no exception based on the domain or tools used. This thesis focus on the architectural degeneration in systems written in multi-paradigm run-time evaluated languages like Python. The focus on Python in this kind of investigation is to our knowledge the first of its kind; thus the thesis investigates if the methods for measuring architectural degeneration also applies to run-time evaluated languages like Python as believed by other researchers. Whom in contrast to our research have only researched this phenomenon in systems written in compiled languages such as Java, C, C++ and C#. In our research a tool PySmell has been developed to recover architectures and identify the presence of architectural smells in a system. PySmell has been used and evaluated on three different projects Django, Flask and PySmell itself. The results of PySmell are promising and of great interest but in need of further investigating and fine-tuning to reach the same level as the architectural recovery tools available for compiled languages. The thesis presents the first step into this new area of detecting architectural degeneration in interpreted languages, revealing issues such as that of extracting dependencies and how that may affect the architectural smell detection.

Acknowledgments

We would like to thank our supervisors from both Linköpings University and FindOut Technologies for believing in us and supporting us when everything did not go our way. Anders Fröberg at Linköping University has always been there for us, especially for questions regarding the report and academic requirements. David Lindahl and Nils Kronqvist at FindOut Technologies, thank you for our weekly meetings were you have helped us keep a continuous workflow and give another perspective on our work. We are also thankful to Marco Kuhlmann, a machine learning and language technology researcher at Linköping University, for his time and insights into our work in machine learning. Last but not least, Helena Gällerdal you have made our time at FindOut Technologies brighter with your energy and commitment towards us, thank you!

Contents

List of Figures

List of Tables

1 Introduction

Since the early days of humanity, the urge to describe things with drawings has prevailed and been an integrated part of our culture and evolution. From Da Vinci to Turing, one of the renaissance most distinguished engineers [1] to the father of modern software [2], both utilising architectures to describe their masterpieces. We see that the importance for a medium such as an architecture to efficiently communicate an thought of what is to be engineered, have been used for a long time.

Moving forward to software development, the demand for an abstract, high-level view of the system to develop or maintain is of vital importance especially when the software grows [3]. The need for sound architectures that are adaptable to changes introduced by ever-changing demands on the software is also important today [4]. In a software system with an ever-changing implementation, it is essential to have a changeable architecture which is continuously revised to match the de-facto implementation of the system [5]. An interesting question then arises when a system undergoes continues evolution, is this still the same system? This is the same question Theseus raised in his Paradox[1]. To further discuss this phenomenon of architecture changes in particular changes that goes against the intended architecture Hochstein and Lindvall coined the term architectural degeneration to describe how an architecture changes for the worse over time [5].

Over time, software systems architectures degenerate, as changes are made and more features are added [6]. The degeneration of software architectures can have devastating effects. An example of this is brought up by Godfrey and Lee, they investigated the architecture of the Mozilla browser Netscape and found that its architecture had decayed in a very short period of time or was not very well thought through to begin with [7]. Either way it resulted in that the browser had to be re-written from scratch in 1998, during a period of 5 five years two million LOC had to be re-written [8]. This process of architectural degeneration is a natural process and unless a special effort is put in to prevent it manifests itself into a system as the system grows and evolves [9]. To measure architectural degeneration, architectural smells

[1] Theseus Paradox – *"is this the same ship?"*

have been used in previous research [10]. These architectural smells can be thought of as indications of a flawed architecture much like how symptoms are indications of a disease.

Architectural degeneration, erosion, decay or debt; this issue has many different names depending on context, perspective and author as all of the synonyms has been investigate by the academic community. How to avoid it [9], how to diagnose it [5], how it affects the maintenance [10]. What these different investigations have in common is that they are all performed on systems written in C, C++, Java or C#. This is not very strange since the industry have been and still are dominated by these languages as can be seen in the TIOBE Index[2] that is a popularity index for programming languages.

In the TIOBE Index, it can be seen that there are a few newcomers such as Python and JavaScript which are dynamically typed and interpreted languages compared to the statically typed and compiled languages C, C++ Java and C# used in existing research. With these dynamically typed and interpreted languages it is interesting to see if the techniques for recovering architectures and measuring degeneration applies to systems written in these type of languages. This is were our research tries to contribute.

1.1 Purpose

To strengthen the research on architectural degeneration. Reinforcing the conclusions earlier researcher has drawn regarding language independence of architectural recovery techniques with a focus on the dynamically typed and interpreted languages.

1.2 Research Question

The aim of this study is to answer the following research question:

> *How does architectural degeneration evolve in large scale systems written in the dynamically typed multi-paradigm and interpreted language Python?*

1.3 Delimitations

To limit the scope of our study we have chosen to only use one kind of architectural recovery technique. This was done because we only had 20 weeks to complete our research, study and report. Since no previous research into recovering architectures or measuring degeneration in systems written in Python was found, we had to create our own tools. Implementing architectural recovery and architectural degeneration detection techniques takes time. This limited the amount of techniques we could cover and the amount of validation we could perform.

1.4 Expected Results

We expected to create a tool that can measure architectural degeneration with some degree of certainty. Using this tool we then believed that we would be able to measure architectural degeneration at multiple stages in a systems development. From those measurement we thought there would be a increase in architectural degeneration as a system grows larger over time.

[2] https://www.tiobe.com/tiobe-index/ Accessed 2019-02-18

1.5 FindOut Technologies

The company we worked with was FindOut Technologies, a small company according to the EU definition [11]. They develop tools for software development and consults other companies within software development. There were a bit over 30 employees working at FindOut Technologies at the time of writing this thesis. As consultants, they help other companies create new and update their existing software. In house, FindOut develops visual tools for understanding existing software architectures. These tools are both standalone products and used by FindOut in their consultant work.

2 Theory

This chapter aims to give the reader significant background knowledge, which is needed to comprehend the thesis.

2.1 Architecture

To discuss the term architecture, a clear and widely adopted definition of an architecture in the context of this thesis is adopted from the IEEE Standard 1471 is used.

> "*The fundamental organisation of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.*" [12]

With a definition of an architecture as a set of components, connections and constrains as well as the needs of the stakeholders and a reasoning for how the implementation of the components, connections and constraints will satisfy the stakeholders needs [13]. An architecture can be described as in figure 2.1. Each component is made up of software entities such as implementation classes and functions [10]. The connections are defined by dependencies between different entities. In "Comparing software architecture recovery techniques using accurate dependencies" Lutellier *et al.* compare symbol dependencies with include dependencies to see which gives the best result when recovering an architecture [14]. Two entities have a symbol dependency when a function or class calls upon another function or class method, then there is a symbolic dependency from the caller to the called. Include dependencies are defined by the import or include statements in a file. If a file has a include statement including another file there is a dependency between those two files. The conclusion Lutellier *et al.* draws is that symbolic dependencies have a finer granularity than include dependencies and that gives a better results during architectural recovery [14]. A connection between two entities will therefore in our report be defined by a symbolic dependency between two entities.

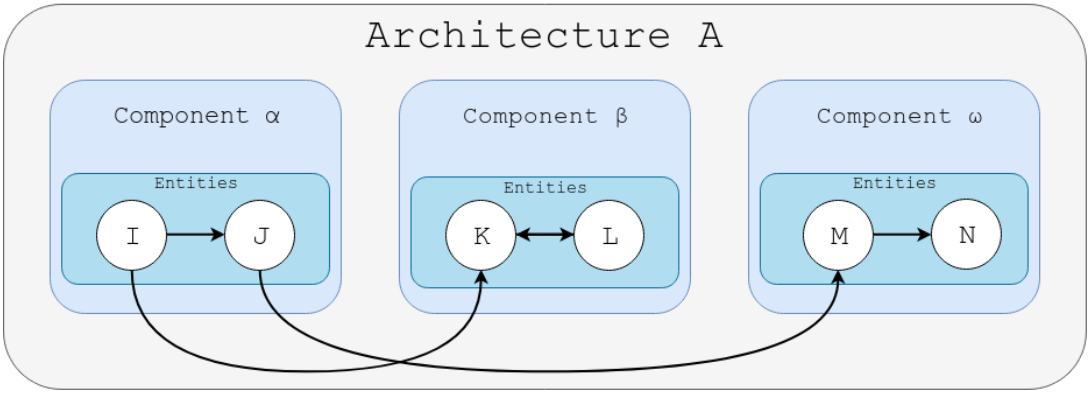

Figure 2.1: Example of basic architecture building blocks.

2.2 Architectural Degeneration

Architectural Degeneration is a term for when a software implementations structure worsens due to changes made as the software is maintained and new features are added resulting in more source code, new components, increased coupling and complexity [9]. The result of architectural degeneration is that code becomes harder to change and new features become more and more costly [5]. This is quite easy to understand in big systems with many developers coming and going but it can also occur in smaller projects as shown by Tvedt *et al.* in "Does the code match the design? A process for architecture evaluation" where a project with between 10k - 12k source lines of code (SLOC) showed architectural degeneration[15]. One of the causes of architectural degeneration that are raised by multiple different papers, are new developers adding new features without having understood the intended architecture [6], [16], [17].

We will use the term Architectural degeneration in this thesis but in the literature it has many related semi-synonyms;

- Architecture Erosion – The phrase erosion refers to when an explicit architecture decision is violated, either unintentionally or intentionally [18] [19].
- Architecture Degradation – Degradation of the architecture focus on the effect source code smells has on the evolving de-facto architecture compared to the intended architecture [20].
- Architecture Debt – The term debt is often used to relate the abstract term opus-refactoring, to able to explain to none-engineer why refactoring is a crucial part in maintaining a large software system. Where the analogies between a monetary debt and the postponed refactoring are made to make interdisciplinary communication possible [21].
- Architecture Drift – A phenomenon that occurs when the architect does not enforces the architecture decision made [18] [19].
- Architecture Decay – An decay in the architecture during its progression is equivalent to a decrease of the quality regarding the sustainability of the system [22].

2.3 Architectural Smells

The term architectural smell is an indicator that anti-patterns or that bad practices has gotten a foothold during the evolution of a software system [10]. There are several architectural

smells defined in the literature, the ones used in this thesis are presented and explained in this section.

2.3.1 Dependency Cycles

The dependency cycle smell addresses circular links between components in the software system. Cyclical dependencies are hazardous for several reasons. Firstly, in a cyclical situation, there is a high probability that a misinterpretation of the design has been made. Secondly, the maintainability implication effects of using cyclical structure are devastating, whereas a change in the cyclical dependencies chain may affect all components involved in the chain. [10]

It is measured in the following way:

$$link(\mathcal{C}_i, \mathcal{C}_j) \wedge link(\mathcal{C}_j, \mathcal{C}_k) \wedge ... \wedge link(\mathcal{C}_k, \mathcal{C}_i) \tag{2.1}$$

where: $link(\mathcal{A}, \mathcal{B})$: if $\mathcal{A}$ and $\mathcal{B}$ are connected by dependencies.
$\mathcal{C}_\alpha$: the component number α.
α : the set of components, $\{i, j, k\}$.

With the previous mentioned formula in mind, we see that for a dependency cycle to occur, there needs to be at least three components in the architecture.

2.3.2 Link Overload

The phenomenon of link overload occurs when an excessive amount of dependencies are connected to one component, where dependencies come in two different types; *in-going* dependencies on the overloaded components, and *out-going* the overloaded components dependencies. The term excessive is a predefined threshold (see 2.4) for when the composed smell becomes a fact. [10]

The process of measuring the link overload is achieved by the formula below:

$$\sum_{c}^{\mathcal{C}} threshold > \sum_{d}^{\mathcal{D}} link(c, d, \mathcal{L}) \tag{2.2}$$

where: $link(c, d, \mathcal{L})$: c:s dependencies in direction d from the set of all links $\mathcal{L}$.
c : the component drawn from the set of components.
d : the directionality drawn from the set of directionalities.
$\mathcal{L}$: all dependencies concerning the component.

With formula (2.2) that measured the number of dependencies, which then will be compared to the previously defined threshold, which flags for the architectural smell.

2.3.3 Unused Interface

An interface is a class-public method. So, the unused interface smell is detected when a class with at least one public method but no other entity that are dependent on that class by using

one of its public methods. This smell is in violation of the incremental development process according to Flower and Scott [23]. Moreover, the presence of unused code may induce a degree of unnecessary complexity to the code base. [10] To formally measure this smell, the number of entities with public methods but no links pointing to it are counted. This is described in equation (2.3):

$$\sum_{c}^{\mathcal{C}} \sum_{e}^{c(\mathcal{E})} getNumInterfaces(e) > 0 \wedge getNumLinks(e) = 0 \tag{2.3}$$

where: $getNumIntefaces(e)$: the number of interfaces for the entity e,
$getNumLinks(e)$: the number of links ending in the entity e,
$c(\mathcal{E})$: the set of entities in component c,
c : the component drawn from the set of components $\mathcal{C}$,
e : the entity drawn from the set of entities $\mathcal{E}$.

2.3.4 Sloppy Delegation

Sloppy delegation occurs when a component delegates task it should preform by itself, e.g. a component responsible for monitoring stock prices and sell stock but then delegates the task to buy stocks to another component. [10]

The procedure to measure sloppy delegation is done by the formula below:

$$\sum_{c}^{\mathcal{C}} \sum_{e}^{\mathcal{E}} \sum_{l}^{\mathcal{L}} c \neq component(end(l)) \wedge getLinks(end(l), out) = 0 \wedge getLinks(end(l), in) < threshold \tag{2.4}$$

where: $component(e)$: the component that the entity e belongs to.
$end(l)$: the end entity of link l.
$getLinks(e, d)$: gets the number of links from entity e in direction d.
$threshold$: predefined threshold to indicate when sloppy delegations occurs, see 2.4,
$\mathcal{C}, \mathcal{E}, \mathcal{L}, c, e, l$: the respective set and its corresponding index.

2.3.5 Concern Overload

Concern overload occurs when a component have too many responsibilities (or concerns) [10] for example a component in a bank software that are responsible for both money transfers and sing in. Instead this should be handled by two different components. Concern overload violates the traditional SOLID principle, a single responsibility for each component [24]. The name overload is derived from the way a single component are getting intertwined in too many concerns, thus overloading the component which would benefit from being split up into one component for each concern [10].

The measurement of concern overload is computed by the two formulas below:

$$\zeta_c = \sum_{c}^{\mathcal{C}} \sum_{\omega}^{\Omega} P(\omega|c) > threshold_P \tag{2.5}$$

$$\zeta_c > threshold_{CO} \tag{2.6}$$

where: $P(\omega|c)$: the probability for concern ω given component c,
$threshold_p$: predefined threshold to indicate when it is probable that ω exists given c,
$threshold_{CO}$: predefined threshold to indicate when concern overload occurs, see 2.4
ζ_c : the concerns counted in component c,
$\mathcal{C}, c$: the component set and its corresponding index,
Ω, ω : the concern set and its corresponding individual concern,

To summaries, all these architectural smells are key components in the work to classify the the overall architectural degeneration of the system. On its own a smell does not guarantee architectural degeneration but a higher number of different smells found indicates a higher risk that architectural degeneration has occurred. [10]

2.4 Threshold

To understand how the threshold is used for determining if a smell is present or not, one first needs to understand the statistical term quantile. The term quantile is derived from the procedure of dividing a probability distribution into discrete parts [25], [26]. Within the scope of this thesis, four quantiles have been used, as proposed by Garcia *et al.*[27]. The procedure to calculate a quantile is illustrated by equation (2.7) and figure 2.2.

$$Q_i = \alpha\sigma \tag{2.7}$$

where: Q_1 : is found at $\alpha = -0.6745$ which is at the 25% mark,
Q_3 : is found at $\alpha = +0.6745$ which is at the 75% mark.

