2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678947

Where to Start: Studying Type Annotation Practices in Python

Wuxia Jin[†‡], Dinghong Zhong[†‡], Zifan Ding[†‡], Ming Fan[‡], Ting Liu[‡]*
jinwuxia@mail.xjtu.edu.cn, {zhongdh,dingzifan19}@stu.xjtu.edu.cn,{mingfan,tingliu}@mail.xjtu.edu.cn
[†] School of Software Engineering, Xi'an Jiaotong University, Xi'an, China
[‡] Ministry of Education Key Laboratory of Intelligent Networks and Network Security, Xi'an Jiaotong University

***Abstract*—Dynamic programming languages have been embracing gradual typing, which supports optional type annotations in source code. Type-annotating a complex and long-lasting codebase is indeed a gradual and expensive process, where two issues have troubled developers. First, there is few guidance about how to implement type annotations due to the existence of non-trivial type practices; second, there is few guidance about which portion of a codebase should be type-annotated first. To address these issues, this paper investigates the patterns of non-trivial type-annotation practices and features of type-annotated code files. Our study detected six patterns of type-annotation practices, which involve recovering and expressing design concerns. Moreover, we revealed three complementary features of type-annotated files. Besides, we implemented a tool for studying optional typing practice. We suggest that: 1) design concerns should be considered to improve type annotation implementation by following at least six patterns; 2) files critical to software architecture could be type-annotated in priority. We believe these guidelines would promote a better type annotation practice for dynamic languages.**

***Index Terms*—type annotation, dynamic languages**

I. Introduction

Dynamic programming languages have been embracing gradual typing [1], supporting optional type annotations in source code. For example, Python community proposes PEP483 to support optional type hints for Python 3.5 and later versions in 2014. TypeScript [2] with the first release version in 2014 allows static type definitions for JavaScript. In 2020, Ruby 3.0 [3] claims featuring type annotations by introducing RBS [4]. This trend indicates that dynamic typing community increasingly acknowledges the benefits of type annotations, i.e., facilitating early bug detection and software maintenance [5], [6]. Notable Python communities like Apache PySpark have started modern type annotation practices on their long-lasting codebases. By *long-lasting* codebases, we mean that they were originally developed during that period of optional typing feature unavailable.

Adding type annotations requires a monumental effort [5]. Type-annotating a complex and long-lasting codebase is not an overnight process, indeed a gradual migration from unannotated codebase to a completely type-annotated one. During this gradual and expensive process, two issues have troubled developers:

First, there is few guidance on how to implement type annotations. Some type annotation implementations require additional definitions, besides the types themselves. We define them as *non-trivial* type annotation practices. Recent studies [7], [8] concluded that adding type to a single variable consumes about two minutes on average. However, the patterns followed by non-trivial type annotation are still unclear. Revealing these patterns would promote an effective and efficient type annotation practice.

Second, there is few guidance on which portion of a codebase should be type-annotated in priority. When type-annotating a large-scale codebase, developers have struggled with where to start adding types–*"It should be decided if annotations should cover only the public API(s), or internals as well"*, as stated by PySpark developers [9]. They agree that there should be *"a trade-off between completeness (of the typing coverage) and the cost of maintenance (of type annotations)."* However, it is unclear for type beginners about which source files can be type-annotated with priority.

This paper focuses on the two issues by studying popular Python projects that have experienced type annotations. Two significant considerations motivate our study. First, we conjecture that the inconsistency between type annotation implementations and the corresponding source code may imply the patterns of non-trivial type annotations, due to the introduction of new code entities. Second, we assume that type-annotated files are likely *critical* to maintaining a software system. The reason is that one intention of using types is to improve codebase maintainability, which is commonly evaluated based on code dependency structures [10], [11]. As a result, we will explore three research questions:

RQ1. What are the patterns that non-trivial type-annotation practices follow?

RQ2. Do type-annotated files present different dependency structure?

RQ3. Do type-annotated files incur different maintenance cost?

To support our study, we selected notable Python projects with type hint experiences. These subjects are with diverse

* Ting Liu is the corresponding author.

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00054

Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on February 15,2023 at 12:34:51 UTC from IEEE Xplore. Restrictions apply.

domains, sizes, and type-hint manners [12]. We designed the THProfiler to analyze projects' source code and mine their revision history. We have created and released the dataset [13].

First, our study figured out six patterns followed by non-trivial type hint practices. They are *Typing Compatability*, *API Visibility*, *BaseClass Presentation*, *Function Overloading*, *Function-assigned Variable*, and *Typing Extension*. These type annotation practices heavily involve recovering and expressing design or code decisions made by developers during codebase development. Second, we revealed three features of type-annotated files. Type-annotated files have a bigger value of degree centrality in dependency structure, reside at the uppermost layer in the hierarchical structure, and incur higher maintenance costs.

Our work is the first to inspect non-trivial type-annotation practices and identify features of type-annotated files, to the best of our knowledge. Our findings benefit type hint practices and tools– 1) architecture concerns should be considered to improve the effectiveness and efficiency of type implementations by following these patterns; 2) files critical to software architecture could be type-annotated in priority for a gradual migration from un-annotated codebase to type-annotated one.

Overall, our work makes the following contributions:

1) We define six patterns followed by non-trivial practices and reveal three features of type-annotated files.

2) We suggest following these patterns with design concerns to implement type annotations; we recommend type-annotating files with the revealed features in priority.

3) We contribute a dataset and a tool for the continued study of type annotation practices in dynamic languages.

The organization of the rest: Section II presents the key concepts and our THProfiler tool. Sections III, IV, and V report the study setup, study results, and potential impact of our empirical study. Sections VI and VII discuss the threats to validity and related work. Section VIII draws conclusions.

II. Methodology

This section will introduce the concepts and THProfiler we designed to support our study.

A. Concepts

Listing 1: Example A

```
#a is type-annotated
#b,c are un-annotated
def func(a:int, b):
  c = a + b
  return c
```

Listing 2: Example B

```
#_P is a newly defined
_P=TypeVar("_P",bound=int)
def func(a:_P, b):
  c = a + b
  return c
```

1) Optional Type Annotation: Gradual typing [1] supports optional type annotations in source code. The "optional" means that developers can add type annotations to a portion of source files, and functions or variables inside one file. Only type-annotated entities will be statically type-checked while others remain type-checked at run-time. Listing 1 shows a snippet of Python code. In the function $func$, its parameter a is annotated with int while the parameter b and a variable c have no type annotations.

2) Type Annotation Implementation: Following PEP484 [12], Python3.5 and later versions support two manners of type annotation implementation:

inline type hints. Type annotations are directly inlined in the source code. In Listing 3, type hints of var and add are inlined in $test.py$.

stub files. Type annotations are separately managed in stub files named with $*.pyi$, which only contain type hints and are only used by static type checkers (i.e., mypy [14], pytype [15]). In Listing 4, $test.pyi$ is the stub file corresponding to $test.py$. $test.pyi$ only contains type-associated declarations and function signature of add. The function body of add is a single ellipsis (...).

Python projects can adopt either one or both two manners above at the same time.

Listing 3: Inline type hints

```
#test.py


var: int = 10

def add(a:int,b:int)
    ->int:
  c = a + b
  return c
```

Listing 4: Stub file

```
#test.py
var = 10
def add(a,b):
  c = a + b
  return c

#test.pyi
var: int
def add(a:int,b:int)->int:...
```

3) Trivial and Non-trivial Type Practices: We classify type annotation implementations into two types:

Trivial type practices mean that type-annotating a code entity simply requires assigning the type by following a syntax of : *type*. In this case, the source code before and after type annotation implementations are consistent with each other. Type-annotating a in Listing 1 is trivial by only adding $: int$.

Non-trivial type practices are complex implementations that require introducing additional definitions such as variables, functions, and classes. This practice leads to the inconsistency between the code and type-annotated code. In listing 2, $\_P$ is introduced to type-annotate a.

