Explorando Como Bibliotecas Python Lançam Exceções ao Longo de Sua Evolução

Allan Gonçalves[1], **Cinthia Nascimento**[1], **Eiji Adachi**[1]

[1]Instituto Metrópole Digital – Universidade Federal do Rio Grande do Norte (UFRN)
Caixa Postal 1524 – 59.078.970 – Natal – RN – Brasil

{allangoncalves, cinthia.katiane}@ufrn.edu.br, eijiadachi@imd.ufrn.br

Resumo. *Para explorar como bibliotecas implementadas em Python estruturam o código responsável por lançar exceções, coletamos 26 bibliotecas-alvo do GitHub e analisamos como seus lançadores de exceção evoluíram ao longo do tempo. Analisamos os tipos de exceção mais comuns usados pelos lançadores e se estes tipos são definidos pelas próprias bibliotecas, pela biblioteca padrão de Python ou por código de terceiros. Os resultados do nosso estudo mostram que há uma preferência por lançar exceções com tipos pré-definidos pela biblioteca padrão de Python, que estes tipos são específicos e que as estratégias adotadas nas versões iniciais das bibliotecas costumam permanecer até as versões finais.*

1. Introdução

O reúso de software é uma meta almejada desde os primórdios da Engenharia de Software [McIlroy et al. 1968]. Por meio do reúso de software, almeja-se reduzir custos e tempo de desenvolvimento de soluções de software. Atualmente, com a consolidação e popularização de bibliotecas de software disponibilizadas em repositórios públicos na Internet, é impensável construir um sistema de software que não faça uso de funcionalidades providas por essas bibliotecas.

Um aspecto importante no projeto de uma biblioteca reutilizável é o seu tratamento de exceções. Tratamento de exceções refere-se à identificação de situações excepcionais que impedem a execução normal de um programa e à implementação de ações que respondem à ocorrência destas situações e que permitem um programa retornar à sua execução normal [Goodenough 1975]. No contexto de uma biblioteca reutilizável, o tratamento de exceções influencia tanto na sua robustez, uma vez que é necessário impedir que exceções causem falhas durante sua execução, e também em sua usabilidade, uma vez que é necessário lançar exceções indicando ao código cliente determinadas condições excepcionais que lhe impedem de prover seus serviços normalmente.

Apesar da importância do tratamento de exceções para o projeto e a implementação de bibliotecas, ainda há pouco conhecimento empírico sobre como bibliotecas implementam o tratamento de exceções. A literatura de tratamento de exceções tem focado principalmente em falhas causadas por defeitos em tratamento de exceções [Ebert et al. 2015, Barbosa et al. 2014], em como tratadores de exceção são implementados [Sena et al. 2016], ou como o código de tratamento de exceções de aplicações evolui ao longo do tempo [Osman et al. 2017]. Não há, no entanto, trabalhos focados em como bibliotecas lançam exceções e como tal estratégia evolui ao longo do tempo.

Este artigo apresenta um estudo exploratório cujo objetivo é caracterizar como o código de tratamento de exceções, em especial o código responsável por lançar as

exceções, é estruturado ao longo da evolução de bibliotecas e de código aberto implementadas na linguagem de programação Python. Analisamos bibliotecas implementadas em Python devido a sua popularidade em ambientes de desenvolvimento, principalmente com o advento de bibliotecas para áreas "quentes", como análise de dados, aprendizado de máquina, dentre outros.[1] Além disso, ainda há poucos estudo empíricos realizados sobre código de tratamento de exceção em sistemas implementados nesta linguagem. Desta forma, este estudo preenche uma lacuna na literatura de tratamento de exceções e de bibliotecas de software. Os resultados do nosso estudo mostram que há uma preferência por lançar exceções com tipos pré-definidos pela biblioteca padrão de Python e que as estratégias adotadas nas versões iniciais das bibliotecas costumam permanecer até as versões finais.