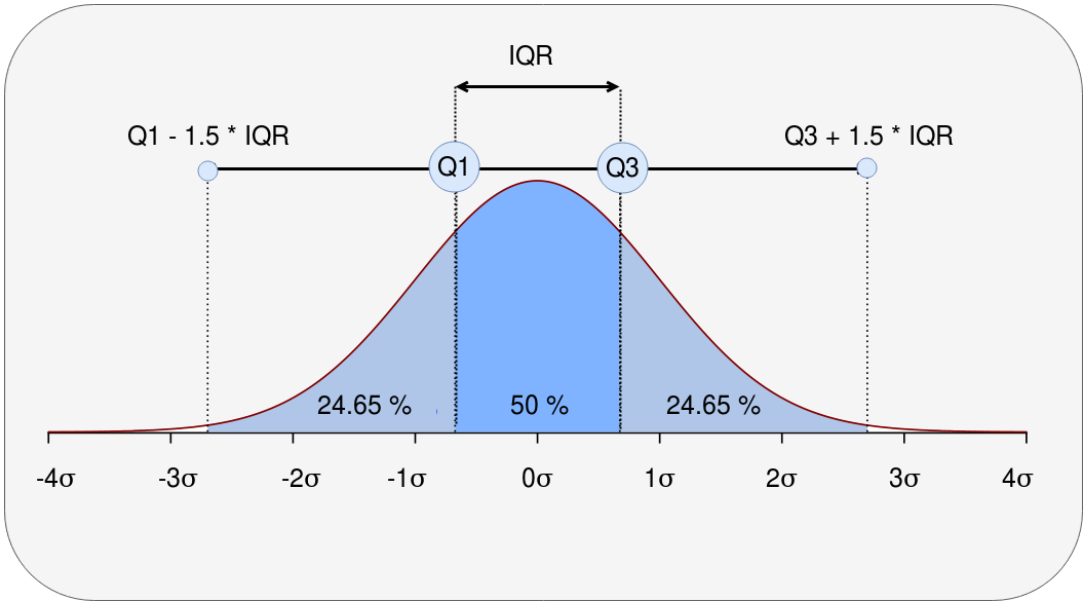

Figure 2.2: An illustration of how the different components in the threshold calculation is defined.

To calculate the threshold used by Garcia *et al.* the formula below (equation (2.8)) is used.

$$t = IQR \cdot 1.5 + Q_3 \tag{2.8}$$

where: t : the threshold to be defined,
IQR : the inter-quantile range,
Q_3 : the third quantile.

This method of calculating a threshold is used in architectural smell detection to detect if a value is a statistical outlier [10].

2.5 Architecture Recovery

To handle architectural degeneration developers need to be able to measure how architectures evolve, thus they need to be able to determine what the implemented architecture looks like at a certain moment in time [27]. This is done through architectural recovery. There are many techniques to recover an architecture that do not need access to the architects themselves [27]. Since access to architects is not always possible and is always expensive [28]. Garcia *et al.* compares six different recovering techniques in "A comparative analysis of software architecture recovery techniques"[29] the techniques tested are:

- scaLable InforMation BOttleneck (LIMBO),
- Bunch,
- Algorithm for Comprehension-Driven Clustering (ACDC),
- Weighted Combined Algorithm (WCA),
- Architecture Recovery using Concerns (ARC),
- Zone-Based Recovery (ZBR)

of which the best techniques were ARC and ACDC [27]. Of these two ARC was chosen for this thesis, this choice is discussed in section 6.1.1. To determine how good a architecture recovery technique was Garcia *et al.* used the MoJoFM method which measures distances between architectures [27]. Using ARC Garcia *et al.* achieved an average accuracy of 58.76% when validating ARC on eight different large scale software systems [27].

2.5.1 Architecture Recovery using Concerns

The architecture recovery using concerns (ARC) recovery method is first brought up by Garcia *et al.* in "Enhancing architectural recovery using concerns"[29]. It is then used in multiple studies [10], [27], [30] to recover architectures. ARC enhances structural recovery by also using text mining data when clustering the code around different concerns, where concerns translates to areas of responsibilities in the source code. An example could be that in a server application, database management is a concern and encryption another. The following steps in ARC are used to recover the architecture, they are also visualised in figure 2.3:

- Concern extraction

- Structural information extraction
- Brick recovery
- Concern meta classification
- Component/Connector classification

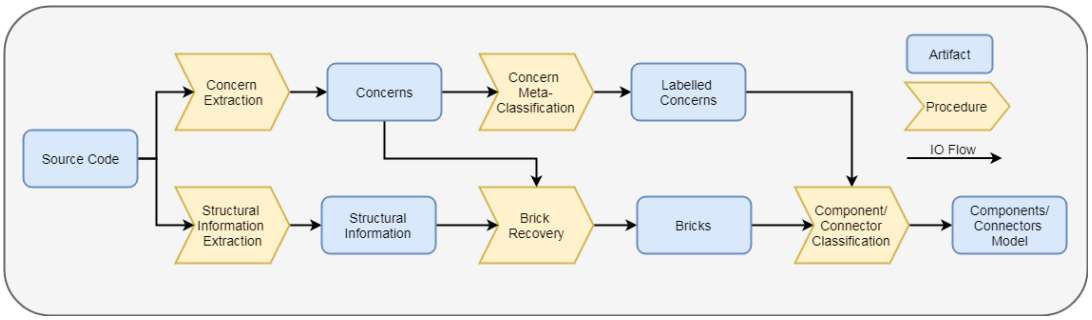

Figure 2.3: Overall view of the ARC recovery method.

Concern extraction and Structural information extraction are both done on the source code. To extract concerns, a statistical language model, Latent Dirichlet Allocation (LDA) (see section 2.6.1) are used. By representing the software system as a corpus, a set of documents which in turn each document is a set of words that occur in a software entity (a function or a class). These words are extracted from the comments and identifiers within the entities. With this data in form of the corpus, a statistical model is formed over a predetermined number of topics, that consists of a set of words and a probability for each word to appear within that topic. These topics are called concerns in the context of architecture recovery [29].

When extracting structural information, the source code is parsed to find all the software entities and their connections in the form of dependencies between each other and external modules and components. Many other structural features can be extracted from the source code e.g. file names, directory paths, LOC and time of the last update. It is important to carefully choose what information to extract and use for recreating an architecture. [31]

In ARC a brick is defined as a set of entities and brick recovery clusters the software entities together according to both the structural information and concerns. This creates bricks that are not only structured after dependencies but also what the purpose of the code is. For example, a class for logging and a function for exporting to excel might both be dependent on a read file function, but that does not mean that they should be put in the same cluster. To be able to take both structural information and concerns into account a similarity measure that measures probability distributions for concerns and combine it with a similarity measure for boolean values used to represent dependencies is done [29]. Read more about the combined similarity measurement in section 2.8.

To be able to automatically determine if the bricks recovered through brick recovery are components or connectors, the concerns have to be classified as application-specific or application-independent. This is done through concern meta-classification that uses supervised learning to create a classifier that can classify the concerns as application-specific or application-independent. This classification is done using the k-nearest neighbour algorithm on the words of each concern; each concern is classified as application-independent if its words are more common in other application-independent concerns [29].

With the labelled concerns, usage of known connector libraries such as a socket or datagram library and known connector design patterns such as Proxy, Mediator, Chain of Responsibility, Adaptor/Wrapper and Observer it is possible to use supervised learning to classify the bricks. The result is a set of components and connectors that together make up the recovered architecture [29].

2.6 Text Mining

In this section we explain more in depth about the text mining used in the thesis. In our thesis, text mining is used to extract concerns used by ARC to recover the architecture from a soft

2.6.1 Latent Dirichlware system.et Allocation

To be able to discuss Latent Dirichlet Allocation (LDA) some notation and terminology needs to be defined. The following words are of special meaning in the context of text classification and LDA:

- Word – The atomic building block of text in the case of text classification and is defined as any word contained in the pre-defined vocabulary $\mathcal{V}$ [32].

$$\mathcal{W} = (c_1, c_2, \ldots, c_n) \tag{2.9}$$

where: $\mathcal{W}$: The word in question, where $\mathcal{W} \in \mathcal{V}$.
c_i : The i:th character in the word.

- Document – A document is a sequence of N words, which in text is one or more sentences. The formal definition of a document is [32]:

$$\mathcal{D} = (w_1, w_2, \ldots, w_n) \tag{2.10}$$

where: $\mathcal{D}$: The document.
w_i : The i:th word in the document.

- Corpus – A collation of documents, it is the dataset top-level description and is all the text which classification is performed on. The definition of a corpus is similar to the previously defined word and document [32]:

$$\mathcal{C} = (d_1, d_2, \ldots, d_n) \tag{2.11}$$

where: $\mathcal{C}$: The corpus.
d_i : The i:th document in the corpus.

- Topic – A topic is an abstract description about characteristics of a document. The number of topics is constant to a pre-defined value.

Continuing with the notation described above, LDA is a generative probability statistical model of the a corpus [32]. The aim for an LDA model is to assign each document in the corpus to a set of topics coupled with the probability that the document belongs in that topic. This is done given the probability of different words occurring in the document [32].

The procedure of assigning topics to documents is well described by Blei *et al.* who formulates the following steps:

1. Choose the N according to equation 2.12:

$$N \sim Poisson(\xi) \tag{2.12}$$

2. Choose the θ according to equation 2.13:

$$\theta \sim Dirichlet(\alpha) \tag{2.13}$$

3. Calculate for each word:

   i) Choose a topic z_i from the set of topics (a multinomial distribution) according to equation 2.14:

$$z_i \sim \mathcal{M}(\theta) \tag{2.14}$$

   ii) Choose a word based on equation 2.15:

$$P(w_n|z_n, \beta) \tag{2.15}$$

For a more extensive explanation of LDA read the article "Latent dirichlet allocation" by Blei *et al.* [32].

2.6.2 $\mathcal{N}$-gram model

The $\mathcal{N}$-gram model represents a series of $\mathcal{N}$ continuous words from a text or a speech. The model strives to capture the conditional probability given by the $\mathcal{N}-1$ previous words, thus making the meaning of verbs more important to the model [33]. The $\mathcal{N}$-gram model can also be seen as a $(\mathcal{N}-1)$-order Markov model, thus a probabilistic language model for text prediction. The $\mathcal{N}$-gram can be implemented with $\mathcal{N}$ = 1,2, ..., n; where the classical n is unigram (bag of words), bigram and trigrams, where the different $\mathcal{N}$-gram models are implemented with the formula below.

$$P(\boldsymbol{w}_1^{\mathcal{N}}) = \prod_{k=1}^{\mathcal{N}} P(w_k|\boldsymbol{w}_{k-\mathcal{N}+1}^{k-1}). \tag{2.16}$$

where: $\boldsymbol{w}_1^{\mathcal{N}}$: the vector of words $(w_1, \ldots, w_{\mathcal{N}})$,
$\mathcal{N}$: the order of the model,
w_k : the word at index k.

2.6.2.1 Bag-of-Words

The first n of the $\mathcal{N}$-gram model is the Unigram model, also known by the name Bag of Words (BoW). The key idea behind the representation format bag of words is obtained by studying the name itself, it iterates through the input text and adds each unique word (atomic unit can be a character also) to a dictionary with the word as the key and a counter as the value. Thus, resulting in a vector of dictionaries for the count of each unique word. [34].

2.6.3 Term Frequency-Inverse Document Frequency

The term frequency-inverse document frequency or tf-idf is one of the most prominent models for text mining [35]. Tf-idf captures the importance of a unique word in a set of documents, the tf-idf is defined by two sub-formulas, the term frequency and the inverse document frequency, both further explained in equation 2.17:

$$tf(t, D) = \frac{1}{2} + \frac{f_{t,d}}{2 \cdot \max\{f_{t',d} : t' \in d\}} \tag{2.17}$$

where: $tf_{t,d}$: the term frequency,
$f_{t,d}$: the raw frequency,
t : the count of the word,
t' : the most frequent word,
d : the document.

Continuing to the inverse document frequency equation shown in equation 2.18:

$$idf(t, D) = log(\frac{N}{1 + |d \in D : t \in d|}) \tag{2.18}$$

where: idf : the inverse document frequency,
t : the count of the word,
N : the number of documents in D,
d : the document,
D : the documents.

The combination of the two formulas above defines the equation for the tf-idf, which is presented in equation 2.19:

$$tfidf(t, d, D) = tf(t, d) \cdot idf(t, D) \tag{2.19}$$

One of the essential reasons for using the tf-idf is to make words frequently occurring in a limited amount of documents not be over-represented by the model [35].

2.6.4 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is an alternative method to LDA. SGD is an optimisation algorithm that in combination with neural networks can be applied to text classification problems, where the data are plentiful as in the text mined source code of the projects [36]. The analytical formula behind SGD is explained in equation (2.20), and for the interested reader the background around SGD, Bottou provides a more verbose explanation in "Large-scale machine learning with stochastic gradient descent" [36].

$$w_{t+1} = w_t - \gamma_t \Delta_w Q(z_i, w_t) \tag{2.20}$$

where: w : weight vector,
γ : a pre-defined gain,
$Q(z, w)$: minimized loss function,
z : randomly picked example.

2.6.5 Logistic Regression

A second alternative to LDA is Logistic Regression (LR) and it appears in several forms; binary, ordinal and multinomial, in the scope of this thesis LR will refer to the multinomial logistic regression. Where the multinomial extends the binary LR to a multi-class problem. In the multinomial case, the output vector has more than two states, in the scope of this thesis, the number of states should be equal to the number of concerns. The mathematical formula for LR is described by equation (2.21) and the for the curious reader a more extensive explanation of LR is made by Krishnapuram *et al.* in "Sparse multinomial logistic regression: Fast algorithms and generalization bounds" [37].

$$P(y^{(i)} = 1|\boldsymbol{x}, \boldsymbol{w}) = \frac{\boldsymbol{x}e^{\boldsymbol{w}^{(i)^T}}}{\sum_{j=1}^{m} \boldsymbol{x}e^{\boldsymbol{w}^{(j)^T}}}. \tag{2.21}$$

where: $\boldsymbol{w}^{(i)}$: weight vector corresponding to class i,
$y^{(i)} = 1$: true prediction corresponding to class i,
$y^{(i)} = 0$: false prediction corresponding to class i,
$\boldsymbol{x}$: the input vector,
^{T} : the superscript^T is the transpose.

2.7 Clustering

In this section, techniques for clustering are brought up, which relates to this thesis. Clustering is used in our work to combine entities into components when recovering an architecture using ARC.

2.7.1 Agglomerative Clustering

Agglomerative clustering can be implemented in many different ways. A high-level basic algorithm is illustrated in figure 2.4 and also described by the following steps:

1. Calculate the distances between all objects.
2. Merge the two closest objects into a cluster.
3. Recalculate the distance between the new cluster and all other objects.
4. If we have not reached the desired number of clusters go to step 2.