B. THProfiler

We designed and implemented THProfiler to support our study. THProfiler analyzes the source code and mines revision history of type-annotated Python codebases. As illustrated in Figure 1, the input of THProfiler is a Python codebase with type implementations. THProfiler consists of three parts as follows.

1) Typing Practice Identification: This part detects entities annotated with types and type usages in source code.

Typing Coverage Detection. Using Python *ast* library, this module first identifies all code entities and type-annotated entities. Then it calculates the typing coverages at various levels. We define *typing coverage* as the proportion of entities (i.e., functions, files, variables) with type hints.

Diverse Type Extraction. Through traversing the AST structures of source files and stub files (if available), this module detects all types assigned to entities.

Complex Typing Usage Detection. This module detects the usage of protocols and function overloadings. Overloaded

Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on February 15,2023 at 12:34:51 UTC from IEEE Xplore. Restrictions apply.

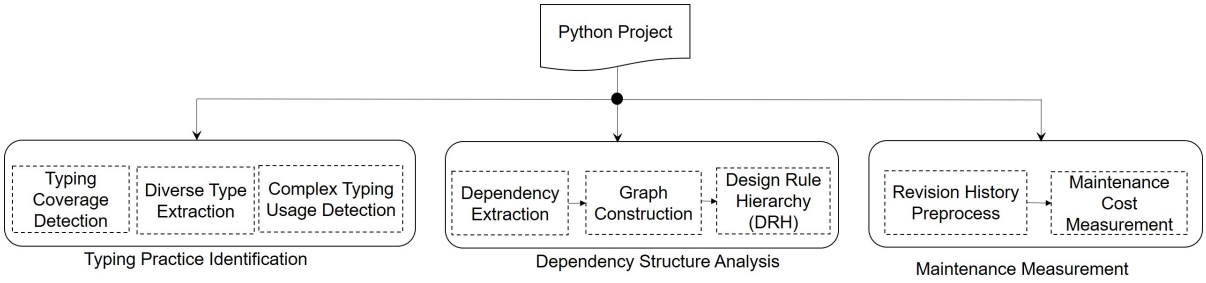

Fig. 1: The overview of THProfiler

functions are declared with $@overload$ decorator, which is imported from $typing$ module. The $@overload$ decorator allows describing functions with same names but with different combinations of argument types [16]. Protocol classes [17] explicitly extend from $typing.Protocol$ that supports *structural* typing. Structural typing can be considered as a static equivalent of duck typing [18]. Protocol classes act as an implicit base-class in the static type analysis. That is, if a class has some members that are also defined by protocols, this class can be (implicitly) treated as a sub-class of the protocols.

2) Dependency Structure Characterization: This part extracts and analyzes dependency structure of source code, following the Design Rule Theory [19]. Design Rule Theory assumes that design rules and modules are major design concerns in a software system. Design rules decouple the rest of a system into mutually independent modules. In the source code that follows Object-Oriented Programming pragmatics, design rules are often manifested as interfaces or abstract classes. The removal, modification, and addition of a module should have no influence on design rules.

Dependency Extraction. This module considers both explicit and possible dependencies in source code. As termed by Jin et al. [10], *explicit dependencies* are syntactic dependencies that are explicitly referenced in source code, while possible dependencies are invisible and non-deterministic dependencies due to duck typing [18]. Following this work [10], this module also employs SCITool Understand [20] and ENRE [21] to extract explicit and possible dependencies, respectively.

Graph Construction. This module constructs the Attributed Dependency Graph (ADG) from the union of explicit and possible dependencies. In an ADG, each node denotes one source file, and each directed edge denotes one dependency between two files. The node attribute specifies whether a node has type hints (denoted as $typed$) or not. The attribute information is obtained from the *Typing Practice Identification* of THProfiler.

Design Rule Hierarchy (DRH). Based on the ADG, this module employs DRH [22], [23] algorithm to identify design rules and independent modules in a software system. Following the Design Rule Theory [19], DRH clusters source files into a layering structure, where files in upper layers represent design rules, and files in lower layers are organized into independent modules decoupled by those design rules. Modules in lower layers depend on the modules in upper layers, but not vice versa. Modules in the same layer are mutually independent.

We use a subset of source files in Django, one of our subjects, to explain hierarchical structures created by DRH. In Figure 2, each row or column corresponds to a file and "dp" in a cell (i, j) indicates one dependency from file i to file j. Dependencies can be "inherit" or "call" relations identified by *Dependency Extraction* of THProfiler. A diagonal cell means a self-dependency. DRH clusters these 14 files into three layers, i.e., L_0, L_1, and L_2. The uppermost layer L_0 (rows 1 to 6) contains 6 files that denote design rules of this sub-system; L_1 contains row 7 to 8; L_2 (row 9 to 14) includes four mutually independent modules and they are row 9, 10, 11-13, and 14.

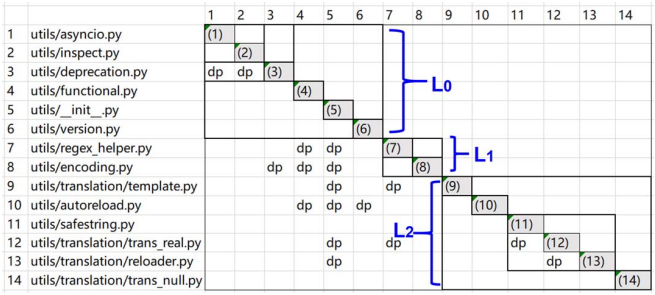

Fig. 2: The hierarchical structure formed by a subset of files in Django

3) Maintenance Measurement: This part computes the maintenance cost of $typed$ and $untyped$ files according to the revision history managed by VCS (Version Control System) like Git.

Revision History Preprocess. This module exports the commit log from a code repository. Each commit record includes the commit ID, the author making this commit, a list of modified files, the IDs of issues fixed by this commit, and the LoC (Lines of Code) of addition and deletion.

Maintenance Cost Measurement. Based on commit records, this module quantifies the effort taken on maintaining source files. Similar to the work of [10], [24], six measures are computed, including *#commit*—the number of commits made to a file; *#changeLoc*—the total lines of changed code of modifying a file; *#author*—the number of developers for maintaining a file; *#issue*—the number of issues that a file gets involved; *#issueCmt*—the number of commits of a file for fixing issues; *#issueLoc*—the total LoC changed to a file

Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on February 15,2023 at 12:34:51 UTC from IEEE Xplore. Restrictions apply.

for fixing issues. The bigger value of these measures indicates the more maintenance cost invested on a file.

III. Study Setup

We collect Python projects that experience type annotation practices and conduct a preliminary analysis during the setup. We will explore three research questions as follows.

RQ1: What are the patterns that non-trivial type-annotation practices follow? This study will figure out and categorize the patterns that complex type-hint practices will follow.

RQ2: Do type-annotated files present different dependency structure? This study will compare the type-annotated files with other files, based on file-level dependency structure.

RQ3: Do type-annotated files incur different maintenance cost? The answer will advance our understanding of the maintainability of type-annotated files when compared with other files.

A. Collection and Subjects

We manually selected Python projects as subjects. The basic selection criteria include: the project 1) is partially type-hinted; 2) is widely used, frequently starred, or downloaded; 3) has a well-managed revision history with commits and issues; 4) follows type annotation syntax supported by Python 3.5 or later versions; 5) uses non-Any types since Any is compatible with every type and has no semantics.

Following these requirements, three contributors of our work initially selected subjects from 105 public projects studied by a recent work [10] and selected frequently starred Python projects in Github. The selection process requires manually inspecting projects, the source code, and community discussions to determine type-related information, such as type-hint manners they adopted and repositories that manage type implementations.