2. Tratamento de Exceções em Python

Python possui um mecanismo de tratamento de exceções nativo baseado em cláusulas `try-except`. A estrutura genérica dessa cláusula segue o modelo:

```
1  try:
2          <try_block>
3  except ExceptionType1 as exc1:
4          <exception_block1>
5  except (ExceptionType2, ExceptionType3, ExceptionType4):
6          <exception_block2>
7  except:
8          <exception_block3>
9  else:
10         <else_block>
11 finally:
12         <finally_block>
```

A cláusula `try` delimita um conjunto de instruções que se deseja proteger da ocorrência de exceções. A uma cláusula `try` podem estar associados um ou mais tratadores, que em Python são definidos como blocos `except`. Cada bloco `except` declara como argumento um tipo de exceção que serve como filtro para capturar exceções: qualquer exceção que seja subtipo do argumento do bloco `except` será capturada por esse bloco. Em Python, também é possível declarar um bloco `except` com uma lista de exceções, como mostrado na linha cinco do exemplo anterior. A instrução `raise` é usada para indicar a ocorrência de um erro, lançando uma instância de exceção que for passada em seu argumento. Em alguns casos, quando não se quer manipular a exceção no bloco `except` onde ela foi capturada, a exceção pode ser capturada e relançada para ser tratada posteriormente por outro método.

Exceções em Python são representadas como objetos de classes que herdam de `BaseException`. Python também permite que desenvolvedores definam tipos de exceção específicos para suas aplicações. Para isto, há o tipo `Exception` e é considerada uma boa prática da linguagem usar tal tipo, ou um de seus subtipos, como ponto de extensão para os tipos de exceção definidos pelos usuários. Desta forma, os desenvolvedores podem aumentar o vocabulário de tipos de suas aplicações, representando erros específicos com tipos de exceção mais adequados do que os providos pela linguagem.

[1]Python aparece como a linguagem mais popular conforme o índice https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017 e a quarta mais popular conforme o índice https://www.tiobe.com/tiobe-index

3. Configuração do Estudo Exploratório

Questões de Pesquisa e Coleta dos Dados. Para caracterizar como o código de tratamento de exceções, em especial o código responsável por lançar as exceções, é estruturado ao longo da evolução de bibliotecas de código aberto implementadas na linguagem de programação Python, nós investigamos as seguintes questões de pesquisa:

QP1: Quais categorias de tipos de exceção são mais usadas em bibliotecas implementadas em Python?
QP2: Quais os tipos de exceção são mais usados em bibliotecas implementadas em Python?

Na primeira questão de pesquisa, investigamos quais categorias de tipos de exceção são mais utilizadas nos lançadores de exceção das biblioteca analisadas. Definimos como *"categoria de tipo de exceção"* a origem onde o tipo de exceção é definida. Consideramos as seguintes origens: (i) *App* – abrange os tipos de exceção que são definidos explicitamente pela própria equipe de desenvolvimento da biblioteca analisada, (ii) *Python* – abrange os tipos de exceção que são definidos pela biblioteca padrão da linguagem Python; e (iii) *Outros* – abrange os tipos de exceção que não se encaixam nas duas categorias anteriores; são tipos de exceção definidos por código reutilizado de terceiros (outras bibliotecas) no contexto das bibliotecas analisadas. Ao investigarmos tal questão de pesquisa, nós pretendemos observar se existe uma preferência em definir tipos de exceção para serem lançados pelas bibliotecas analisadas, ou se estas tendem a lançar exceções com tipos definidos pela própria linguagem, ou por terceiros. Já na segunda questão de pesquisa, nós investigamos quais os tipos que são mais utilizados nos lançadores de exceção das biblioteca analisadas. Investigamos assim se existem tipos de exceção específicos que são comuns a bibliotecas de diferentes propósitos.