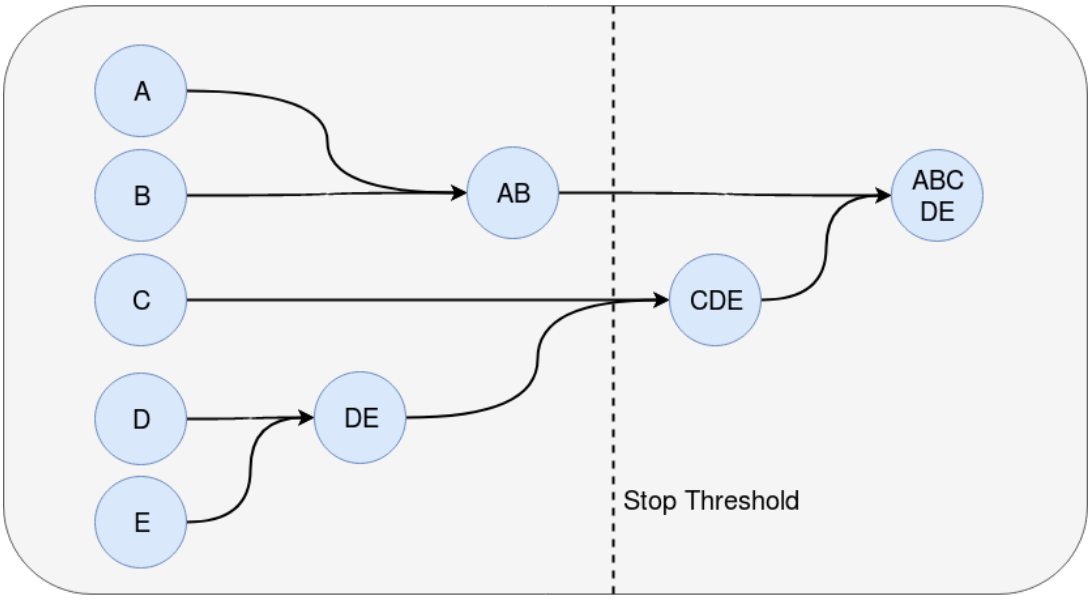

Figure 2.4: An illustration of how agglomerative clustering process works.

This is not a fast algorithm, step 1 has a time complexity of $\mathcal{O}(n^2)$. When calculating step 3 there are multiple methods to choose from. Three common ones are single, complete and average link distance methods. Single link distance takes the closest neighbours distance of the cluster to every other object. The opposite is done in complete link distance there the longest distance from the newly formed cluster is chosen as the distance to every other object. In average link distance, on the other hand, the average distance is calculated for all the objects within the cluster to every other object. [38]

2.7.2 K-Medoids

K-Medoids also known as Partitioning-Around-Medoids (PAM), is a variation of the more known approach K-Means. K-Medoids randomly selects a set of medoids among the points to cluster were each medoid represent a cluster. It then partition the rest of the points in the cluster that each point is closest to. The distance between a point and a cluster is the distance between the point and the medoid representing the cluster. When all points are partitioned into clusters new medoids are chosen. This is done by calculating which point is in the "centre" of the cluster, that point is chosen at the new medoid. This is repeated until a maximum number of iterations are reached or the algorithm converges. [39] A more formal algorithm is shown in algorithm 1.

2.8 Distance Measurements

In this section, two important distance measurements, Jaccard and Jensen-Shannon distance will be presented and how these distances can be combined.

2.8.1 Jaccard Distance

Jaccard distance is a classical and well-adopted way of measuring the binary asymmetrical distance between two vectors, Jaccard distance is defined by equation (2.22) [31]. According to Maqbool and Babri the Jaccard similarity is one of the best measurement to use when working with structural information extracted from source code. Moreover, Maqbool and

Algorithm 1 $\mathcal{K}$–Medoids

function $\mathcal{K}$–MEDOIDS($\mathcal{K}$, $\mathcal{N}$) **return** $\mathcal{K}$–clusters
  input: $\mathcal{K}, \mathcal{N}$ is the number of Cluster to create and number of data points.
  persistent variables: TC, the total cost for the cluster.
  $'$ denotes the variables previous iterations values.

  if $first$ **then**
    randomly select $\mathcal{K}$ data points as medoids
    assign all $\mathcal{N} - \mathcal{K}$ none-medoids to the closest medoid
    calculate the TC for each cluster
  while $TC < TC'$ **do**
    swap each medoid $\mathcal{M}$ and none-medoid $\mathcal{O}$
    set TC to TC' calculate the new TC
  return $\mathcal{K}$–clusters

Babri show why the binary asymmetrical version of the distance should be used [31].

$$\mathcal{J}_{distance}(v,u) = \frac{tn + fn}{tp + tn + fn} \tag{2.22}$$

where: u, v : the two input vectors,
tn : true negative,
tp : true positive,
fn : false negative,

2.8.2 Jensen-Shannon Distance

The Jensen-Shannon is a distance measurement for probability distributions based on the Jensen-Shannon divergence, which is the terms in the square root expression in equation (2.23) [40]. The Jensen-Shannon divergence is also known as the information radius or the total divergence by the average. [40]

$$\mathcal{JS}_{distance}(u,v) = \sqrt{\frac{\mathcal{D}_{KL}(u||m) + \mathcal{D}_{KL}(v||m)}{2}} \tag{2.23}$$

where: $\mathcal{D}_{KL}$: the Kullback-Leibler divergence [40],
u, v : probability distributions,
m : the pointwise mean of u and v.

2.8.3 Combined Distance

The process of combining distance metrics was proposed by Garcia *et al.*[29]. One way to aggregate two mathematical distance metrics is by using addition. Addition between two different distance metrics results in a new distance metric which will keep the properties of a mathematical distance metric, thus obeying these four rules:

$$d : \mathcal{M} \times \mathcal{M} \to \mathbb{R} \tag{2.24}$$

where: d : any distance function that satisfies these four rules,

1. Non-negativity : thus, $\forall x, y \in \mathcal{M} : d(x, y) \geqslant 0$,
2. Identity : $\forall x, y \in \mathcal{M} : d(x, y) = 0 \iff x = y$,
3. Symmetry : $\forall x, y \in \mathcal{M} : d(x, y) = d(y, x)$.
4. Triangle inequality : $\forall x, y, z \in \mathcal{M} : d(x, y) \leqslant d(x, z) + d(z, y)$ [41].

By using the criteria above in equation (2.24) the resulting metric-space is given by: $(\mathcal{M}, d)$.

2.9 Multi-Paradigm Programming Languages

The area of multi-paradigm programming languages, lives in great synergy with modern agile development processes. Providing the developers the freedom to chose specific language construct and paradigms to implement the proposed system architecture. Multi in this context represent the following four paradigms:

- Imperative – Imperative is a synonym for state-driven programming, which utilise the execution of certain state both to pass information and to store internal information. [42]
- Functional – The pure mathematical programming paradigm, where the unique language construct is a λ–calculus, which consists of three distinct operations to handle functions. The three functions are; α–conversion, β–reduction and η–conversion which is explained further by Michaelson. [43]
- Object-Orientated – The paradigm prevailing in the industry, which has three primary requirements; hiding data with encapsulation of data in classes or objects, inheritance by construction of new objects with the base from previously defined objects, polymorphism providing abstract interfaces implemented by multiple base classes. [44]
- Procedural – The line by line approach, where the key concept is function calls and different types of iterations. [45]

Even with four distinct paradigms, their differences does not make them mutually exclusive, in fact, they work excellent together in different combinations, to harness the full capabilities of a multi-paradigm programming language.

2.9.1 Python

Python was created by Guido van Rossum and first released in 1991 [46]. It is an interpreted, interactive and object oriented programming language. It features modules, exception handling, dynamic typing, high level data types and classes. It can be extended with C or C++ and runs on multiple operating systems including Windows Mac OS and most UNIX variants [47]. Python supports the programming paradigms object-oriented, functional, imperative and procedural programming and is therefore considered a multi-paradigm programming language [48].

There are multiple implementations of the Python language, some examples are Jython, PyPy, Python for .NET and CPython [49]. Of these implementations, CPython is the original and most-maintained one, written in C [49]. CPython is the implementation used for this thesis.

2.9.2 Abstract Syntax Tree

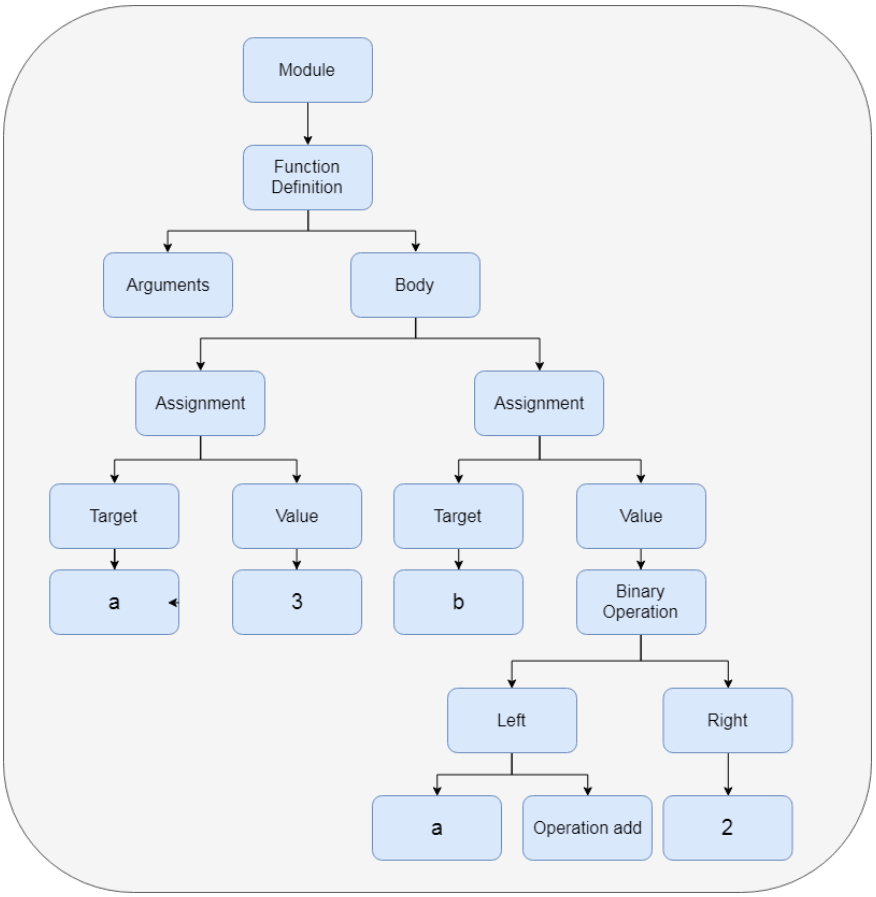

Figure 2.5: AST representation of a simple function with assignment and addition.

Abstract Syntax Trees (ASTs) are created from source code. They contain the semantics of the code as a tree of expressions built up by tokens. ASTs are used when compiling, interpreting or analysing source code from many different languages [50]. A basic example is a Python function with some assignment and addition that can be seen in the code listing 2.1. This code results in a AST as shown in figure 2.5.

```
1 def foo():
2   a = 3
3   b = a + 2
```

Code Listing 2.1: Simple function with assignment and addition

In Python the AST is built up of different nodes such as function definitions, assignments and binary operations as seen in figure 2.5. There are many more types of nodes and each node also contains meta data such as data types and commands which are a bit simplified in figure 2.5. The AST in Python is created and navigated through by using the ast module that comes with Python[1].

[1] https://docs.python.org/3.7/library/ast.html

As mentioned ASTs can be used for many things such as analysing and even modifying code. Baxter *et al.* uses it for detecting clones within source code and discuss how that could extend that work by automatically edit code when this is detected to increase maintainability [51].

3 Related Work

This section aims to illustrate key concepts to the reader, to facilitate insight in the conducted thesis. Related work will be presented to give a context frame to work within. We will start by providing some context to architectural degeneration and continue by examining the related work within architectural recovery and detecting architectural smells.

3.1 Architectural Degeneration

That software is ever changing and that change not only brings good things such as new features but also adds complexity making it harder to understand and change. This has been known for a long time as brought up by Lindvall *et al.* [9]. In their conference paper "Avoiding architectural degeneration: an evaluation process for software architecture" they report on a case study where they like us reconstructed a architecture to measure degeneration. Unlike us they had a maintainability focus and re-engineered the system they were investigating. Using coupling metrics they showed that the new architecture they created was better than the old one. The new architecture had a lower coupling between modules than the old architecture and was therefore considered better. Lindvall *et al.* also concludes that the actual architecture implemented differed a little from the design that their evaluations had shown well structured and maintainable. This differs from the original design poses a threat to the maintainability of the system. To conclude how big this threat is further studies of the systems maintainability has to be conducted.

In the conference paper "Diagnosing architectural degeneration" by Hochstein and Lindvall [5] they present an overview of the diagnosing process and the tools used like the tool we have developed. The process they present consists of the following steps:

1. Select perspective for evaluation.
2. Define/select guidelines and establish metrics to be used in the evaluation.
3. Analyse the planned architectural design in order to define architectural design goals.
4. Analyse the source code in order to reverse engineer the actual architectural design.

5. Compare the actual design to the planned design in order to identify deviations.
6. Formulate change recommendations in order to align actual and planned designs.

This process is quite similar to how we have made our analysis of different projects. We have not used an architectural design and do not give recommendations since our focus have been on the evolution of software systems.

Hochstein and Lindvall present tools for recovering the actual architecture and sort them into filtering and clustering, architecture recognition and design pattern identification tools. Our tool would probably fall into the clustering category. Other tools they bring up are pattern-lint tools that when identified the original architecture can compare the implemented architecture with the planned architecture. The paper concludes that architectural degeneration is a problem and references some examples, in one of the examples Mozilla had to rewrite their entire web browser. Diagnosing this problem Hochstein and Lindvall concludes, cannot be fully automated, but assisted by the tools they have presented. The tools are not complete and to make a full recovery, interviews with the original architects are required. This is something we agree with because during our investigation we have discovered that manual work and supervision is needed for the methods implemented by these tools to be as effective as possible.

In "A comparative analysis of software architecture recovery techniques" Garcia *et al.* compares six different architecture recovery techniques on eight different open source software systems. Unlike us they look at systems that are all written in Java, C or C/C++ but like us they have a wide variety in the size of the projects, they look at projects in the size of 180K - 4M LOC. They conclude that no methods are perfect but two of the six stand out Algorithm for Comprehension-Driven Clustering (ACDC) and Architecture Recovery using Concerns (ARC) which is the one we are using for our study. [27]

Tornhill writes in the conference paper "Prioritize technical debt in large-scale systems using codescene" about his own software CodeScene and how it can be used to identify and prioritise technical debt. CodeScene visualises a multiple static analysis and machine learning algorithms that are continuously run on source code. It uses both a technical and a social aspect in the analysis. [52]

Furthermore, Tornhill builds upon the previous mentioned paper in "Assessing Technical Debt in Automated Tests with CodeScene". In which the author presents an interesting take on technical debt, where a machine learning algorithm is applied to the project and presents where possible technical debt has developed. The predictions are based on the metrics, cyclomatic complexity, lines of code, and commented lines of code, to derive areas in new of refactoring to uphold a high-quality software. By the use of machine learning for classification of possible high technical debt risk areas, the need for manual code inspections decreases. Thus, freeing up time for actual refactoring. [53]

Unlike us Tornhill primarily look at code defects and commit history but it shows what the research into architectural degeneration and the methods and tools to detect it could lead to in form of commercial products.

Nugroho *et al.*[54] measures like us how architectures worsen over time. They have a more economic focus, measuring architectural debt by estimating the maintainability of parts of the code. Where low maintainability on large parts of the code base results in a large technical debt [54]. For measuring maintainability Nugroho *et al.* refer to "A practical model for measuring maintainability"[55]. Nugroho *et al.* concludes that measuring maintainability and

the effort of restoring maintainability to its optimal value is a good method for estimating technical debt they also conclude that giving architectural degeneration a more economical perspective is important to explain to non-technical decision makers and that is were architectural debt is very useful [54]. The use of the debt metaphor is also brought up as useful for explaining architectural degeneration is brought up by other authors [56], [57].