To alleviate the bias to our study, from the projects collected by three contributors, we finally selected subjects with various diversities. 1) *Diverse type hint manners*: we chose the projects, which only adopt $inline$ type annotations, only adopt $stub$ type files, or adopt both $inline$ and $stub$ together. 2) *Diverse domains*: projects cover different domains such as web framework, scientific computing, and type check. 3) *Diverse sizes*: projects can be small-scale (less than 10k LoC), medium-scale (at least 10k LoC), or large-scale (at least 100k LoC). Finally, we collected 19 subjects, as shown in Table I.

In Table I, *Version* is the project version we studied, *#File* counts files, *#LoC* counts lines of code, *TypeManner* lists type-hint manners used by projects, *TypeCodeURL* is the Github repositories holding type implementations of projects, and *Star* denotes the project popularity. We can see that 3 projects manage type implementations in $stub$ files; 9 projects adopt $inline$ type annotations; the remaining 7 projects adopt both $inline$ and $stub$ manners. As listed in *TypeCodeURL*, Django, DRF, and Matplotlib manage type stubs in separate repositories from their codebase repositories.

B. Statistics of Typing Coverage

By employing *Typing Coverage Detection* of our THProfiler, we detected type-annotated entities, then computed typing coverage in terms of LoC, files, functions, and variables. $Coverage(loc)$ is the ratio of LoC annotated with types to the total LoC. The total LoC counts both the source code and stub code in $*.pyi$ (if have). $Coverage(file)$ is the proportion of files with type hints. $Coverage(func)$ is the proportion of functions or methods containing type-annotated declaration signatures. $Coverage(var)$ counts the proportion of type-annotated names or variables. The computation of $Coverage(var)$ excludes parameter variables since they are considered for $Coverage(func)$.

Table II lists typing coverage results. The values less than 100% indicate that partial code entities are assigned with type annotations. The values of $Coverage(file)$ and $Coverage(func)$ are bigger than those of $Coverage(loc)$ and $Coverage(var)$. For example, more than 90% files are type-hinted in DRF, while the coverage values of variables and functions are smaller. Besides, these results reveal a substantial development and maintenance effort invested in type implementations.

C. Statistics of Diverse Type Usage

We used *Diverse Type Extraction* of THProfiler to observe type usages in subjects. We counted the types imported from $typing$ like $Union$ and $List$ since $typing$ module is officially designed to support optional typing.

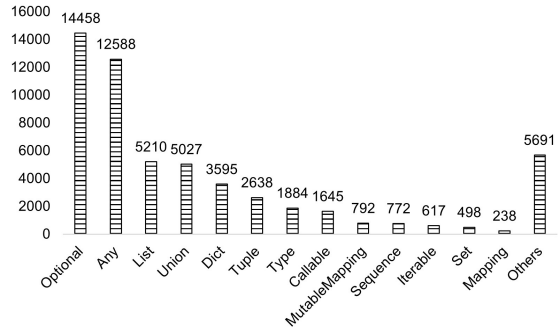

Fig. 3: The usage of diverse types

Figure 3 depicts shows that Any and $Optional$ are most frequently used in subjects. $Optional$ means that the type of a symbol can be a specified type or $None$. The $Optional$ usage indicates code objects are prone to be declared with $None$ by default. Any has no type semantics, meaning that objects annotated by it will escape from static type-checking.

Our study will focus on non-Any annotation practices. If a file only has Any types associated with code symbols, we consider this file is un-annotated. In this case, the type-annotated file coverage as listed in Table III is sightly different with that in Table II for some projects.

Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on February 15,2023 at 12:34:51 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Subjects

Project	Version	File	LoC	TypeManner	TypeCodeURL	Star
Chainer	522e01	1,169	163,349	stub,inline	https://github.com/chainer/chainer	5.6k
Django	v3.1.7	2,531	280,529	stub	https://github.com/typeddjango/django-stubs	56.4k
DRF	v3.12.2	240	38,825	stub	https://github.com/typeddjango/djangorestframework-stubs	20.6k
Elasticsearch	25b50e	225	42,546	stub,inline	https://github.com/elastic/elasticsearch-py	3.2k
gRPC	v1.36.2	280	32,317	inline	https://github.com/grpc/grpc	29.8k
Matplotlib	v3.3.3	1,151	213,369	stub	https://github.com/predictive-analytics-lab/data-science-types	13.3k
Mypy	v0.812	408	100,264	inline	https://github.com/python/mypy	10.4k
Numpy	v1.20.0	896	312,134	stub,inline	https://github.com/numpy/numpy	16.6k
Pandas	v1.2.3	1,350	349,932	inline	https://github.com/pandas-dev/pandas/	29.1k
Prefect	c423a7	538	87,311	inline	https://github.com/PrefectHQ/prefect	6.6.k
Pyproj	dd84df	51	14,699	inline	https://github.com/pyproj4/pyproj	633
PySpark	v3.1.1	181	39,506	stub,inline	https://github.com/apache/spark	29.2k
Pytest	940c6e	236	63,761	inline	https://github.com/pytest-dev/pytest	7.5k
Rasa	a4ee09	474	97,490	inline	https://github.com/RasaHQ/rasa	11.7k
Returns	v0.15.0	332	19,119	stub,inline	https://github.com/dry-python/returns/	1.6k
Sanic	021da3	141	18,943	inline	https://github.com/sanic-org/sanic	15.1k
Scipy	6caa33	787	290,055	stub,inline	https://github.com/scipy/scipy	8.4k
Uvicorn	62825d	64	5,554	inline	https://github.com/encode/uvicorn	4.1k
Werkzeug	08624d	134	29,209	stub,inline	https://github.com/pallets/werkzeug	5.8k

[1] DRF is Django-Rest-Framework.

TABLE II: Typing coverage results

Project	Coverage (loc)	Coverage (file)	Coverage (func)	Coverage (var)	Project	Coverage (loc)	Coverage (file)	Coverage (func)	Coverage (var)
Chainer	0.21%	2.86%	2.63%	0.00%	Django	14.75%	75.89%	50.96%	27.92%
DRF	19.47%	91.67%	68.14%	33.56%	Elasticsearch	43.46%	96.77%	95.88%	44.77%
gRPC	1.46%	8.20%	7.66%	0.00%	Matplotlib	0.03%	4.87%	0.09%	0.01%
Mypy	10.22%	90.00%	99.98%	0.00%	Numpy	2.12%	10.09%	2.07%	3.59%
Pandas	1.26%	20.37%	14.90%	0.57%	Prefect	7.44%	69.78%	90.04%	0.00%
Pyproj	4.91%	61.90%	74.12%	0.00%	PySpark	14.09%	43.55%	55.70%	6.38%
Pytest	11.48%	89.06%	93.11%	0.00%	Rasa	8.45%	79.92%	100.00%	0.00%
Returns	11.82%	80.36%	90.64%	11.32%	Sanic	7.32%	78.05%	52.19%	0.00%
Scipy	0.22%	3.01%	0.67%	0.03%	Uvicorn	8.02%	75.61%	62.59%	0.00%
Werkzeug	11.98%	91.49%	98.39%	1.62%					

TABLE III: $Coverage(file)$ results after excluding Any types

Project	Coverage	Project	Coverage	Project	Coverage
Chainer	2.86%	Django	56.77%	DRF	79.17%
Elasticsearch	60.22%	gRPC	8.20%	Matplotlib	2.21%
Mypy	90.00%	Numpy	7.62%	Pandas	20.37%
Prefect	69.40%	Pyproj	61.90%	PySpark	42.47%
Pytest	89.06%	Rasa	79.92%	Returns	80.36%
Sanic	78.05%	Scipy	3.01%	Uvicorn	73.17%
Werkzeug	91.49%				

IV. Evaluation

A. RQ1: Patterns of Non-trivial Type-annotation Practices

This RQ explores the patterns that non-trivial type annotation practices follow. We first illustrate this motivation. After that, we manually screen subjects to categorize and explain the patterns. We automatically detect these patterns through a static code analysis on ASTs of subjects.

1) Motivation: When we employed *Typing Coverage Detection* of THProfiler in Section III-B, we found an interesting observation in subjects with stub files: besides adding a single type to a symbol, developers sometimes define new entities which were absent in the source files.