Para nos auxiliar a responder as questões de pesquisa propostas, nós coletamos métricas que quantificam diretamente quais os tipos que são utilizados nos lançadores de exceção, quantas ocorrências de cada tipo ocorre e a qual categoria cada tipo pertence. Para isto, implementamos uma ferramenta de análise estática baseada no módulo *Abstract Syntax Tree – ast* que é nativo da linguagem Python. Tal módulo provê uma interface para um analisador sintático de código fonte Python, provendo uma estrutura de árvores de sintaxe abstrata que implementam o padrão de projeto *Visitor* e que permite a extração de informações do código fonte através de análise estática. Desta forma, implementamos um analisador estático que visita as estruturas dos lançadores de exceção – instruções *raise* – e registra o tipo de exceção lançado.

Após registrados os tipos de exceção lançados no contexto das bibliotecas analisadas, passamos para o passo de categorizar os tipos. Para isto, realizamos o seguinte procedimento. Inicialmente, compilamos manualmente todos os tipos de exceção definidos pela biblioteca padrão da linguagem Python.[2] Em seguida, nossa ferramenta analisa o código das bibliotecas-alvo para identificar definições de classes que, por sua vez, são analisadas para identificar aquelas que estendem algum tipo de exceção. Caso exista uma classe estendendo um tipo de exceção, então o tipo definido por esta classe é classificado na categoria *App*. Os outros tipos identificados no código fonte e que não foram classificados nas categorias *Python* ou *App* são classificados na categoria *Outros*.

[2]Esta listagem pode ser encontrada em CTPS://docs.python.org/2/library/exceptions.html

Por fim, para compreender como o código que lança exceções nas evolui ao longo do tempo, coletamos as métricas na primeira e na última versão disponível no repositório de cada biblioteca. Embora esta seja uma análise simplificada da evolução do código dos lançadores de exceção, ela é suficiente para uma caracterização inicial de como as decisões tomadas nas versões iniciais comportam-se ao longo do tempo.

População e Seleção da Amostra. A população alvo do nosso estudo é formada por bibliotecas de código aberto implementadas em Python. Para a realização deste estudo, selecionamos uma amostra dessas bibliotecas que estão disponíveis na plataforma de hospedagem de código fonte GitHub. Atualmente, o GitHub é a maior plataforma de hospedagem de código aberto além de prover uma API de acesso aos projetos de código aberto hospedados na plataforma.

Com base nas recomendações dadas por Kalliamvakou et al. [Kalliamvakou et al. 2014], definimos os seguintes critérios para selecionarmos nossa amostra dentre os projetos disponíveis no GitHub: (i) o projeto deve ser uma biblioteca reutilizável implementada em Python; (ii) o projeto deve ser popular, evitando assim projetos de pouca relevância prática; (iii) o projeto deve ter um histórico de evolução relevante, de modo a observarmos bibliotecas que já atingiram um estágio de maturidade mínimo; e (iv) o projeto deve ser ativo, de modo a observarmos bibliotecas que ainda estão sendo mantidas e evoluídas.

A partir da API do GitHub, nós listamos os 1000 projetos implementados em Python com maior número de *seguidores* (do inglês *watchers*). O número de *seguidores* foi utilizado neste estudo como um indicador da popularidade dos projetos disponíveis no GitHub. Em seguida, aplicamos um filtro para selecionar apenas projetos com um histórico de evolução relevante. Para isto, filtramos apenas os projetos com pelo menos 30 *releases* publicamente disponíveis. De acordo com a documentação oficial do GitHub, toda *release* cadastrada deve estar associada a uma *tag*. Como a funcionalidade de cadastrar *releases* foi implementada apenas em 2013, consideramos *releases* todas as *tags* cadastradas pelos desenvolvedores em seus respectivos repositórios.

Após esta filtragem, restaram 263 projetos. Destes projetos restantes, analisamos manualmente a descrição presente no *read-me* ou na *wiki* de cada um e selecionamos apenas aqueles que explicitamente se denominavam como uma biblioteca (procuramos pelo termo *library*). Dos 263 projetos, restaram 38 projetos. Destes projetos restantes, novamente, analisamos manualmente cada um para descartarmos *forks* de projetos-base. Foram descartados por este critério outros 12 projetos, restando uma amostra contendo 26 projetos de bibliotecas de código aberto implementadas em Python. Esta amostra compreende bibliotecas de diferentes domínios, como bibliotecas para computação simbólica (SymPy), aprendizado de máquina (Scikit-Learn), construção de gráficos (Matplotlib), computação paralela (Dask), dentre outros domínios. A listagem das bibliotecas analisadas é apresentada na Tabela 1.