3.2 Architectural Recovery

When reconstructing an architecture from source code some form of clustering is often used to group entities into components. In the article "Hierarchical Clustering for Software Architecture Recovery" Maqbool and Babri investigates different hierarchical clustering methods. They conclude that only because an algorithm or a certain similarity measurement works well overall do not guarantee that it will work all the time. Different software systems are more suitable for recovery using certain algorithms and similarity measurements. This fact and as mentioned by Pollet *et al.* that clustering is a quasi-automatic recovery technique show that human interaction with recovery tools are important. We have incorporated this in our tool making it possible to tweak the different steps in the recovery process.

Even though you cannot know exactly what methods will work best for recovering the architecture of a given system, some algorithms and similarity measurements are often outperforming the rest. Maqbool and Babri brings up hierarchical clustering as superior to partitioning clustering because it is often faster, do not require the number of clusters to be predetermined and fits better with how software systems often are constructed, as components that in turn consists of smaller components. As for similarity measurements, the ones within the Jaccard family as the one we have used to measure the distance between bitmaps proved superior. [31]

Seriai *et al.* were pioneers in the area of extracting lexical data using text mining. The mined data was from the source code in OO-languages applications. Then combining it with the structural data extracted. They trailed both the data separately and then in a combined measurement and concluded that the combined measurement was superior to the individual measurements [59]. Similar to our study where we apply text mining to extract and concern related data that we combine with structural data. With the big difference that we applied our methodology to the run time evaluated multi-paradigm language Python.

In the article "Arcan: a tool for architectural smells detection" Fontana *et al.* describes their study in which they also made a tool for discovering architectural smells. Their tool called ARCAN was designed to detect the smells Cycle Dependency, Unstable Dependency and Hub-Like Dependency in systems written in Java, unlike us who are looking at systems written in Python. Both our and ARCAN detect Cycle Dependencies but unlike us Fontana *et al.* choose to only focus on Dependency smells and have chosen smells that are focused on Java since Unstable Dependency is a smell detected on packages and Hub-Like Dependency is detected on classes. Like us Fontana *et al.* have earlier had some issues with validation but in "Arcan: a tool for architectural smells detection" they use experts within the two projects DICER and Tower4Clouds to validate their detected smells. They observed a 100% precision since all architectural smells detected were verified, but the experts knew about multiple architectural smells that were not detected resulting in false negatives.[60]

4 Method

The method described in this section aims to generate results that will give insights regarding the research question.

> *"How does architectural degeneration evolve in large scale systems written in the dynamically typed multi-paradigm and interpreted language Python?"*

Architectural degeneration was measured at multiple points in time accessing the different versions of the systems through the version control system as done in similar studies [10], [8]. The measurements consisted of two parts, recovering the architecture and using the recovered architecture to measure the architectural smells presented in section 2.3. By analysing the data from these measurements the results for how the architectural degeneration has evolved were produced.

4.1 Developing PySmell

To create a tool (PySmell) that would be as general as possible but without risking not finishing a tool that could be used for the research within the given time parameters the chose to work in iterations was opted for. By working in three iterations of four weeks the goal was to have a tool and some form of result at the end of each iteration. The use of an iterative development process made it possible to iteratively evaluate and improve the results, this to achieve a result of as good quality as possible, given the time frame.

4.2 Architecture Recovery

The performed architectural recovery was done by following the ARC (described in section 2.5.1) method. The goal was to extract and partition the entities into components that smell detection could be applied to. This was done by implementing the three first steps of ARC, structural extraction, concern extraction and brick recovery in PySmell.

4.2.1 Structural Information Extraction

Entities and symbolic dependencies were extracted from the source code, modelling it as a graph. In the graph, vertices represented entities and edges represented symbolic dependencies between these entities.

To extract the entities and symbolic dependencies the abstract syntax tree (AST) library that comes with Python was used. Using the AST the code was parsed for function, method and class definitions adding them to a database of entities, tying the methods to their class. When a database of entities was created, the source code was parsed a second time for all the call expressions. For each call, a look-up is performed in the database and if it was a call to one of the known entities a dependency entry was created in the database.

The method for extracting dependencies was not able to catch all dependencies, for example if a function was passed on as a parameter and then renamed as showed in 4.1 it was not possible to find the entity called in the database. There was also issues with entities used as decorators.

```
1 def foo():
2     print("foo")
3
4 def bar(func):
5     func()
6
7 if __name__ == "__main__":
8     bar(foo)
```

Code Listing 4.1: Function changing name when sent in as parameter example.

To avoid issues with calls matching multiple entities in the database the import statements were used. By parsing the import nodes within the AST, conclusions could be made which entities were imported. With this knowledge we could try to match found calls with those entities first, reducing the number of calls matching multiple entities in the database. This did not eliminate duplicate matches but reduced it significantly.

Dependencies between internal components are not the only structural information that was extracted. Entities can also be dependent on external modules. Therefore if a call was found that was not to an internal entity the call was matched with imported modules. If it matched any of the imported modules it was stored as an external dependency.

4.2.2 Concern Extraction

Extracting concerns was done by using LDA (see section 2.6.1). A document containing words from comments and identifiers (such as function and variable names) was created for each entity. This process resulted in a list of concerns within the project and each entity being given a probability distribution over concerns and how likely they belong to them.

The procedure that was applied to the extracted sequences of text was to populate a Pandas[1] dataframe. Pandas is an open-source library for natural language processing and the dataframe is the preferred data structure to work within Pandas in most machine learning projects. With all the data conveniently available in a Pandas dataframe the pre-processing followed, where removing of extremes[2], stop-words, Python-words[3]. Furthermore, the initial

[1] https://pandas.pydata.org/
[2] Words which appear with a frequency of 50% or less then two times.
[3] Classical Python words: self, super, list, int, str, float, tuple, dict, class, def.

model of the vocabulary was created from a bag-of-words model. The bag-of-words model was then transformed to the tf-idf (see section 2.6.3) representation of the data, reason for the decision to use the tf-idf is made clear by Vijayarani *et al.* [35]. The pre-processing is performed to limit the size of the vocabulary, thus yielding better results.

With the pre-processing steppes completed the LDA model could be trained on the tf-idf data representation and then was used to extract concerns (known as topics in the literature) given a set of words. With the knowledge of what concern being most probable given a set of words, all entities were assigned a concern, given their set of words, thus making entities with similar linguistics properties assigned to the same concern. The term linguistic properties translate to a similar set of words and sentence structure. Furthermore, all entities were assigned the probability distribution corresponding to their assigned concern.

4.2.3 Brick Recovery

With information about both dependencies and concerns, the entities are clustered together with the clustering technique Agglomerative clustering, to create bricks. During the clustering, the dependencies on both internal entities and external modules were represented as Boolean-vectors (bitmaps) and concerns as a probability distribution vector. This resulted in a set of bricks where each brick consists of a set of entities that belong together.

Agglomerative clustering uses a distance measure between each data point and merges the two closest points to create a cluster. To create this distance measure between two entities Jensen-Shannon distance and Jaccard distance was combined. Jensen-Shannon distance was used to represent the distance in regards to concerns since it measures the distance between probability distributions. Jaccard distance was used to represent the distance between two entities in regards to dependencies. The combined distance measures are mathematically described as equation 4.1.

$$Distance(e_i, e_j) = \mathcal{EM}(e_i, e_j) + \mathcal{C}(e_i, e_j) + \mathcal{D}(e_i, e_j) \tag{4.1}$$

where: e_i, e_j : the i:th and j:th entity,
$Distance$: the combined distance,
$\mathcal{C}$: the Jensen-Shannon distance between the entities concerns,
$\mathcal{EM}$: the Jaccard distance between the entities external modules,
$\mathcal{D}$: the Jaccard distance between the entities internal dependencies,

In Agglomerative clustering, the number of clusters to find needs to be predetermined. To find a good number, multiple clusterings were performed using different numbers of clusters. The number of clusters that gave a result with as few outliers without merging bigger clusters was considered the best.

When two data points or clusters are merged, the distance matrix used by Agglomerative clustering is updated. This can be done in multiple ways as described in section 2.7.1. For clustering entities into bricks average linkage was chosen.

4.3 Measuring Architectural Smells

Given the recovered architecture, the next step was to measure architectural smells metrics in it. In this section, the method for how that was performed is described.

4.3.1 Dependency Cycles

Counting the number of cycles in the system was done by modelling the architecture as a graph. Bricks were modelled as nodes and directed edges from node A to node B represented a dependency between A and B in which A were dependent on B. To determine if there is a dependency from node A to node B a dependency threshold was used. If there were more dependencies from node A to node B than this threshold an edge was created representing the dependency from node A to node B. The threshold was set manually as a parameter. After the graph had been created the cycles in the graph were counted to determine the number of dependency cycles.

4.3.2 Link Overload

The process presented in the theory chapter, section 2.3 and illustrated in equation (2.2) was implemented in PySmell. The implementation used the same graph that was used for counting dependency cycles to determine the amount of dependencies going in and out from each brick. A threshold was calculated using the statistical models described in section 2.4. This was used in the implementation to determine what number of links were required to detect the link overload smell.

4.3.3 Unused Interfaces/Entities

The number of unused entities were counted instead of the number of unused interfaces as described in section 2.3. Why this change was made is brought up in the discussion in section 6.1. To count the number of unused entities the database created during the structural extraction was queried for entities that no other entity were dependent on.

4.3.4 Sloppy Delegation

To detect sloppy delegations within the recovered architecture the process described in section 2.3 and illustrated in equation (2.4) were implemented in PySmell. The data was fetched from the database created by the structural extraction. The threshold that was used to determine if a smell was detected or not was implemented according to the description in section 2.4.

4.3.5 Concern Overload

Just as sloppy delegation and link overload, concern overload was calculated by a implemented version of the process described in section 2.3 and illustrated in equation (2.5) and (2.6). The implementation used data from both structural and concern extraction unlike the other smell detection implementations. As in sloppy delegation and link overload a threshold was used to determine if to detect a concern overload smell or not and it was also calculated as described in section 2.4 in the implementation.

4.4 Validation

Validating PySmell was done by using it on itself. Expert knowledge in how the architecture should look and experimentation was used to optimise parameter choices to recreate an architecture as close to the intended one as possible. To choose the number of concerns, a range from five to ten number of concerns was tested and evaluated from the top 10 most common words in each concern. The words were evaluated by using our PySmell expertise to decide if the collection of words belonged to one ore more concerns and if there were concerns within PySmell not represented by any found concern. Then a different number of bricks was tested using the chosen concerns to find one that created the best distribution of entities

and clusters that were considered most accurate. A good distribution was defined as a even distribution with as few small and big clusters as possible. For example a component with one or two entities is too small a component when the system is consists of 50-100 entities. On the other hand when one component holds between a third and a half of the entities in the entire system that is a too big component.

The architecture created by PySmell was evaluated by us as experts. The bricks were labelled according to what the majority of the entities intended to belonged to. The fact that two component could not have the same label and that the bricks had to correspond to the components that were a part of the intended architecture was taken into account. The percentage of entities within a brick that was determined to belong there was the accuracy of that brick. The total accuracy was determined by the amount of entities that were correctly placed out of the total amount of entities.

The evaluation of the smell detection was also performed on the tool itself. By going through each detected smell and classifying it as a true or false positive an accuracy percentage was calculated for each smell. No investigation into false negatives was done for any smell due to time limitations.

4.4.1 Parameters

During the architectural recovery process, there are three critical parameters with great impact on the results in the form of detected smells:

- Dependency threshold - A threshold from which the number of dependent entities within a brick with dependencies to another brick must be higher than to create a brick dependency between the two bricks.
- Bricks - The amount of clusters to assigned the entities to. This parameter is heavily dependent on the size of the project, which the architectural recovery is performed on.
- Concerns - The number of architectural concerns that should be found in the project in question.

The most prominent feature of a project when choosing the parameter was the size. That together with a manual inspection of the source code determined the number of concerns. After choosing the number of concerns the number of bricks was selected to minimise the number of bricks with very few or too many entities.

The dependency threshold was chosen based on the size of the bricks. By choosing a dependency threshold of around 5% of the average brick size but also looking at the smaller bricks making sure they were able to get dependencies.

4.5 Analysing the Architectural Degeneration

Analysing the architectural degeneration was done by applying PySmell at several different versions of the projects that were investigated, the process is illustrated in figure 4.1.

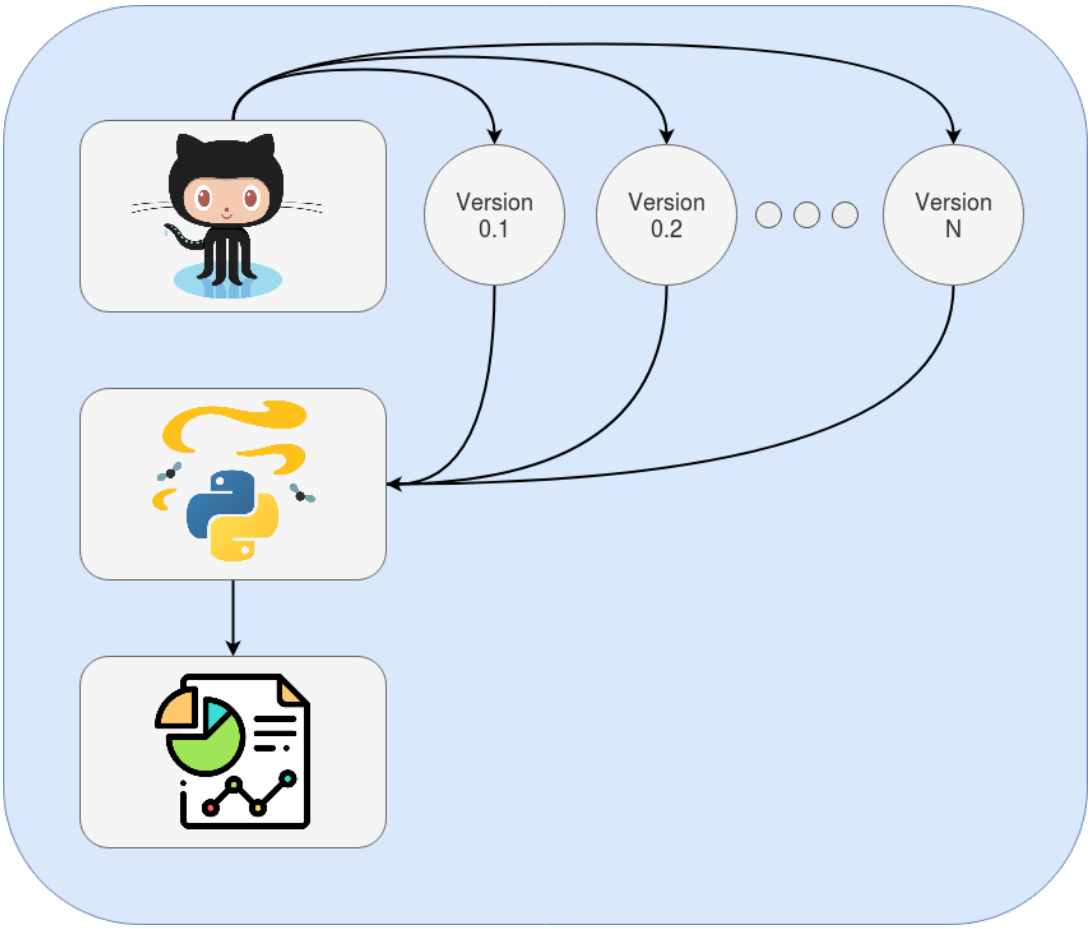

Figure 4.1: Illustration of the process of analysing how architectural degeneration evolves over time.

First, different versions of the selected project were downloaded from GitHub. For each version, PySmell was applied for detecting smells from the projects. The detected smells were then collected in a spreadsheet for analysis and presentation through graphs in the report in chapter 5.