Table IV summarizes the stub files, classes, and functions in the stub code, which are directly unmatched with the source code. The results indicate that nine projects (excluding Chainer) have a substantial number of entities newly introduced for type implementations. In Django, 13 stub files ($*.pyi$) have no corresponding source files ($*.py$); 50 classes and 266 functions are additionally defined in stub code.

We assume that this inconsistency may capture non-trivial type-annotation practices. Inspired by this assumption, we inspect the projects with such inconsistency.

2) Categorization Results: Due to the non-negligible inconsistency (as shown in Table IV) between source code and stub code, we manually inspected the subjects with $stub$ files, as listed in Table I. We conducted a qualitative analysis and quantitative analysis supported by *Complex Typing Usage Detection* of THProfiler. Finally, we figured out and categorized six patterns that non-trivial type-hint practices follow.

Category 1 (Typing Compatibility): With the evolution of $typing$ module, $typing$ inevitably provides fresh types that were unavailable in earlier versions. When implementing type annotations in older versions, such fresh types need to be user-defined for version compatibility.

Listing 5: An example for Category 1

```
#numpy/core/function_base.pyi
if sys.version_info >= (3, 8):
  from typing import SupportsIndex
```


Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on February 15,2023 at 12:34:51 UTC from IEEE Xplore. Restrictions apply.

TABLE IV: Summary of stub files, classes and functions that are defined in $*.pyi$ but absent in $*.py$

Project	Absent Stub File	Absent Class	Absent Function
Chainer	0	0	0
Django	13	50	266
DRF	0	23	14
Elasticsearch	0	1	10
Matplotlib	3	5	0
Numpy	3	35	309
PySpark	8	0	96
Returns	0	1	2
Scipy	9	2	4
Werkzeug	0	0	56

* Since values in Chainer are zero, the study of RQ1 will exclude Chainer.

```
else:
  from typing_extensions import Protocol
  class SupportsIndex(Protocol):
    def __index__(self) -> int: ...
```

For example, in Numpy project listed in Listing 5, $SupportsIndex$ is only supported by $typing$ in Python3.8 and later versions. $SupportsIndex$ should be user-defined in older versions. As a result, $SupportsIndex$ is absent in $function\_base.py$ while it is introduced in $function\_base.pyi$.

Category 2 (API Visibility): One module defines an API while its corresponding type-annotated API is declared in the stub file of another module that depends on this API. In this case, type-annotated APIs are directly visible to their dependents.

Listing 6: An example for Category 2

```
#numpy/core/multiarray.py
__all__ = ['empty_like', ...]
def empty_like
(prototype, dtype=None, order=None, subok=None,
    shape=None):
  return (prototype,)

#numpy/core/numeric.pyi
def empty_like(a: _ArrayType, dtype: None = ...,
    order: _OrderKACF = ..., subok: Literal[True] =
    ..., shape: None = ...,) -> _ArrayType: ...
```

In Listing 6 excerpted from Numpy, $empty\_like()$ is defined in $multiarray.py$ module. This function is also included in $\_\_all\_\_$, meaning that it is visible to other modules by "*from multiarray import* $*$". However, type implementation of $empty\_like()$ appears in $numeric.pyi$ instead of $multiarray.pyi$. As a result, $numeric$ can access the semantic types of $empty\_like()$.

Category 3 (Baseclass Presentation): Baseclasses, either extending from $typing.Protocol$[1] or not, are newly introduced into type implementations, making originally invisible interfaces explicit.

Listing 7: An example for Category 3

```
#rest_framework/permissions.py
class AND:
  def __init__(self, op1, op2): ...
  def has_permission(self, request, view): ...
  def has_object_permission(self, request, view, obj
      ): ...
class OR:
  def __init__(self, op1, op2): ...
  def has_permission(self, request, view): ...
  def has_object_permission(self, request, view, obj
      ): ...

#rest_framework-stubs/permissions.pyi
class _SupportsHasPermission(Protocol):
  def has_permission(self, request: Request, view:
      APIView) -> bool: ...
  def has_object_permission(self, request: Request,
      view: APIView, obj: Any) -> bool: ...
class AND(_SupportsHasPermission): ...
class OR(_SupportsHasPermission): ...
```

[1]https://mypy.readthedocs.io/en/stable/protocols.html

TABLE V: The classes present in stub code

Project	Class	All new class	Protocol class	Subclass
Django	1528	50 (3.27%)	10	25
DRF	236	2 (0.85%)	8	9
Elasticsearch	74	1 (1.35%)	0	0
Matplotlib	23	5 (21.74%)	0	2
Numpy	36	35 (97.22%)	27	18
PySpark	483	0 (0%)	0	0
Returns	1	1 (100%)	1	1
Scipy	1619	2 (0.12%)	3	0
Werkzeug	207	0 (0%)	1	0

In Listing 7, classes And and OR have no explicit relationship in $permissions.py$. However, in $permissions.pyi$, both of them explicitly inherit a newly defined baseclass named *_SupportHasPermission*. The baseclass acts as an interface, which was implicitly implemented by AND and OR in $permissions.py$.

Table V summarizes the classes defined in stub files. Considering Django, 1528 classes are type-annotated. 3.27% (i.e., 50) of them are absent in source code but defined for type hints. Among 50 new classes, 10 classes explicitly extend from *typing.Protocol*. 25 classes extend from these new classes. The presence of new base-classes helps build an explicit relationship between base-classes and their sub-classes.

Category 4 (Function Overloading): A function in source code may become overloaded in its type implementations. Overloaded functions have the same name but are declared with different types of parameters or returns. In Listing 8, $smart\_text()$ is overloaded, generating two different definitions in type stub code.

Listing 8: An example for Category 4

```
#django/utils/encodings.py
def smart_text(s, encoding='utf-8', strings_only=
    False, errors='strict'):
   warnings.warn("...")
   return smart_str(s, encoding, strings_only,
       errors)

#django-stubs\utils\encoding.pyi
from typing import TypeVar, overload
@overload
def smart_text(s: _P, encoding: str = ...,
    strings_only: bool = ..., errors: str = ...)
    -> _P: ...
@overload
```


Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on February 15,2023 at 12:34:51 UTC from IEEE Xplore. Restrictions apply.

TABLE VI: Summary of the overloaded functions (i.e., *OFunc*)

Project	$\#OFunc_{all}$	$\#OFunc_2$	$\#OFunc_3$	$Max(OFunc_i, i)$
Django	45	35	9	4
DRF	4	4	0	2
Numpy	33	22	9	4
Pyspark	115	92	14	28
Scipy	18	2	16	3
Werkzeug	18	12	5	4

```
def smart_text(s: _PT, encoding: str = ...,
    strings_only: Literal[True] = ..., errors: str
     = ...) -> _PT: ...
```

Table VI lists function overloading in subjects. For instance, 45 different functions in Django are overloaded in type implementations, as indicated by $\#OFunc_{all}$. $\#OFunc_2 = 35$ means that 35 of 45 functions are overloaded twice, i.e., each function has two different declarations; $\#OFunc_3 = 9$ shows that 9 functions are overloaded with 3 times; the remaining one (i.e., 45-35-9=1) is overloaded with more than 3 times. $Max(OFunc_i,\ i) = 4$ shows that there exists a function overloaded with up to 4 separate declarations.

The common usage of function overloading in Table VI implies a non-trivial effort for "splitting hairs" influenced by different parameters.

Category 5 (Function-assigned Variable): For variables assigned with function objects, type implementations may explicitly declare them as functions. This practice will generate new function definitions in stub files.

Listing 9: An example for Category 5

```
#numpy/core/_internal.py
class _ctypes:
  get_data = data.fget
  get_shape = shape.fget

#numpy/core/_internal.pyi
class _ctypes:
  def get_data(self) -> int: ...
  def get_shape(self) -> Any: ...
```

get_data in Listing 9 is a variable in *_internal.py*, then declared as an explicit function $get\_data(self)$ in *_internal.pyi*. *get_shape* is a similar case.