5 Result

In this chapter, the results of the architectural recoveries and the measurement of the architectural smells will be illustrated as well as the resulting tool and the validation results.

5.1 PySmell

In this section, we describe the resulting tool that was developed to enable architectural recovery and smell detection on software systems developed using the Python programming language. In the list below the different components in PySmell will be described.

- Structural extraction component, responsible for traversing the source code and both detecting entities and assigns inter-entities dependence and store all this in a MySQL database.
- Concern extraction component, this component preprocess the data from the database and assigns each entity to a concern.
- Brick recovery component, by the use of both the structural and concerns related data, cluster the entities into bricks.
- Smell detection component, where the recovered architecture is tested for architectural smells.

5.2 Investigated Projects

To achieve measurements of architectural degeneration our method as described in chapter 4 was applied to the following projects:

- Django[1] - A open source web framework for Python that had its first release in 2012.

[1] https://github.com/django/django

- Flask[2] - A open source micro-framework written in Python and was released in 2004 as an April's fool, by then rapidly grew in popularity.
- PySmell - The tool itself was used for testing during development and also for validation since we had expert knowledge into how the architecture was structured and how the code worked.

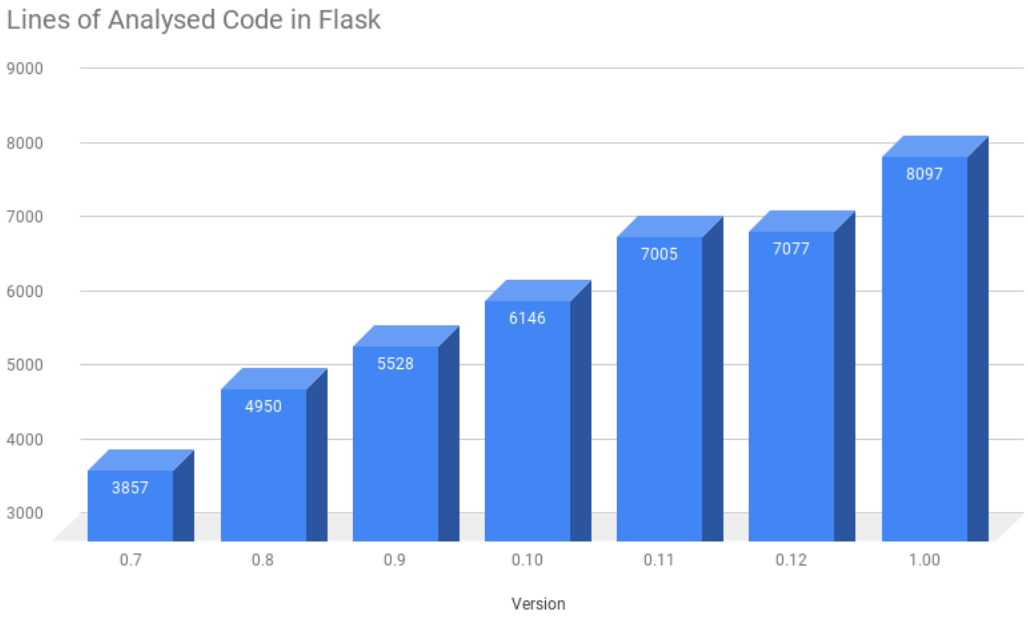

Figure 5.1: Illustration of how the analysed size of Flask evolves over time.

5.3 Architectural Smells

In this section, the found architectural smells in Flask and Django will be presented as well as the parameter used for the investigations.

5.3.1 Parameters

During the architectural recovery, there are three parameters needed by PySmell; dependency threshold, number of bricks, number of concerns. The optimal parameter tuning is unique for each system that is investigated and requires expert knowledge into the system to optimise, but several different combinations have been tested and evaluated. The parameters chosen for Flask and Django are presented in table 5.1 and 5.2.

Table 5.1: The parameter configuration for the architectural recovery of Flask.

Configuration Level	Dependency Threshold	Number of Bricks	Number of Concerns
Low (Blue)	1	12	4
Medium (Red)	1	20	6
High (Yellow)	1	28	12

[2]https://github.com/pallets/flask

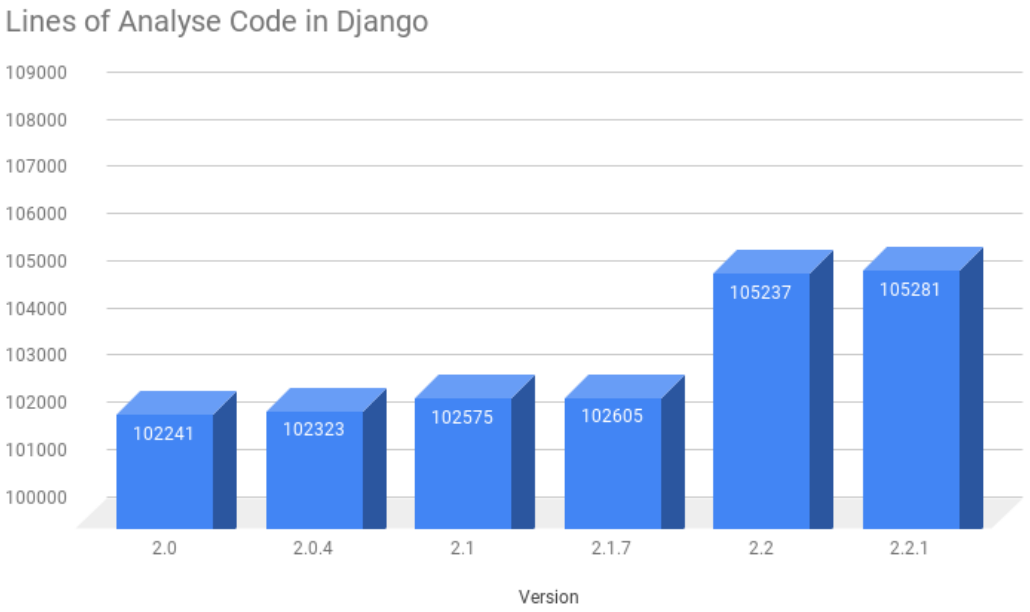

Figure 5.2: Illustration of how the analysed size of Django evolves over time.

Table 5.2: The parameter configuration for the architectural recovery of Django.

Configuration Level	Dependency Threshold	Number of Bricks	Number of Concerns
Standard (Blue)	5	35	16

5.3.2 Dependency Cycles

Figure 5.3 shows the number of dependency cycles found in Flask and our tool finds more and more cycles as the project grows when the low set of parameters are used. Otherwise, no cycles are found by our tool. In Django there is a bigger variation where the tool finds as few as four cycles in version 2.1 and as many as 89 cycles in version 2.2 as can be seen in figure 5.4.

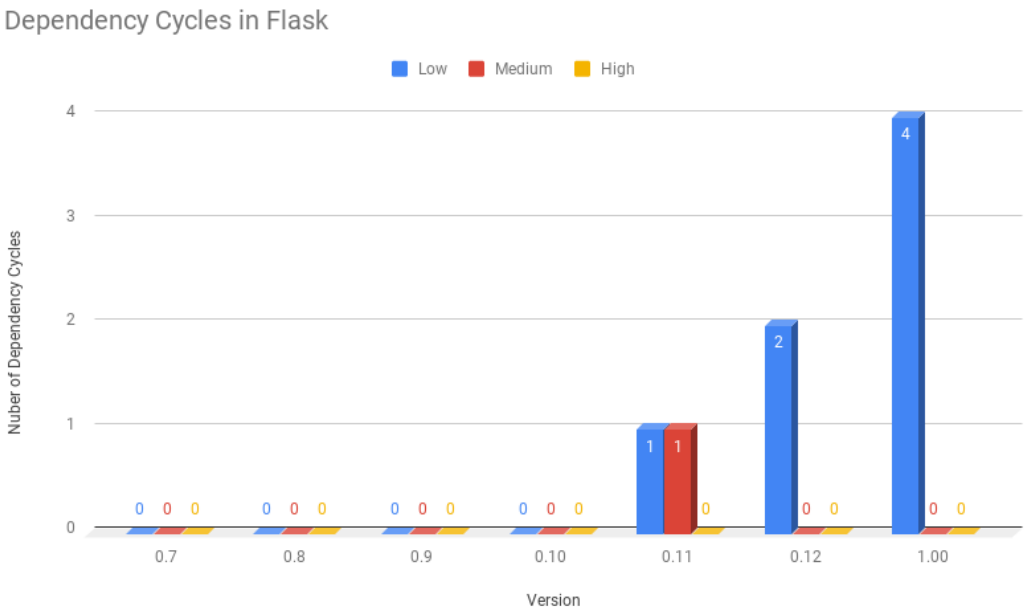

Figure 5.3: Illustration of the dependency cycles detected in different version of Flask.

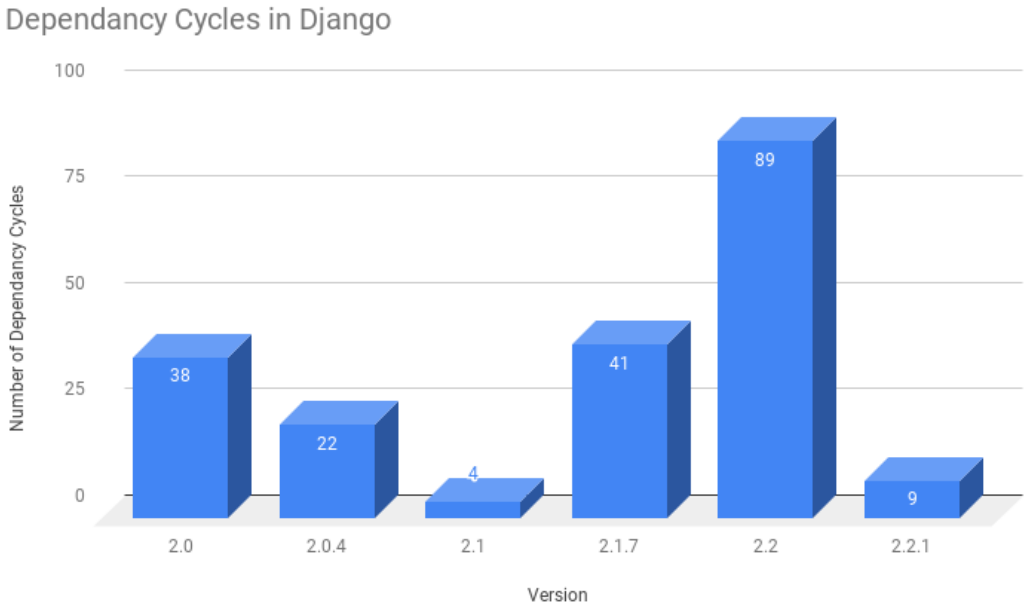

Figure 5.4: Illustration of the dependency cycles detected in different version of Django.

5.3.3 Link Overload

Link overload in Django as shown in figure 5.6 do not show any special trend but more a variation around ten Link Overloads, varying from fifteen in version 2.0 to six in version 2.2. The result from Flask also shows a significant variation that is very different depending on the set of parameters used. As seen in figure 5.5 there are more link overloads found using a

set of parameters resulting in many bricks with the highest link overload count of eleven in version 0.11. Fewer bricks and concerns results in fewer link overloads found.

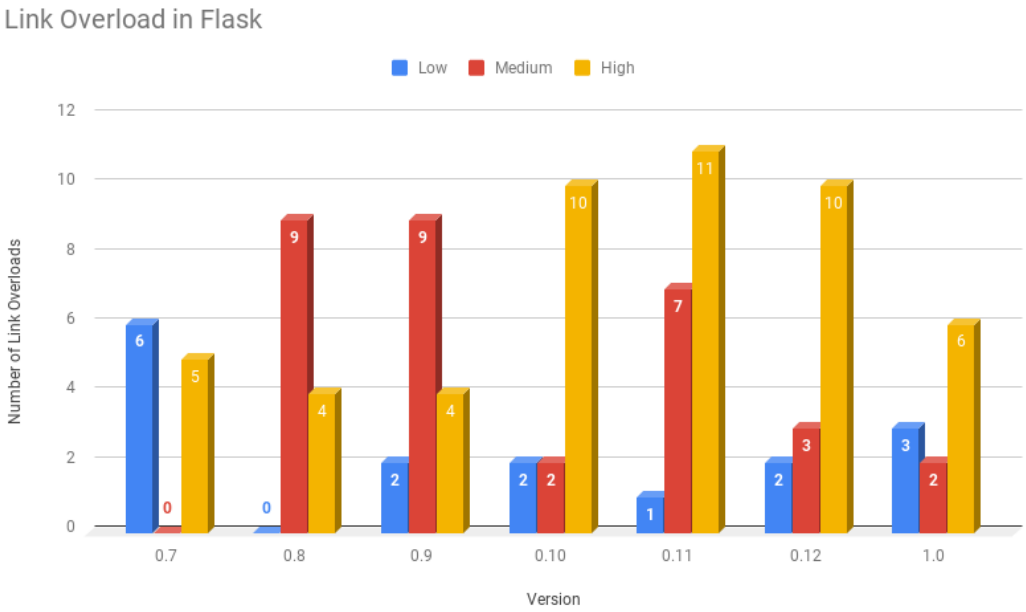

Figure 5.5: Illustration of the link overload detected in different version of Flask.

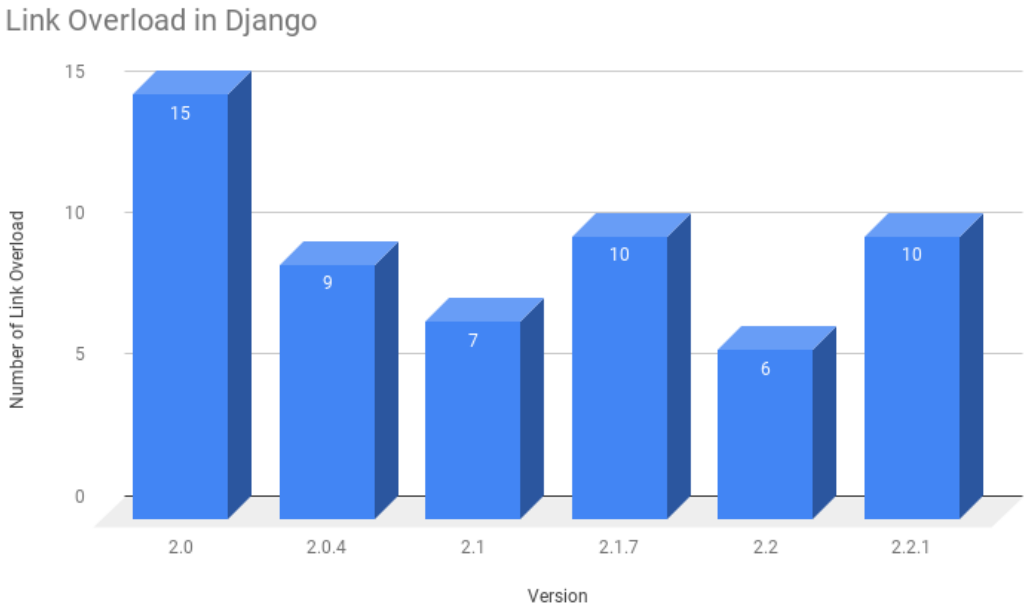

Figure 5.6: Illustration of the link overload detected in different version of Django.

5.3.4 Unused Entities

The number of unused entities is very high in Django with 1345 unused entities out of a total of 2637 entities in version 2.0, resulting in 51% of the entities goes unused. As the project

grows from 2637 entities to 2729 entities we see that the number of unused entities also grows to 1421 unused entities but the percentage is about the same with 52% of the entities that go unused. This is presented in figure 5.8 were we can see how the number of unused entities grows over time. In Flask we see a similarly high number of unused entities as in Django, with 63% of the entities not being used in version 0.7 as shown in figure 5.7. Unlike Django as the project grows we have a decrease in unused entities in relation to the total number of entities with version 1.0 of flask having 50% of its entities unused. The figure 5.7 also shows that the parameter settings have no impact on the results.

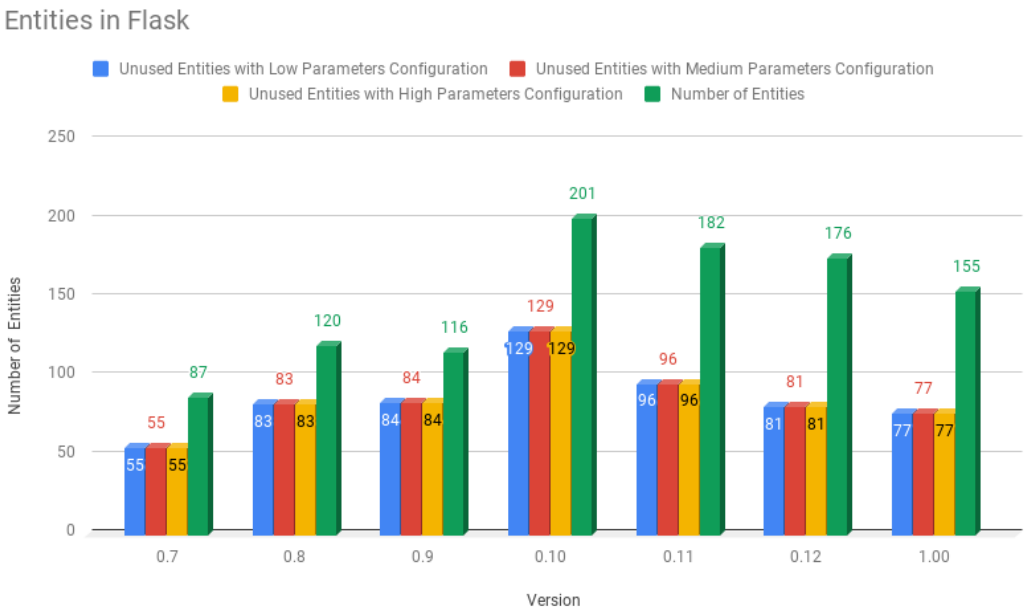

Figure 5.7: The unused entities detected in different version of Flask.

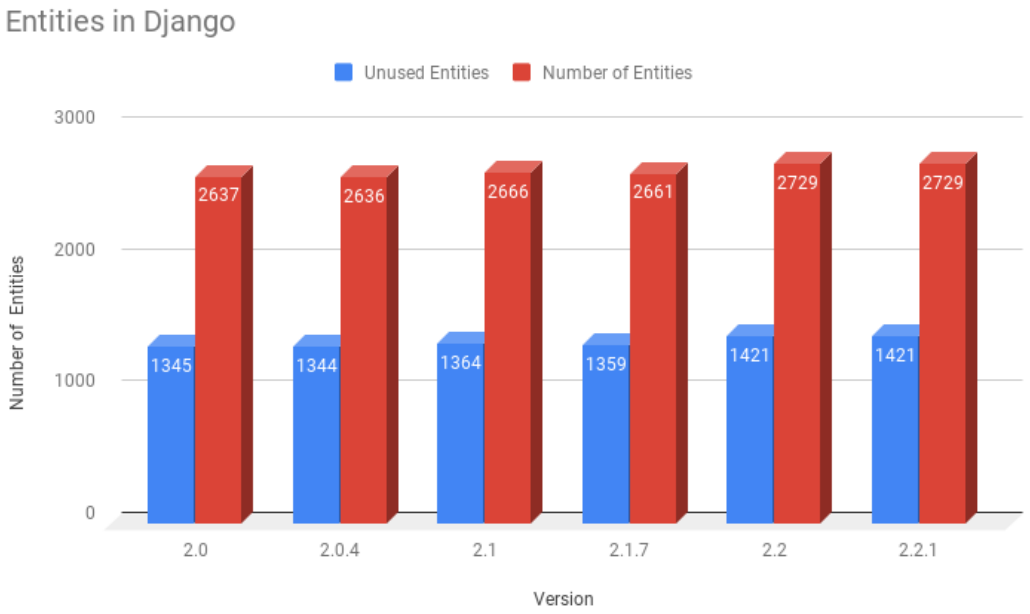

Figure 5.8: The unused entities detected in different version of Django.

5.3.5 Sloppy Delegation

In figure 5.10 we can see a quite stable level around 194 sloppy delegations with version 2.2 standing out with 203 sloppy delegations in Django. In Flask we can see how sloppy delegations increase from version 0.7 to 0.10 in figure 5.9 then the number of sloppy delegations decrease from version 0.10 to 1.0. Figure 5.9 also shows how the different parameter settings do not impact the number of sloppy delegations too much, the trend with an increase to version 0.10 and then a decrease to version 1.0.

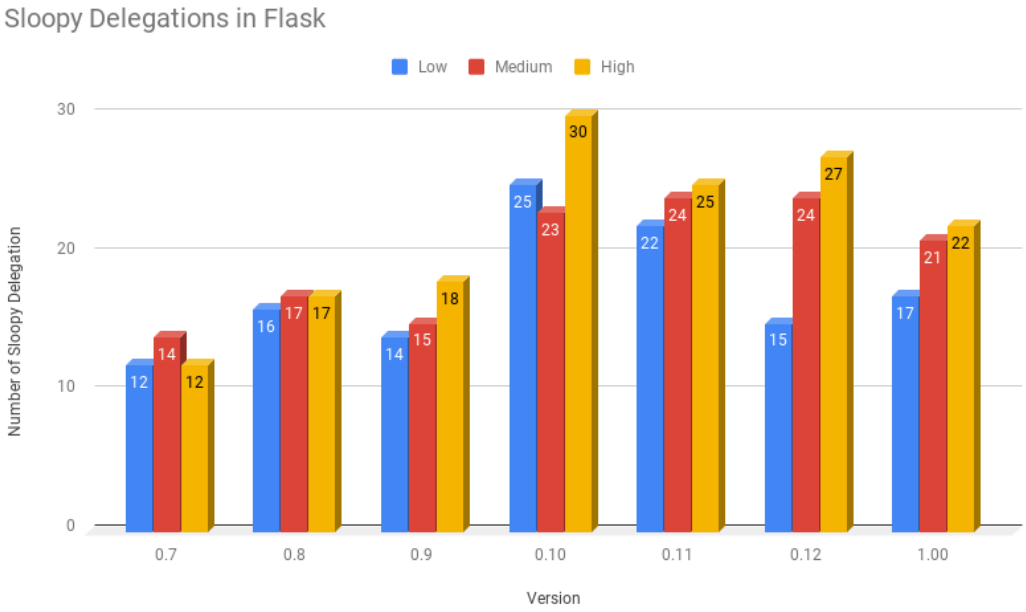

Figure 5.9: Graph representation of the sloppy delegation detected in Flask.

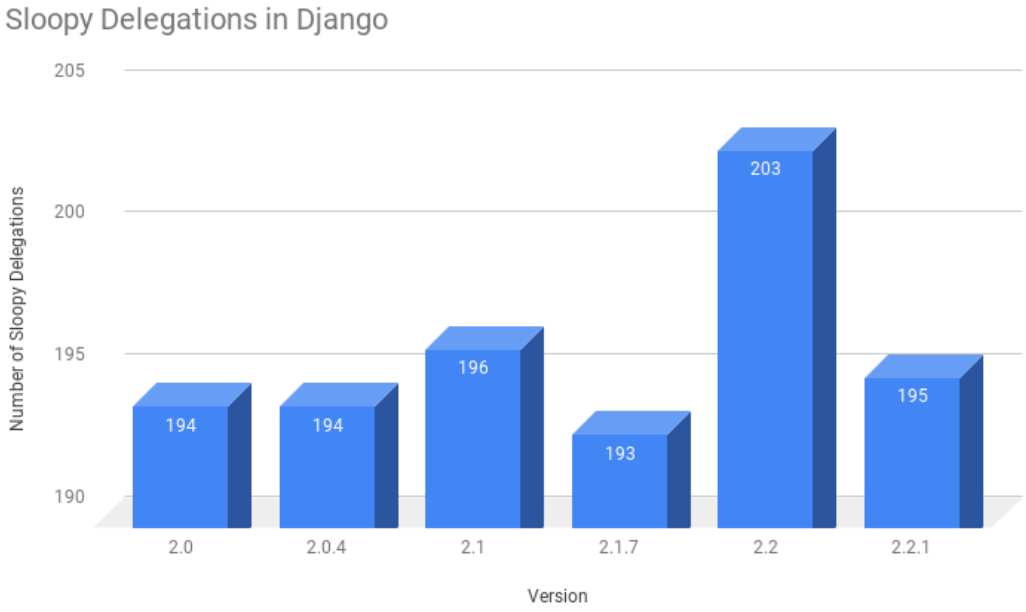

Figure 5.10: Graph representation of the sloppy delegation detected in Django.

5.3.6 Concern Overload

Concern overload detected in Django are illustrated in figure 5.11. In that figure, we see that the concern overloads propagation in the different versions of Django. We see a slight increase over time from version 2.0 with zero concern overloads to version 2.2.1 in figure 5.11 with two concern overloads. No concern overloads were detected in any version of Flask that

we tested. The three different parameter choices all yielded zero concern overloads in Flask. Possible reasons for those changes will be presented in the discussion chapter 6.

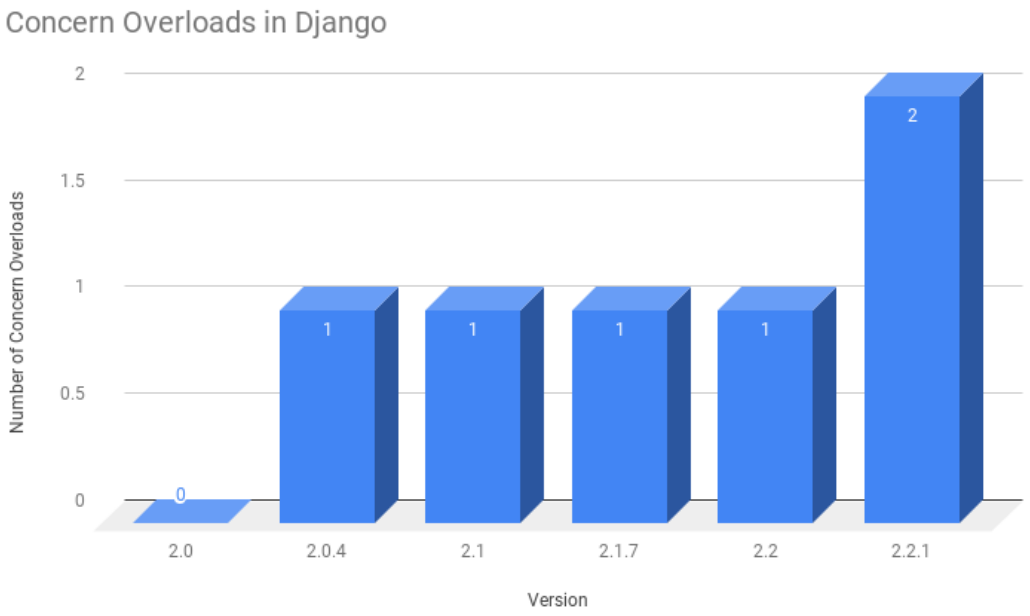

Figure 5.11: Graph representation of the concern overload detected in Django.

5.4 Validation

Results from validating PySmell by applying it on itself will be presented in this section. First, the parameters used will be presented then the architectural recovery accuracy and finally, the architectural smell detection and its accuracy will be presented.

5.4.1 Parameters

The following parameter configuration was chosen; six concerns, six bricks and no dependency threshold, as shown in table 5.3. The choice of dependency threshold equal to zero is because the system was of modest size, with only 46 entities. Therefore a single dependency between entities should be enough to create a dependency between components. The reasons behind choosing six concerns is a consequence of experimenting with the parameters because of the original thought of around five to eight possible concerns for the system. This initial thought of five to eight concerns stems in our expert knowledge of the system we developed. After experimenting the discovery was made that PySmell found concerns that matched what was expected by the architects of it, in regards to concerns for the system when using the six concerns. The same number of bricks yielded the best clustering results with the most accurate results.

Table 5.3: The parameter configuration for the architectural recovery of PySmell.

Dependency Threshold	# of Bricks	# of Concerns
0	6	6

5.4.2 Architectural Recovery

Using the parameters shown in table 5.3 the concerns presented in table 5.4 and the bricks presented in table 5.5 was recovered.

Table 5.4: Top ten words from each concern recovered from PySmell.

Concern 1	Concern 2	Concern 3	Concern 4	Concern 5	Concern 6
sql	threshold	graph	parser	test	file
relationship	brick	brick	node	entity	path
id	high	simple	text	python	files
concern	np	nx	path	sql	node
base	documents	type	args	repression	queue
tablename	queue	valueerror	file	inherits	ignore
probabilities	model	draw	topic	words	size
log	processed	dependency	ast	concern	args
dictionary	overload	cycles	init	session	param
entity	link	dependencies	model	overload	dependencies

Table 5.4 illustrates the different concerns recovered from PySmell. The hypothesis that Concern 1 is related to the database with words as "sql", "relationship", "id" and "tablename" is made even clearer by knowing that there are database classes called "concern", "probabilities" and "entity". Concern 2 is about detecting concern related smells and extracting concerns with the words: "threshold", "high", "brick" and "overload" clearly related to smell detection and the words: "documents", "processed" and "model" are frequently used in concern extraction. Detecting dependency related smells is what Concern 3 is about. Another obvious concern is Concern 6 that is about file parsing. With words such as "parser", "node", "path", "ast" and "file" Concern 4 relates to the structural extraction. Concern 5 leans towards concern extraction with words such as: "words" and "concern" but most of the words are quite general such as "python", "entity", "sql" and "session" and are used all through the system as well as in the concern extraction part.

All words in the top 10 word-lists for each concern is presented in table 5.4, which do not point straight to the concern. For example "topic" in Concern 4 do not relate to structural extraction and "log" in Concern 1 do not bring your thoughts towards databases. This is furthered discussed in section 6.2.

Table 5.5: Bricks recovered from PySmell.

Label	Total	Misplaced	Dependencies	Accuracy
Database	19	9	2	52.6%
Smell Detection	4	1	3	75%
File Traversal	10	2	12	80%
Concern Extraction	4	2	12	50%
File Parsing	7	4	9	42.8%
Structural Extraction	2	0	6	100%

In table 5.5 we can see the different bricks labelled after the majority of entities that belong together. We have the total number of entities in each brick, how many of these entities that were deemed to be displaced by the architectural recovery process. The dependencies column show the total number of dependencies that brick had. Finally, the accuracy column shows how big part of the entities within each brick that belonged there as a percentage. We can see that the accuracy for each brick is mostly over 50% and that corresponds to the total accuracy of the recovered architecture. Using manual inspection we found that of the 46 entities, 18

were misplaced. A misplaced entity was considered not to belong in the brick that it were placed in, by us the developers of PySmell. This resulted in an overall accuracy of 61%.