Category 6 (Typing Extension): Some stub files correspond to extension files like $*.pxd$. $*.pxd$ works like C header files and is provided by Cython, supporting writing C extensions for Python language.

Listing 10: An example for Category 6

```
#numpy/__init__.pxd
ctypedef class numpy.flatiter [object
    PyArrayIterObject, check_size ignore]:...

#numpy/__init__.pyi
class flatiter(Generic[_NdArraySubClass]):...
```

In Listing 10, class $flatiter$ with type annotations is defined in $\_\_init\_\_.pyi$ while its declaration appears in $\_\_init\_\_.pxd$ instead of $\_\_init\_\_.py$.

Pattern Summary: Table VII summarizes the six patterns detected in projects with stub files. First, Function Overloading and Function-assigned Variable are common practices adopted by developers. The numbers of Baseclass Presentation, Typing Compatibility, and Typing Extension are smaller. API Visibility is the most infrequent pattern. Second, projects in diverse domains present different number of patterns. For instance, SciPy and Numpy have the Typing Extension since they integrate with C/C++ extensions for scientific computation.

3) Answering RQ1: We figured out and categorized six kinds of type practices that are non-trivial to conduct. They tightly require recovering and expressing design decisions in long-lasting codebase:

- *Typing Compatability* practice introduces new dependencies into a codebase, due to new definitions for compatibility with older versions of *typing* module.
- *Baseclass Presentation* practice recovers and explicitly expresses design rules, which are manifested as baseclasses or protocols in type implementations.
- *API Visibility* practice makes some APIs directly visible to the API dependents.
- *Function Overloading* practice reconstructs type semantics for complex functions, unambiguously presenting return types based on different parameter types.
- *Function-assigned Variable* practice makes implicit functions explicit.
- *Typing Extension* practice connects code modules across different programming languages in a software system.

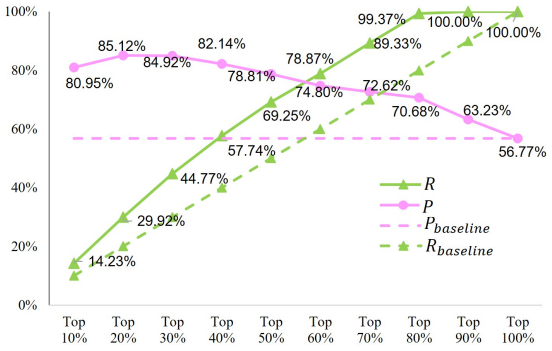

Fig. 4: Using degree centrality to capture type-annotated files in Django

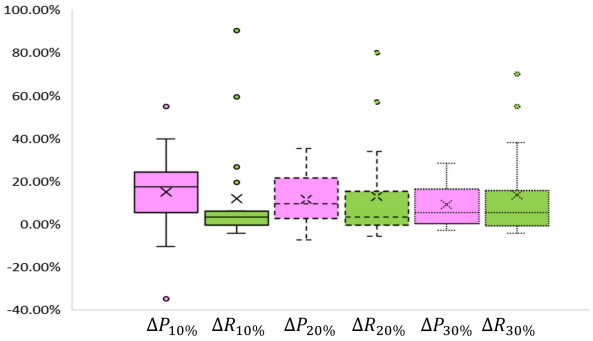

Fig. 5: Performance of degree centrality when capturing type-annotated files

Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on February 15,2023 at 12:34:51 UTC from IEEE Xplore. Restrictions apply.

TABLE VII: The number of patterns detected in subjects with type stubs

Project	Typing Compatibility	API Visibility	Baseclass Presentation		Function Overloading	Typing Extension	Function-assigned Variable
			New Baseclass	New Protocol			
Django	0	2	7	7	45	0	6
DRF	0	0	2	8	4	0	1
Elasticsearch	0	1	0	0	0	0	0
Matplotlib	0	0	2	0	0	0	0
Numpy	12	4	4	4	33	2	4
PySpark	0	0	0	0	115	0	73
Returns	0	0	1	0	0	0	0
Scipy	2	0	0	2	18	8	0
Werkzeug	0	0	0	0	18	0	56
Sum	14	7	16	21	233	10	140

B. RQ2: Characterizing Dependency Structure

This study explores whether type-annotated files present different dependency structures when compared to files without type annotations. Since *Degree Centrality* [25] and *DRH* method [22], [23] have been widely used by prior work [25]–[27], we also employed them to observe software dependency structure.

Degree centrality of a file is the fraction of files it connects to. The higher value of the degree centrality, the more central it is in the dependency structure. *DRH*, as aforementioned, creates a hierarchical structure of a software system. Files at the uppermost layers (L_0) are most influential since they represent design rules of software architecture. We assume that type-annotated files would dominate the file set with higher degree centrality and dominate L_0 layers in the hierarchical structure.

1) Measures: We first employed *Networkx* [28] to compute degree centrality of files in the ADG built by *Dependency Structure Analysis* of THProfiler. We used $Precision$ (P) and $Recall$ (R) to measure the ability of degree centrality when capturing type-annotated files.

$$P = \frac{F_{typed} \cap F_{top}}{F_{top}}, \quad R = \frac{F_{typed} \cap F_{top}}{F_{typed}} \tag{1}$$

where F_{typed} is a set of all type-annotated files in a project. Ranking all files based on degree centrality measurements in a decreasing order, F_{top} is a set of top files such as top 10%, top 20%, ..., and top 100%. Top 100% files include all files in a project. We can obtain ten pairs of P and R results from top 10%, top 20%, ..., to top 100%. Based on them, we will observe whether top ranking files with higher degree centrality are prone to be type-annotated.

Second, we employed *Design Rule Hierarchy (DRH)* of THProfiler to construct hierarchical structure based on the ADG. We computed the proportion of type-annotated files at the layer L_0 (i.e., $Coverage(L_0)$) vs. other layers (i.e., $Coverage(L_{Others})$). If $Coverage(L_0)$ is bigger than $Coverage(L_{Others})$, it would indicate that files, which are manifested as software design rules, are prone to be type-annotated.

2) Results: Using Django as an example, Figure 4 illustrates precision (P) and recall (R) results when using top $10\%, 20\%, ..., 100\%$ files ranked by degree centrality to capture type-hinted files. The red-dotted line labeled with 56.77% is the baseline precision, and the green-dotted line is the baseline recall. Now we explain the baseline performance. The baseline denotes the performance when capturing type-annotated files by a random sample of project files. Considering a random sample of 10% files, type-annotated files should take 56.77% of this sample since 56.77% files (as listed in Table III) are type-annotated in Django. Because only 10% files of a project are randomly sampled, 10% type-annotated files should be captured statistically. We can observe that, the precision and recall results (at top 10%, top 20%, ..., and 100%) present consistent observations: the performance of precision and recall based on degree centrality is bigger than baseline performance.

Figure 5 shows the boxplots of precision and recall improvements when compared with baselines in all subjects. $\Delta P = P - P_{baseline}$ and $\Delta R = R - R_{baseline}$. Due to the page limitation, we only present performance improvement results from top 10% ($\Delta P_{10\%}$, $\Delta R_{10\%}$), top 20% ($\Delta P_{20\%}$, $\Delta R_{20\%}$), to top 30% ($\Delta P_{30\%}$, $\Delta R_{30\%}$). The boxplots indicate that performance improvements of degree centrality are commonly positive despite the existence of several outliers. On average, precision improvements are 14.80%, 11.40%, and 9.18%; recall improvements are 11.73%, 12.79%, and 13.50%, respectively at top 10%, top 20%, and top 30%.

Figure 6 depicts the results of $Coverage(L_0)$ and $Coverage(L_{Others})$. We can observe that 13/19 projects exhibit consistent results with $Coverage(L_0) > Coverage(L_{Other})$. Recall that files in layer L_0 represent design rules of a software system. In Django, 82.97% files are type-annotated at the uppermost layer L_0 while 40.46% files are type-annotated at other layers. The results indicate that files manifested as architectural design rules are more likely to be type-hinted than other files.

3) Answering RQ2: Our results indicate that type-annotated files present different features in the software structure. Concretely, they have a higher value of degree centrality and they reside at the uppermost layer in a hierarchical structure, hence manifesting as design rules of a software system.

C. RQ3: Characterizing Maintenance Cost

This study investigates whether the maintenance cost of type-annotated files differs from other files. Using

Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on February 15,2023 at 12:34:51 UTC from IEEE Xplore. Restrictions apply.

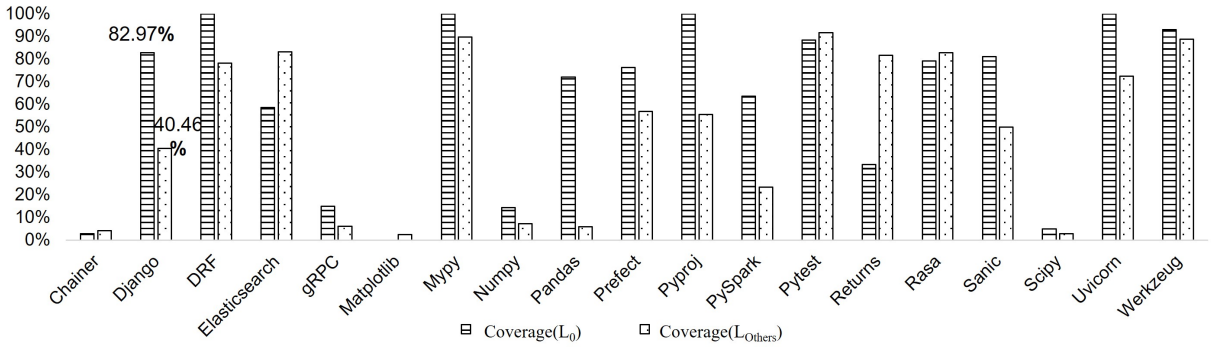

Fig. 6: The proportion of type-annotated files at the uppermost layer (L_0) vs. other layers (L_{Others}) formed by the DRH

TABLE VIII: The performance of maintenance cost measures when capturing type-annotated files in Django project

Subject (Coverage)	Top	$\#author$		$\#commit$		$\#changeLoc$		$\#issue$		$\#issueCmt$		$\#issueLoc$	
		P	R	P	R	P	R	P	R	P	R	P	R
Django	10%	95.24	16.74	95.24	16.74	96.43	16.95	96.43	16.95	96.43	16.95	96.43	16.95
(56.77%)	20%	89.88	31.59	91.07	32.01	91.07	32.01	89.29	31.38	89.88	31.59	92.86	32.64
	30%	89.29	47.07	89.68	47.28	88.89	46.86	84.52	44.56	88.89	46.86	87.70	46.23

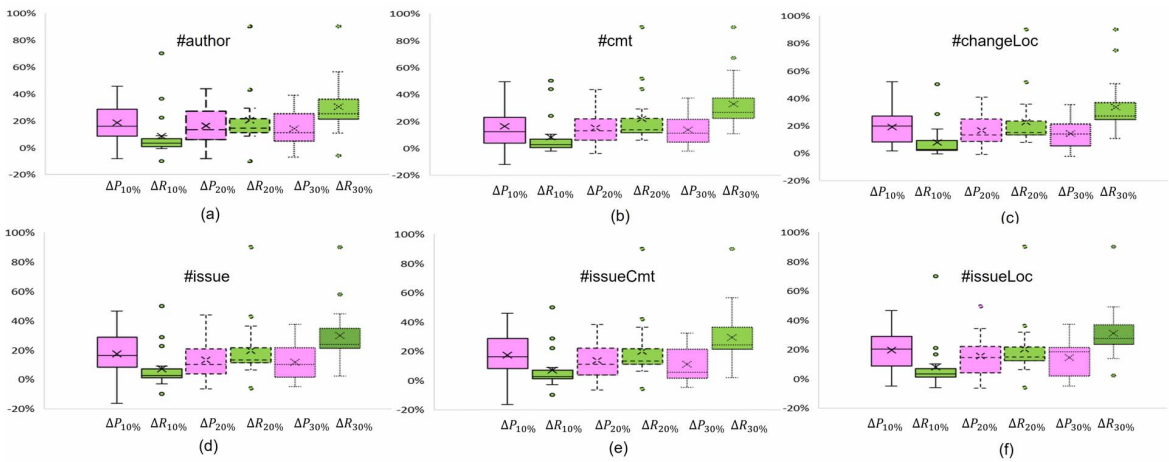

Fig. 7: The performance of maintenance cost measures when capturing type-annotated files

the *Maintenance Measurement* of THProfiler, we computed $\#author$, $\#changeCmt$, $\#changeLoc$, $\#issue$, $\#issueCmt$, $\#issueLoc$ for a source file. These measures have been used by the work of [10], [24] to assess software maintainability. The larger these measurements of a file, the heavier maintenance effort taken on it.

1) Measures: Similar to RQ2, we evaluated the precision and recall when using top files ranked by maintenance cost to capture type-hinted files. The precision measure, recall measure, and baseline values are same with those used in Section IV-B. If values of precision and recall are bigger than those of baseline, it would indicate that type-annotated files present a different feature in terms of maintenance cost.

2) Results: Table VIII lists evaluation results in Django. 56.77% files are type-annotated in total as shown in the first column. The $(\#author, P)$ column indicates that, among the top 10% files with higher maintenance cost 95.24% files have type annotations, greatly larger than the baseline precision (56.77%). As shown in $(\#author, R)$ column, type-annotated files captured by top 10% measurements take 16.74% of all typed files in Django, bigger than the baseline recall, i.e.,10%.

Figure 7 illustrates the boxplots of precision improvements (ΔP) and recall improvements (ΔR) in all projects. The six sub-figures correspond to the results of six maintainability measures. Similar to Section IV-B, we consider the top 10%, top 20%, and top 30% files ranked by maintenance cost measurements.

From Figure 7, we can see that performance improvements are positive when averaged on all 19 projects. The six sub-figures present consistent observations: using heavily maintained files to capture type-annotated files, the precision values and recall values outperform the baseline performance. This

Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on February 15,2023 at 12:34:51 UTC from IEEE Xplore. Restrictions apply.

observation indicates that files with greater maintenance costs are more likely to be type-annotated in priority.

3) Answering RQ3: The results demonstrate the difference of maintenance cost between type-annotated files and other files. This observation indicates that files difficult to maintain could be promising candidates to be type-annotated in priority.

V. Potential Impact

This section will discuss the possible impact of our findings on type annotation practices, by citing developer discussion lists in notable Python projects, including Django [29], Numpy [30], and PySpark [31], which are studied in our work. They have experienced type annotations for about four years, and their developers have been discussing type hint practices.

A. The Consideration of Design Concerns

Python developers have noticed the non-trivialness of type annotation implementations. *"Annotating the original code is more than just adding annotations"*, as discussed by Django contributors. PySpark developers said that *"some parts (of type annotations) are close to trivial, other(s) are rather."* Recent academic works by Ore et al. [7], [8] demonstrate that annotating a single variable consumes about two minutes on average. However, it is still unknown about code-level patterns of non-trivial type annotations, which may guide beginners supplementing types more effectively.

Our results of RQ1 revealed six code patterns of non-trivial type practices. First, these patterns demonstrate the difficulty to include automated approaches for handling type annotations, since type annotation of a code entity sometimes requires introducing additional entities. Second, these patterns highlight that such non-trivial type practices deeply involve retrospecting, recovering, and expressing the design and coding decisions originally made in the codebase development.