5.4.3 Architectural Smells

By applying PySmell to itself, PySmell indicated that the following architectural smells were active in the system; dependency cycles, sloppy delegation, link overload and unused entities. PySmell detected four dependency cycles within itself, where 50% of the indications were true-positive. Moreover, PySmell detected 15 instances of sloppy delegations, in which 26.67% were true positives. As for unused entities, PySmell detected ten occurrences of that and three of them were classified as true positives. Thus, achieving an accuracy of 30%. Continuing to the link overloads detected by PySmell in itself, it detects the presence of one link overload in the Database brick. That classification was then confirmed as a true-positive, thus yielding an accuracy of 100%. Lastly, no concern overload has been detected by PySmell within itself. These are exciting results and will be further discussed under Results in the Discussion chapter 6.2.

6 Discussion

With the result in mind, the next chapter offers reflections about the practices used and the extracted results.

6.1 Method

In this section, we discuss our method. We will try to answer the questions: why we choose to do as we did, what could have been done differently and what worked well.

6.1.1 ARC

As mentioned in the theory, Garcia *et al.* concluded that ARC and ACDC was the two most prominent techniques for architectural recovery. Our choice of ARC over ACDC was based on multiple things. Firstly, we believe that the text mining element with concerns is very language independent, since the aim is to write code that are easy to understand with good variable names and comments regardless of language. Adequate commenting of code and variable naming, is maybe something that developers do not always succeed with but that is a whole other issue. Secondly, we had experience with both clustering and text mining that we believed would make the development easier and faster, which was good as we had limited time.

6.1.2 Structural Extraction

As structural extraction have been done by many researchers on systems written in compiled and statically typed languages [10], [60], [27]. It was our curiosity to see how the methods used carried over into interpreted, dynamically typed languages like Python that made us create our tool. Because Python is interpreted, we only had the abstract syntax tree to work with and most importantly we only had limited time, thus we were not able to gather all dependencies. We chose to collect dependencies to both internal entities and external modules with a conservative attitude, making sure that we only collected dependencies we were sure were correct. This resulted in many entities not having any dependencies which resulted in some issues while clustering. There is room for improving the structural extraction catching

more dependencies, especially to external modules, when using decorators and for entities passed down as parameters to functions. There is also room for improvement in the structural extraction part of PySmell that does not have to do with dependencies. We believe other types of structural features, such as file names and directory paths as brought up in "Hierarchical Clustering for Software Architecture Recovery" [31], could be useful. However, this would have to be used with care since the goal often is to find issues such as functionality being implemented in the wrong part of the application when looking at architectural degeneration.

6.1.3 Concern Extraction

By applying text mining to the source code, using the text found in comments, variable and function names, to establish the concern of the entity. We could use a tf-idf representation of the bag of words mined from the text data and then later train an LDA model. Where possible areas for improvement and changes could be done in the bag of word representation of the data, which is the case $\mathcal{N} = 1$ in the $\mathcal{N}$-gram model, which can be implemented with any $\mathcal{N}$; $\mathcal{N} = 2$ gives the bigram representation, $\mathcal{N} = 3$ gives the trigram. Where the use of a large $\mathcal{N}$ gives the verbs in the sentences mined more weight. A higher value on $\mathcal{N}$ might have proved suitable for the text mined comments but not so well for the names and assignments, as we achieved useful results by only using the unigram model we decided it will suffice for this thesis, however this is an area which still could be further explored in regards to a combination of large values for $\mathcal{N}$ for the mined comments and the standard unigram for name and assignments.

Furthermore, we did not apply any stemming or lemmatisation of the text data. Where we decided on not performing it based on the article of Schofield *et al.* in "Understanding text pre-processing for latent Dirichlet allocation" and the thoughts that the grammatical form of a word has more meaning in the assignment and naming convention than in regular text, thus opting out of common practices [61].

The concern extraction procedure described in the literature for compiled languages regarding text mining generalises to interpreted languages as Python according to the literature [10], [29], [27]. All of the concern extraction is based on the text data extracted from comments and identifiers which should be descriptive about what the code does in both interpreted and compiled languages. Therefore, we believe the thought that there should be a cohesion with the concern of an entity with the corresponding text data in that entity to be sound. Thus, concern extraction should work equally well when applied to a multi-paradigm interpreted language as a compiled language.

The choice of machine learning method also offers multiple opportunities for improvements and experimentation. Two good candidates have been identified; Stochastic Gradient Descent (SGD) and Logistical Regression (LR). These two are the most recent stars in text mining, we choose to use LDA since it was used by Garcia *et al.* in the ARC recovery technique [29]. It would be interesting to see future research investigating if SGD and LR improves concern extraction and in turn architectural recovery.

6.1.4 Distance Measurement

The distances chosen for this thesis are; Jaccard and Jensen-Shannon. The reasons for that are both based on the findings of Maqbool and Babri[31], which concluded that the Jaccard family of distance measurements performs well on structural data extracted from source code. Moreover, Maqbool and Babri[31] states clear reasons for why structural data mined from

source code should be treated as binary asymmetrical, which worked well for the chosen data-structure, bitmap for cross-entity dependencies.

Continuing to the Jensen-Shannon distance, which is a well-known distance metric for probability distributions. It has been used in several papers when dealing with vectors of probabilities, which need a comparison to each other [29] [40].

Then to the question concerning the combination of the three distance metrics; external modules, dependencies and concerns to one. We used addition, which seemed like the most simple and straight forward way. There could unquestionably be another way of doing this, for example, a weighted summation (see equation (6.1)) could be applied and possibly produce a better result.

$$\mathcal{WS}_{distance}(e_i, e_j) = \alpha \cdot \mathcal{EM}(e_i, e_j) + \beta \cdot \mathcal{C}(e_i, e_j) + \gamma \cdot \mathcal{D}(e_i, e_j) \tag{6.1}$$

where: α, β, γ : are the different weights,
$\mathcal{WS}_{distance}$: the weighted sum of the distance metrics,
$\mathcal{EM}_{distance}$: the distance metric for external modules,
$\mathcal{C}_{distance}$: the distance metric for concerns,
$\mathcal{D}_{distance}$: the distance metric for dependencies.

An alternative to the weighted sum in equation (6.1) is the dissimilarity measurement for mixed type by Jiawei and Kamber, which is described equation is illustrated in (6.2) below:

$$\mathcal{M}_{dissimilarity}(i, j) = \frac{\sum_{a}^{\mathcal{A}} \delta_{ij}^{(a)} d_{ij}^{(a)}}{\sum_{a}^{\mathcal{A}} \delta_{ij}^{(a)}}. \tag{6.2}$$

where: $\mathcal{M}_{dissimilarity}$: the mixed type dissimilarity,
$\delta_{ij}^{(a)}$: indicates if attribute a is contributing to the dissimilarity between i and j,
$d_{ij}^{(a)}$: the distance between i and j, given attribute a,
i, j : the entities i and j,
$\mathcal{A}, a$: the set of attributes and the current attribute.

By applying equation (6.2), an alternative type of distance could be extracted between entities and evaluated to see if it would perform better or worse.

6.1.5 Clustering

We tried many different ways of clustering our entities before settling on Agglomerative clustering as suggested by Maqbool and Babri [31]. The most prominent other choice was K-Medoids because it was also able to cluster based on distances between entities. The reasons for choosing Agglomerative clustering over K-Medoids were many, first of all, we found the best support for it in the literature [31]. Secondly, there were more detailed documented implementations in widely used libraries for Agglomerative clustering, while K-Medoids implementations were only found on GitHub implemented by single developers without much

documentation. The final nail in the coffin was that K-Medoids took a very different amount of time to run with the same input, from about twenty minutes to none-terminating.

While clustering, we were limited both by time and available implementations of the different algorithms in Python. Not just what algorithms were implemented but also how they were implemented. For example, the Agglomerative clustering algorithm does not require a certain number of clusters to be specified beforehand, but the implementation that we encountered in the Python library sklearn[1] did require the number of clusters to be specified. This could have been mitigated by performing multiple clusterings with different parameters automatically and evaluating the clusters. This is possible since a distance matrix of all distances between entities can be pre-computed and is only needed to be done once. Computing the distances between all entities is the most computationally intensive task during the clustering. If we would perform multiple clusterings with a different number of clusters would require an automatic evaluation of the clusters to give any effect. This could be achieved with more time and is an area for future research.

6.1.6 Measuring Architectural Smells

As can be seen, if you compare the measuring of architectural smells described in the theory chapter, section 2.3 and in the method chapter, section 4.3 the method does not match the theory in every smell. We had to make some modifications to the algorithms either to fit our data structures or our study. Small implementation details are not brought up but the bigger ones are. Unlike in the theory, we count the number of unused entities instead of unused interfaces. In the theory unused interfaces are defined for strictly object oriented languages but since we work on a language that is not strictly object oriented we settled on counting unused entities instead. This means that if a class has one used method or a function is used, the entity is counted as used since we extract dependencies on an entity level. We believe our implemented smell detection indicates the same issue of code not being used as the one presented in the theory. Another difference between the theory and method are our addition of a dependency threshold for creating a dependency between two bricks. We made this addition since without it the detection of dependency cycles never terminated in larger projects like Django that had thousands of dependencies. We also found it reasonable that two bricks containing hundreds of entities needed more than one entity to be dependent on an entity in another brick for the entire brick to be dependent on that brick. We choose to set this manually from the parameters in our tool but we believe this could have been made better by using an adaptive threshold like the one used in the link overload, sloppy delegation and concern overload smells. Unfortunately, we did not have time to investigate this more and implement a better solution. Hopefully, further research can improve this area.

6.1.7 Parameters

As it can be seen in the results chapter 5, Flask is presented with three sets of parameter configurations; one with low values for concerns and bricks, one with a middle ground and one with high values. These setups yield different results for some of the detected architectural smells, which is to be expected. Thus, when performing the architectural recovery with a different number of bricks and a different number of concerns the resulting architecture should be of a similar structure but with a different granularity of the bricks. When using higher values for the bricks and concerns parameters, a finer granularity should be the result. This difference in granularity gives some variation in the detected smells which can be seen in the graphs for Flask. To best detect architectural degeneration, we believe a granularity as close to the intended architecture is desired. Otherwise a smell such as link overload will be

[1] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html accessed 2019-05-07

misleading as two or more components that should be one might have a lot of dependencies between each other. This can also be seen in the results, in figure 5.5 there can be seen how detected link overloads varies a lot depending on what parameters are used.

We did only use one set of parameters for Django because it took much longer to do the measurements on Django than it did on Flask. Since we only used one set of parameters we tried to choose a good match of parameters for Django. However, we are no experts on Django and we could probably have chosen a better set of parameters matching the intended architecture better if we had more knowledge about the intended architecture.

6.1.8 Threats to Validity

The process of reproducing the steps to create a tool that detects and classify architectural smells will always depend on the quality of implementation as well as the architectural smells chosen to investigate. However, the result of the tools created to match PySmell would yield result pointing in the same direction, but not necessarily exactly the same. Thus, leading to the same conclusion as ours, that the research regarding dynamically typed and interpreted languages is still immature and in need of further research.

Our validation method used for validating our tool PySmell is also a threat to the validity of our research. Since we use our own method and evaluate our own code there is a risk of bias that we have tried to keep to a minimum. A bigger issue is the size difference between the system we validated on, PySmell 1k LOC and the systems we investigated Flask 10k LOC and Django 500k LOC. As previously mentioned more research is needed.

6.2 Result

Here we discuss the results from our investigation were we applied the tool PySmell on the open source web frameworks Flask and Django. These results are presented in section 5.3.

6.2.1 Dependency Cycles

From our results from Flask visualised in figure 5.3, we can see that the number of dependency cycles we detect depends on the parameters used. We believe that the dependency threshold is the parameter with the most significant impact since it defines how many of the entities in brick A needs to be dependent on entities in brick B for A to be dependent on B. Therefore this parameter has a significant impact on the number of dependencies between bricks and more dependencies gives a higher probability for cycles.

The big spread of the number of cycles in different Django versions shown in figure 5.4 is interesting. It is possible that a step from version 2.1 to 2.2 increases the number of cycles because of new features that breaks the architecture. However, we do not think this is the only reason for such a big jump. The parameter settings might fit one version of Django much better than another and since Agglomerative clustering and LDA is dependent on the order of the data, the distribution of entities in bricks might have been very different. If there were more small bricks reconstructed from version 2.1, the dependency threshold might have been set too high for that version. The opposite might have been true in version 2.2; many large clusters might result in many more dependencies and thereby more cycles. This theory of parameter settings not fitting every version equally is supported by the drop in cycles from 2.2 to 2.2.1 here not as much have changed as between 2.1 and 2.2 in the form of new entities, but still, an even more significant change in the number of dependency cycles are found. This must not be because of how we set the parameters, of course, it can also be that many issues with 2.2 were addressed in 2.2.1 reducing the number of cycles. Perhaps a mixture of the two

explanations is the most likely reason for the high diversity in the number of dependency cycles in Django.

6.2.2 Link Overload

There do not seem to be any certain trend in how the link overload has changed as the system evolves in either Flask or Django as shown in the figures 5.5 and 5.6. We can see that how many link overloads that are found depends on the parameters as seen in figure 5.5. It is interesting that medium settings find more link overloads in versions 0.8 and 0.9 in Flask while high settings find more in 0.10, 0.11 and 0.12. As we can see in figure 5.1 Flask is significantly larger in version 0.10, 0.11 and 0.12 than in version 0.8 and 0.9. It might be so that the medium settings work better on Flask at a size of around 5k lines of analysed code (LOAC) than when there is 7k of LOAC and for that size, the high settings are a better match.

6.2.3 Unused Entities

There is a high percentage of unused entities in both Django and Flask as shown in figures 5.7 and 5.8 but this might not be as strange as it looks at first glance. Both Django and Flask are web framework and have exposed APIs for other applications to use. Therefore many of their functions and classes are not used by the project itself but are meant to be used by any web application developed using either Flask or Django. Other reasons for this high number of unused entities could be our conservative dependency parsing. Since we do not catch all dependencies and only add dependencies when we are sure that it is correct there are probably dependencies that we did not find and if we would find them the percentage of unused entities would probably be lower.

6.2.4 Sloppy Delegation

In Flask sloppy delegation follows the same curve as the number of entities as seen in figure 5.9 and 5.7. There is a very similar trend between parameters. This indicates that sloppy delegations relationship between entities and number of sloppy delegations is consistent. This might also be the case with Django, as it has about the same amount of entities as seen in figure 5.8. The biggest increase in entities is between version 2.1.7 and 2.2 and that is were we find the biggest increase in sloppy delegations as shown in figure 5.10. There we also see a decrease from version 2.2 to 2.2.1 but the amount of entities stay the same between these versions. Maybe some issues with 2.2 were worked out in 2.2.1 and therefore we see a decrease in sloppy delegations. This decrease in sloppy delegations from version 2.2 to 2.2.1 with a decrease of eight sloppy delegations, which is only a change of 4% which is not that much. An interesting thing is that in dependency cycles there is also a big decrease between version 2.2 and 2.2.1 and since both sloppy delegation and dependency cycles are dependency based smells there might be a connection. This supports the claim that there were some structural improvements between version 2.2 and 2.2.1 in Django.

6.2.5 Concern Overload

With the background of what concern overload represents in mind, a component trying to handle to many concerns. We can see in figure **??** that the concern overload smell is not detected in Flask, we believe one of the main reasons for that is found in the size of Flask. In a smaller projects like Flask, it is easier to make sure that there is a clear separation of concerns between components in the design. If we then compare it to Django which is a lot larger than Flask (13 times larger) we can see in figure 5.11 that PySmell detects concern overloads in Django, from version 2.0.4 and forward. A possible reason for this is in large scale software; the task of separating concerns becomes ever more complex as the size of the projects grows. Furthermore, one may imagine that Django as the larger project might have similar design

patterns in its internal architecture and therefore, trigger the concern overload architectural smell. That issue can have its roots in using a generic naming convention, which then would yield poorer results for the concern classification in the LDA model.