We suggest that type practitioners should take design concerns [32]–[35] into account. For example, when type-annotating a group of classes that provide similar behaviors but do not explicitly extend a base-class, we advice that their common interfaces should be extracted as additional definitions to manifest design rules, as inspired by *Baseclass presentation* practice (Listing 7). Learning from *Function Overloading* practice (Listing 8), we recommend overloading the functions that have complex logic due to type diversity of parameters and returns.

B. Candidate Module Recommendation for Type Annotations

Python developers have struggled with which part of code should be type-annotated first. PySpark developers stated, *"It should be decided if annotations should cover only the public API, or internals as well."* Django developers considered, *"Partial type hinting is useful and viable, but not randomly."* Developers agreed that they should *"trade-off between completeness (of the typing coverage) and the cost of maintenance (of type annotations)."* At the same time, they complained about the lack of such guidance–*"missing high level guidance"*, calling for *"a type theory for beginners, both in Python and more generally"*.

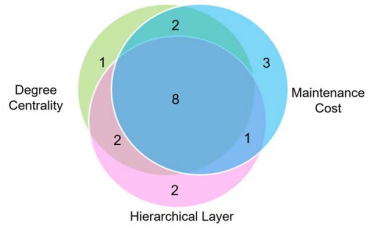

Fig. 8: The Venn diagram of project sets captured by three features

Our findings in RQ2 and RQ3 shed a light on which portion of modules (i.e., Python files) could be type-annotated in priority. We revealed three features of the type-annotated files, i.e., they present a bigger value of degree centrality, they are located at the uppermost layer in a hierarchical structure, and they incur a higher maintenance cost. These features are complementary to each other, as visualized in Figure 8. Among 19 subjects in our study, degree centrality can effectively capture type-annotated files in $\frac{1+2+2+8}{19} = \frac{13}{19}$ projects; the proportion is $\frac{13}{19}$ for the hierarchical layer, and the proportion is $\frac{14}{19}$ for maintenance cost.

For a long-lasting large-scale project, migrating from an un-annotated codebase to type-annotated one is a gradual and expensive process. Our findings imply that Python modules presenting three features could be type-annotated first for a trade-off between the typing coverage and maintenance efforts taken on type annotations. We believe our findings and tools would benefit the projects and developers that plan to experience modern type hint practices. Our study is a preliminary trial at the file level, which will be continued at a finer-grained level like APIs or functions.

VI. Threats to Validity

First, subject collection is non-trivial. To reduce the bias to our study, we followed the collection criteria to select projects with diverse type-hint manners, domains, and sizes, as shown in Section III-A. We will collect more projects for our study in the future.

Second, we manually categorized six patterns in non-trivial type-annotation practices. One threat is that the categories may be insufficient to cover all non-trivial practices. To mitigate this threat, we will continue mining more usable patterns.

Third, different techniques may produce inconsistent observations. To reduce this threat, our study employed well-accepted tools, techniques, and measures. Concretely, we used Understand to extract code dependencies, which is suggested by the industry [36]–[39]. We applied the DRH technique [35], [40], [41] to cluster files hierarchically. Our research employed six maintainability measures [10], [24]. Besides, as shown in Figure 8, the revealed three features are complementary to capture all subjects. One possible reason is that developers in different projects may have different typing decisions. We will analyze them in next work.

Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on February 15,2023 at 12:34:51 UTC from IEEE Xplore. Restrictions apply.

Finally, our study of RQ2 and RQ3 reveals the correlation between three features and type-proneness of a source file, but not the causality. We neither claim how the three features cause a file to be type-annotated, nor the vice versa. Exploring the causality will be our next work.

VII. Related Work

A. Static Typing Systems vs. Dynamic Typing Systems

Much prior work has demonstrated that static typing systems benefit bug detection and software maintenance. Kleinschmager et al. [42] assigned developers with a set of programming tasks, and found that static typing systems can capture type-related errors without program executions. Spiza et al. [43] supported that developers already benefit from type semantics of APIs even without static type checking. Gao et al. [44] manually added annotations to buggy code written in JavaScript and tested whether static typing systems can capture the error on buggy code. Their experiments reported that the static typing systems, Flow [45] and TypeScript [2], can successfully detect a portion of public bugs in revision history. Daly et al. [46] compared Ruby with DRuby, an extension to Ruby with static typing. Their work indicated that DRuby fails in capturing complex errors. Besides discussing the benefits, the work of [47]–[49] concluded that developers who use dynamic languages tend to switch between different files more frequently than developers who use static languages. As shown in their results, the frequent search for different files influences the efficiency of development and maintenance activities.

These studies evaluated the benefits and disadvantages of the usage of static typing feature in dynamically typed languages. Unlike them, our work figured out patterns of non-trivial type annotation implementations and features of type-annotated files.

B. Empirical Study of Type Annotation Practices

Existing work investigated the type annotation usage in development activities. Souza et al. [50] conducted several hypothesis tests to investigate type usage in Groovy language. This work presented that test classes and script files use types less frequently than other files. Another interesting finding is that programmers, who often develop code in an "untyped" language, tend to declare types less often. Groovy is an object-oriented programming language for Java platform. Quite different from Groovy, Python is dynamically typed in nature, thus perhaps leading to different observations. Ore et al. [7] studied 71 programmers using 20 code artifacts in the cyber-physical domain. They assessed the time cost of type annotation supplementation, showing that it takes more than two minutes to annotate a single variable accurately. The considerable time cost demonstrated the complexity and difficulty for developers to assign types to variables. By extending this study [7], a recent work [8] further pointed that developers reason about variable types primarily based on names and operations of identifiers, which points out a direction to improve automated type annotation systems.

Similar to those work, we also studied type annotation practices but presented an evidence that type-annotated files are critical to understanding and maintaining software architecture by revealing three complementary features.

C. Static Type Inference in Dynamic Languages

Static type inference determines the types of program expressions statically without a need to run programs. Early type inference methods such as [51], [52] formalize the type inference problem as the type constraint resolution. The method in [51] first creates a trace graph, from which a set of type constraints are extracted. At last, it computes the least solution of the set of type constraints by least fixed-point derivation. Milojkovic et al. [53], [54] leveraged heuristics such as naming convention to enhance type inference. Recent type inference methods employ machine learning or deep learning techniques. JSNice [55] learns a probabilistic model to predict variable names and variable types for Javascript. DeepTyper [56], [57] transforms the type inference problem into a translation model from a un-annotated code to annotated code. DLTPy [58] is a deep learning type inference solution. It predicts the types in function signatures based on identifier names, comments and return expressions of a function. The work of [59] explored how type inference techniques designed for static language will perform on dynamic languages.

Automated type inference approaches can assist type annotation implementation. Our work also promotes type annotation practices but in a different view. We recommended that such practices should consider architectural concerns due to the existence of non-trivial type annotation implementations.

VIII. Conclusion

Our work is the first to detect non-trivial type annotation practices and reveal possible features of type-annotated files, to the best of our knowledge. We found six type annotation patterns that involve recovering and expressing design concerns made in original codebase development. We also showed that the files, characterized by three complementary features (i.e., presenting higher degree centrality, residing at the uppermost layer of a hierarchical structure, and incurring higher maintenance cost), are prone to be type-hinted.

More projects in dynamic languages are trying to embrace optional typing practice, however, still lacking guidance. During a gradual and expensive process of type-annotating a codebase, we suggest considering the revealed patterns with design concerns and candidate type-annotated file recommendations. A consideration of them would promote a better practice of optional typing.

Acknowledgment

This work was supported by National Key R&D Program of China (2018YFB1004500), National Natural Science Foundation of China (62002280, 61632015, 61772408, U1766215, 61721002, 61833015,61902306), China Postdoctoral Science Foundation (2020M683507, 2019TQ0251, 2020M673439), and Youth Talent Support Plan of Xi'an Association for Science and Technology (095920201303).

Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on February 15,2023 at 12:34:51 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] J. G. Siek and W. Taha, "Gradual typing for functional languages," in IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP. ACM, 2006, pp. 81–92.