6.2.6 Validation

Choosing parameters for such a small project as PySmell with 1k LOAC that we had expert knowledge in, was much easier than for Flask and Django. Still, it took some time of manual work to test different kinds of parameters. Since our system was supposed to handle database access, structural extraction, concern extraction, brick reconstruction, smell detection, logging and parameter parsing we thought that somewhere between five and ten concerns would be a good choice. We also tried to structure the system with one component for each of these tasks so a similar amount of bricks were tested. The dependency threshold was set to zero because of the small size of the system with bricks consisting of only two entities a higher threshold would be unreasonable. We believe this parameter setup is optimal for PySmell and yielded the best possible result as could not have been said for Django and Flask since there we lacked the expert insights into the systems.

In the architecture recovered there can be seen that the Database brick is almost double the size of the second largest brick and has a low accuracy. This pattern with one or two clusters much bigger than the rest is also present in the architectural recovery of Flask and Django. In PySmell it is the Database brick that becomes largest. There can be many explanations for why one brick becomes much larger than the others. In PySmell the database classes are used in many parts of the system and they are involved in a broad mixture of concerns since there are database model classes for many different types of objects. This combined with that all database classes uses the SQLAlchemy module and have similar words such as "relationship", "id" and "sql" that tie the database entities together results in a brick that a lot of entities are close to. Either an entity uses a database class or they are involved in the same concern as one or more of the database classes since they cover most concerns. These reason is quite unique for PySmell even though both Django and Flask also have database access as parts of their systems. Looking closer at the dependencies we found that only 2 of the 19 entities in the Database brick have a dependency as shown in table 5.5, compared to the average of 7.3 entities with dependencies per brick in PySmell. This makes us believe that if more dependencies were found the distribution of entities would improve. There is also the fact that the database layer is the lowest layer in PySmell. This results in it not being dependent on any other part of the application, reducing the number of dependencies to be found.

Overall when we calculate the distance between two entities we only compare what dependencies these two have in common. This does not fully utilise the structural information. If we also looked at if the two entities were dependent on each other or in the same chain of dependencies this would probably improve our results. This could be part of a more general solution solving the issues with big bricks in not only PySmell but Flask and Django as well.

Observation of the architectural smell validation reveals a poor accuracy in most smells. We believe the errors created in the architecture recovery process propagates down into the smell detection. This is probably because the smell detection is so dependent on the architectural recovery. This results in it being quite hard to draw conclusions about how good the smell detection techniques were. Some things however, can be further discussed. As we have previously discussed the issue with not discovering all dependencies affects the architectural recovery badly, it also affects the smell detection both indirectly through the architectural recovery and directly in the unused entities smell. The low accuracy of 30% in the unused entities smell increases the light on the dependency issue that needs to be solved in future

research. Another interesting smell with low accuracy was sloppy delegations. When validating it we discovered how important it is for each entity to be put in the right component. Since the smell detects when a component delegates a task that it should do by itself to another component. A misplaced entity is often found as a sloppy delegation when it is only used by entities in its "correct" component. This is evidence supporting our hypothesis that bad accuracy in the architectural recovery propagates into the smell detection. With a more accurate recovered architecture, we believe there would be a much better smell detection accuracy without more work in that area.

6.3 Research Question

> *How does architectural degeneration evolve in large scale systems written in the dynamically typed multi-paradigm and interpreted language Python?*

It is quite hard to answer our research question when only looking at two different systems we have investigated, especially since Flask is not that big of a software system. There is also much more work to be done with improving the structural extraction to get results reliable enough to draw any conclusions. We have on the other hand made progress towards answering this question as we have broken new ground in measuring architectural degeneration in large scale systems written in the dynamically typed multi-paradigm language Python. Since to our knowledge, there have been no attempts at measuring architectural degeneration in systems written in Python. Issues encountered while doing this type of research have been discovered and that will hopefully help guide future research.

We could have looked at more systems but we do not think it would have created more value towards answering our research question since our validation did indicate that our accuracy when recovering the architecture was low, resulting in an inaccurate smell detection. This low number however is also brought up in previous research [27]. Garcia *et al.*[27] uses ARC and gets an average accuracy of 58.76% which is lower than ours on 61%. However, these two numbers should not be compared too much since there are two completely different validation methods and our validation was done on a project with around 1k LOC and Garcia *et al.*[27] used eight projects with 70k - 4M LOC for their validation. We would have needed more time to develop and validate our tool for it to be good enough to draw any solid conclusion from the results. There is also the issue as brought up by Fontana *et al.* [60] were you need experts to use tools for recovering architectures if you want the best results. To fully answer this question we would need more time to perfect our tool and developers from many different big Python systems to use the tool.

From our results, we might not be able to see how architectural degeneration evolves but our results support the thesis that bigger systems have more architectural issues than smaller systems in dynamically typed multi-paradigm interpreted languages as have been a known for statically typed compiled languages [6].

7 Conclusion

In this chapter, conclusions regarding the research question will be presented together with overall insights regarding the architectural degeneration and its effects on large scale systems written in Python.

In this thesis, we can see a proof-of-concept implementation for architectural recovery and architectural smell detection in the area of the dynamically typed, multi-paradigm and interpreted language Python. Using methods that were illustrated in the literature for compiled languages, in which they claimed that these methods also generalises to all languages [10]. We found that this claim has some traction. The ARC method did enable us to recover an architecture, but more work is needed into structural information extraction to fully verify this claim by recovering an architecture with a higher accuracy than the 61% that we accomplished. A major issue we identified is extracting dependencies from the source code, when functions are not just called but for example, used as decorators or used as parameters and therefore, change the identifier before being called.

Even with further research in the area of architectural recovery, there will still be a need for domain expert involvement in the process, performing manual labour to choose parameters and validate the recovered architecture before applying the smell detection to obtain the optimal results. We can conclude that this will still be the case for some time, but with more research, hopefully, a method for recovering architectures without manual oversight will be presented in the future.

In direct regard to our research question:

> *How does architectural degeneration evolve in large scale systems written in the dynamically typed multi-paradigm and interpreted language Python?*

As to be expected of large scale software systems, with size comes complexity, and the need for architecture principles becomes ever more valuable. As it has been brought forward in both the discussion and the result, we see a slight correlation between the size of the project

and the detected architectural smells. Thus, indicating that the architectural degeneration also has increased overtime, on the performed architectural recovery. The correlation between the two is weak and need further investigation to be able to be empirically verified.

Bibliography

[1] M. Cianchi and A. Vezzosi, *Leonardo da Vinci's Machines*. Becocci Editore, 1988.

[2] E. G. Daylight, "Towards a historical notion of 'turing—the father of computer science'", *History and Philosophy of Logic*, vol. 36, no. 3, pp. 205–228, 2015.

[3] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture", *ACM SIGSOFT Software engineering notes*, vol. 17, no. 4, pp. 40–52, 1992.

[4] J. O. Coplien and G. Bjørnvig, *Lean architecture: for agile software development*. John Wiley & Sons, 2011.

[5] L. Hochstein and M. Lindvall, "Diagnosing architectural degeneration", in *28th Annual NASA Goddard Software Engineering Workshop, 2003. Proceedings.*, Dec. 2003, pp. 137–142. DOI: 10.1109/SEW.2003.1270736.

[6] L. Hochstein and M. Lindvall, "Combating architectural degeneration: A survey", *Information and Software Technology*, vol. 47, no. 10, pp. 643–656, 2005.

[7] M. W. Godfrey and E. H. Lee, "Secrets from the monster: Extracting mozilla's software architecture", in *Proceedings of Second Symposium on Constructing Software Engineering Tools (CoSET'00)*, Citeseer, 2000.

[8] Z. Li and J. Long, "A case study of measuring degeneration of software architectures from a defect perspective", in *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, IEEE, 2011, pp. 242–249.

[9] M. Lindvall, R. Tesoriero, and P. Costa, "Avoiding architectural degeneration: An evaluation process for software architecture", in *Proceedings Eighth IEEE Symposium on Software Metrics*, Jun. 2002, pp. 77–86. DOI: 10.1109/METRIC.2002.1011327.

[10] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software", in *2018 IEEE International Conference on Software Architecture (ICSA)*, IEEE, 2018, pp. 176–17 609.

[11] EU, https://web.archive.org/web/20150208090338/http://ec.europa.eu/enterprise/policies/sme/facts-figures-analysis/sme-definition/index_en.htm, Accessed 20-12-2018, Archived 2-2-2015.

[12] M. W. Maier, D. Emery, and R. Hilliard, "Software architecture: Introducing ieee standard 1471", *Computer*, vol. 34, no. 4, pp. 107–109, Apr. 2001, ISSN: 0018-9162. DOI: 10.1109/2.917550.

[13] C. Gacek, A. Abd-Allah, B. Clark, and B. Boehm, "On the definition of software system architecture", in *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, 1995, pp. 85–94.

[14] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Comparing software architecture recovery techniques using accurate dependencies", in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, IEEE, vol. 2, 2015, pp. 69–78.

[15] R. T. Tvedt, P. Costa, and M. Lindvall, "Does the code match the design? a process for architecture evaluation", in *International Conference on Software Maintenance, 2002. Proceedings.*, IEEE, 2002, pp. 393–401.

[16] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data", *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[17] J. Van Gurp and J. Bosch, "Design erosion: Problems and causes", *Journal of systems and software*, vol. 61, no. 2, pp. 105–119, 2002.

[18] L. De Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey", *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, 2012.

[19] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic, "Mapping architectural decay instances to dependency models", in *Proceedings of the 4th International Workshop on Managing Technical Debt*, IEEE Press, 2013, pp. 39–46.

[20] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms", in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, IEEE, 2012, pp. 277–286.

[21] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice", *Ieee software*, vol. 29, no. 6, pp. 18–21, 2012.

[22] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic, "Relating architectural decay and sustainability of software systems", in *Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on*, IEEE, 2016, pp. 178–181.

[23] M. Flower and K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[24] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[25] R. J. Hyndman and Y. Fan, "Sample quantiles in statistical packages", *The American Statistician*, vol. 50, no. 4, pp. 361–365, 1996.

[26] C. H. Yu, "Exploratory data analysis", *Methods*, vol. 2, pp. 131–160, 1977.

[27] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques", in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Press, 2013, pp. 486–496.

[28] A. Shahbazian, Y. K. Lee, D. Le, Y. Brun, and N. Medvidovic, "Recovering architectural design decisions", in *2018 IEEE International Conference on Software Architecture (ICSA)*, IEEE, 2018, pp. 95–9509.

[29] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns", in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, IEEE, 2011, pp. 552–555.

[30] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural change in open-source software systems", in *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, IEEE, 2015, pp. 235–245.

[31] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery", *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, Nov. 2007, ISSN: 0098-5589. DOI: 10.1109/TSE.2007.70732.

[32] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation", *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[33] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language", *Computational linguistics*, vol. 18, no. 4, pp. 467–479, 1992.

[34] A. M. Kibriya, E. Frank, B. Pfahringer, and G. Holmes, "Multinomial naive bayes for text categorization revisited", in *Australasian Joint Conference on Artificial Intelligence*, Springer, 2004, pp. 488–499.

[35] S. Vijayarani, M. J. Ilamathi, and M. Nithya, "Preprocessing techniques for text mining-an overview", *International Journal of Computer Science & Communication Networks*, vol. 5, no. 1, pp. 7–16, 2015.

[36] L. Bottou, "Large-scale machine learning with stochastic gradient descent", in *Proceedings of COMPSTAT'2010*, Springer, 2010, pp. 177–186.

[37] B. Krishnapuram, L. Carin, M. A. Figueiredo, and A. J. Hartemink, "Sparse multinomial logistic regression: Fast algorithms and generalization bounds", *IEEE transactions on pattern analysis and machine intelligence*, vol. 27, no. 6, pp. 957–968, 2005.

[38] F. Murtagh, "A survey of recent advances in hierarchical clustering algorithms", *The computer journal*, vol. 26, no. 4, pp. 354–359, 1983.

[39] T. Velmurugan and T. Santhanam, "Computational complexity between k-means and k-medoids clustering algorithms for normal and uniform distributions of data points", *Journal of computer science*, vol. 6, no. 3, p. 363, 2010.

[40] J. Lin, "Divergence measures based on the shannon entropy", *IEEE Transactions on Information theory*, vol. 37, no. 1, pp. 145–151, 1991.

[41] G. Poels and G. Dedene, "Distance-based software measurement: Necessary and sufficient properties for software measures", *Information and Software Technology*, vol. 42, no. 1, pp. 35–46, 2000.

[42] L. B. Wilson and R. G. Clark, *Comparative programming languages*. Pearson Education, 2001.

[43] G. Michaelson, *An introduction to functional programming through lambda calculus*. Courier Corporation, 2011.

[44] D. Alic, S. Omanovic, and V. Giedrimas, "Comparative analysis of functional and object-oriented programming", in *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2016, pp. 667–672. DOI: 10.1109/MIPRO.2016.7522224.

[45] A. Jayal, S. Lauria, A. Tucker, and S. Swift, "Python for teaching introductory programming: A quantitative evaluation", *Innovation in Teaching and Learning in Information and Computer Sciences*, vol. 10, no. 1, pp. 86–90, 2011.

[46] C. Severance, "Guido van rossum: The early years of python", *Computer*, vol. 48, no. 2, pp. 7–9, 2015.

[47] Python Software Foundation, https://docs.python.org/3/faq/general.html, Accessed 29-04-2019.

[48] ——, https://docs.python.org/3.6/archives/python-3.6.7-docs-pdf-a4.zip, Accessed 02-05-2019.

[49] G. Van Rossum and F. L. Drake, *The python language reference manual*. Network Theory Ltd., 2011.

[50] J. Aycock, "Compiling little languages in python", in *Proceedings of the 7th International Python Conference*, 1998, pp. 69–77.

[51] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees", in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, IEEE, 1998, pp. 368–377.

[52] A. Tornhill, "Prioritize technical debt in large-scale systems using codescene", in *Proceedings of the 2018 International Conference on Technical Debt*, ACM, 2018, pp. 59–60.

[53] A. Tornhill, "Assessing technical debt in automated tests with codescene", in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Apr. 2018, pp. 122–125. DOI: 10.1109/ICSTW.2018.00039.

[54] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest", in *Proceedings of the 2nd Workshop on Managing Technical Debt*, ACM, 2011, pp. 1–8.

[55] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability", in *null*, IEEE, 2007, pp. 30–39.

[56] A. Martini, J. Bosch, and M. Chaudron, "Architecture technical debt: Understanding causes and a qualitative model", in *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, IEEE, 2014, pp. 85–92.

[57] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, *et al.*, "Managing technical debt in software-reliant systems", in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, ACM, 2010, pp. 47–52.

[58] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Cimpan, and H. Verjus, "Towards a process-oriented software architecture reconstruction taxonomy", in *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, IEEE, 2007, pp. 137–148.

[59] A.-D. Seriai, M. Huchard, C. Urtado, S. Vauttier, *et al.*, "Mining features from the object-oriented source code of software variants by combining lexical and structural similarity", in *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*, IEEE, 2013, pp. 586–593.

[60] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan: A tool for architectural smells detection", in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, IEEE, 2017, pp. 282–285.

[61] A. Schofield, M. Magnusson, and D. Mimno, "Understanding text pre-processing for latent dirichlet allocation", in *Proceedings of the 15th conference of the European chapter of the Association for Computational Linguistics*, vol. 2, 2017, pp. 432–436.

[62] H. Jiawei and M. Kamber, *Data mining: Concepts and techniques, (the morgan kaufmann series in data management systems), vol. 3*.