[2] TypeScript, "https://www.typescriptlang.org/," 2012–2021.

[3] Ruby, "https://www.ruby-lang.org/en/news/2020/12/25/ruby-3-0-0-released/," 2020–2021.

[4] ——, "https://github.com/ruby/rbs," 2020–2021.

[5] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 91–105. [Online]. Available: https://doi.org/10.1145/3385412.3385997

[6] R. Chatley, A. Donaldson, and A. Mycroft, The Next 7000 Programming Languages. Computing and Software Science, 2019.

[7] J.-P. Ore, S. Elbaum, C. Detweiler, and L. Karkazis, "Assessing the type annotation burden," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 190–201.

[8] J.-P. Ore, C. Detweiler, and S. Elbaum, "An empirical study on type annotations: Accuracy, speed, and suggestion effectiveness," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 30, no. 2, pp. 1–29, 2021.

[9] Spark, "Re: [pyspark] revisiting pyspark type annotations," –2021.

[10] W. Jin, Y. Cai, R. Kazman, G. Zhang, Q. Zheng, and T. Liu, "Exploring the architectural impact of possible dependencies in python software," in 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2020, pp. 758–770.

[11] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in 2015 12th Working IEEE/IFIP Conference on Software Architecture. IEEE, 2015, pp. 51–60.

[12] Python, "https://www.python.org/dev/peps/pep-0484/," –.

[13] W. Jin, D. Zhong, Z. Ding, M. Fan, and T. Liu, "https://github.com/xjtucoderesearch/dataset_typehintpractice," 2021–2021.

[14] Python, "http://mypy-lang.org/," 2014–2021.

[15] Google, "https://google.github.io/pytype/," 2015–2021.

[16] Python, "https://www.python.org/dev/peps/pep-0484/#function-method-overloading," 2015–2021.

[17] ——, "https://www.python.org/dev/peps/pep-0544/," –.

[18] P. docs, "https://docs.python.org/3/glossary.html#term-duck-typing," 2001-2020.

[19] C. Y. Baldwin and K. B. Clark, Design rules: The power of modularity. MIT press, 2000, vol. 1.

[20] S. Understand, "https://scitools.com/," 1996-2020.

[21] W. Jin, Y. Cai, R. Kazman, Q. Zheng, D. Cui, and T. Liu, "Enre: a tool framework for extensible entity relation extraction," in Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings. IEEE Press, 2019, pp. 67–70.

[22] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi, "Design rule hierarchies and parallelism in software development tasks," in Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, 2009, pp. 197–208.

[23] L. Xiao, Y. Cai, and R. Kazman, "Design rule spaces: A new form of architecture insight," in Proceedings of the 36th International Conference on Software Engineering. ACM, 2014, pp. 967–977.

[24] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level: a new metric for architectural maintenance complexity," in Proceedings of the 38th International Conference on Software Engineering. IEEE, 2016, pp. 499–510.

[25] C. R. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," Physical Review E, vol. 68, no. 4, p. 046116, 2003.

[26] M. Harman, D. Binkley, K. Gallagher, N. Gold, and J. Krinke, "Dependence clusters in source code," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 32, no. 1, pp. 1–33, 2009.

[27] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in 2012 34th International Conference on Software Engineering (ICSE). IEEE, 2012, pp. 419–429.

[28] Networkx, "https://networkx.org/," –.

[29] Django, "http://python.6.x6.nabble.com/django-developers-f455249.html," –2021.

[30] Numpy, "http://numpy-discussion.10968.n7.nabble.com," –2021.

[31] Spark, "http://apache-spark-developers-list.1001551.n3.nabble.com," –2021.

[32] L. Bass, P. Clements, and R. Kazman, Software architecture in practice. Addison-Wesley Professional, 2003.

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

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

[35] Y. Cai, L. Xiao, R. Kazman, R. Mo, and Q. Feng, "Design rule spaces: a new model for representing and analyzing software architecture," IEEE Transactions on Software Engineering, 2018.

[36] Lattix, "https://www.sdcsystems.com/tools/lattix-software/lattix-architect/," 2004-2020.

[37] Structure101, "https://structure101.com/," 2004-2020.

[38] R. Mo, W. Snipes, Y. Cai, S. Ramaswamy, R. Kazman, and M. Naedele, "Experiences applying automated architecture analysis tool suites," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, 2018, pp. 779–789.

[39] ArchDia, "https://archdia.com," 2004-2020.

[40] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture anti-patterns: Automatically detectable violations of design principles," IEEE Transactions on Software Engineering, pp. 1–1, 2019.

[41] D. Cui, T. Liu, Y. Cai, Q. Zheng, Q. Feng, W. Jin, J. Guo, and Y. Qu, "Investigating the impact of multiple dependency structures on software defects," in Software Engineering, 2019. ICSE 2019. Proceedings. 41th International Conference on. IEEE, 2019, pp. –.

[42] S. Kleinschmager, R. Robbes, A. Stefik, S. Hanenberg, and E. Tanter, "Do static type systems improve the maintainability of software systems? an empirical study," in 2012 20th IEEE International Conference on Program Comprehension (ICPC). IEEE, 2012, pp. 153–162.

[43] S. Spiza and S. Hanenberg, "Type names without static type checking already improve the usability of apis (as long as the type names are correct) an empirical study," in Proceedings of the 13th international conference on Modularity, 2014, pp. 99–108.

[44] Z. Gao, C. Bird, and E. T. Barr, "To type or not to type: quantifying detectable bugs in javascript," in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 2017, pp. 758–769.

[45] Flow, "https://flow.org/," 2014–2021.

[46] M. T. Daly, V. Sazawal, and J. S. Foster, "Work in progress: an empirical study of static typing in ruby," 2009.

[47] A. Stuchlik and S. Hanenberg, "Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time," in Proceedings of the 7th symposium on Dynamic languages, 2011, pp. 97–106.

[48] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik, "An empirical study of the influence of static type systems on the usability of undocumented software," ACM SIGPLAN Notices, vol. 47, no. 10, pp. 683–702, 2012.

[49] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik, "An empirical study on the impact of static typing on software maintainability," Empirical Software Engineering, vol. 19, no. 5, pp. 1335–1382, 2014.

[50] C. Souza and E. Figueiredo, "How do programmers use optional typing? an empirical study," in Proceedings of the 13th international conference on Modularity, 2014, pp. 109–120.

[51] J. Palsberg, "Object-oriented type inference," in Proc. OOPSLA'91, 1991, pp. 146–161.

[52] J. O. Graver and R. E. Johnson, "A type system for smalltalk," in Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL '90. New York, NY, USA: Association for Computing Machinery, 1989, p. 136–150. [Online]. Available: https://doi.org/10.1145/96709.96722

[53] N. Milojković, C. Béra, M. Ghafari, and O. Nierstrasz, "Inferring types by mining class usage frequency from inline caches," in Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies, 2016, pp. 1–11.

Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on February 15,2023 at 12:34:51 UTC from IEEE Xplore. Restrictions apply.

[54] N. Milojković, "Improving the precision of type inference algorithms with lightweight heuristics," 2017.
[55] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from" big code"," ACM SIGPLAN Notices, vol. 50, no. 1, pp. 111–124, 2015.
[56] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering, 2018, pp. 152–162.
[57] R. S. Malik, J. Patra, and M. Pradel, "Nl2type: inferring javascript function types from natural language information," in Proceedings of the 41st International Conference on Software Engineering. IEEE Press, 2019, pp. 304–315.
[58] C. Boone, N. de Bruin, A. Langerak, and F. Stelmach, "Dltpy: Deep learning type inference of python function signatures using natural language context," arXiv preprint arXiv:1912.00680, 2019.
[59] C. M. Khaled Saifullah, M. Asaduzzaman, and C. K. Roy, "Exploring type inference techniques of dynamically typed languages," in 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 70–80.

Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on February 15,2023 at 12:34:51 UTC from IEEE Xplore. Restrictions apply.