4. Resultado e Discussão

A Tabela 1 apresenta os dados coletados que nos auxilia a responder nossa primeira questão de pesquisa. Cada linha da tabela apresenta uma das bibliotecas analisadas, agrupando do lado esquerdo os dados coletados para a sua versão inicial (primeira *release* disponível publicamente no GitHub) e do lado direito os dados para sua versão

Tabela 1. Evolução dos Lançadores e suas Categorias de Tipos

Nome	Versão Inicial				Versão Atual			
	App	Python	Outros	Total	App	Python	Outros	Total
Sympy	107	275	40	422	644	2252	15	2911
Scipy	0	212	303	515	42	2210	78	2330
Pandas	5	139	1	145	162	1664	43	1869
Theano	76	797	23	896	148	1554	23	1725
Scikit-Learn	52	77	1	130	35	1072	69	1176
Matplotlib	96	612	43	751	66	928	12	1006
Moto	1	115	0	116	592	209	33	834
Dask	0	3	1	4	15	528	6	549
Scapy	0	0	0	0	128	134	245	507
Cryptography	27	32	0	59	117	384	0	501
Keras	0	16	0	16	0	484	0	484
Bokeh	9	35	0	44	52	317	64	433
Scikit-image	0	15	0	15	3	384	9	396
Gensim	0	34	0	34	2	266	18	286
Docker-py	0	9	2	11	101	52	60	213
Hypothesis	2	4	0	6	135	51	2	188
TensorLayer	0	40	0	40	0	179	0	179
GitPython	4	3	1	8	29	129	19	177
PyEthereum	0	34	1	35	52	107	2	161
Pika	19	5	4	28	37	84	2	123
Psutil	0	1	0	1	53	64	3	120
Requests	16	13	2	31	30	23	0	53
Snips NLU	0	55	0	55	8	39	6	53
spaCy	0	3	0	3	0	49	1	50
Dedupe	0	34	0	34	8	28	0	36
ChatterBot	0	0	0	0	0	0	27	27
Total	414	2526	422	3399	2459	13191	737	16387

final (última *release* disponível publicamente no GitHub). As colunas representam o número de lançadores por categoria de tipo: *App*, *Python* e *Outros*, além da soma total de lançadores em cada versão.

Ao analisar os dados apresentados na Tabela 1, é perceptível que entre as versões inicial e atual de todas as bibliotecas há aumento no número total de lançadores de exceção. Isto é esperado, pois a medida que as bibliotecas evoluem, novas funcionalidades são implementadas e novas condições excepcionais são identificadas, justificando o aumento no número de lançadores.

Analisando a estratégia de lançamento de exceções na versão inicial das bibliotecas, percebem-se os seguintes fatos: 20 das 26 bibliotecas preferem lançar exceções fornecidas pelo Python, 3 das 26 preferem lançar suas próprias exceções, 1 das 26 prefere lançar exceções de terceiros e 2 das 26 bibliotecas não lançam qualquer exceção. Em uma visão geral sobre os lançadores implementados nas primeiras versões das bibliotecas analisadas, tem-se que 12% de todos os lançadores usam exceções definidas pela própria aplicação, 75% contém exceções definidas pela linguagem e 13% contém exceções definidas por aplicações de terceiros. Ou seja, nas primeiras versões das bibliotecas, 88% dos lançadores usam exceções prontas, sejam estas providas pela linguagem de programação ou por código de terceiros.

Analisando a estratégia de lançamento de exceções na versão atual das bibliotecas, observa-se fatos similares aos observados nas versões iniciais. A predileção da origem das exceções está estruturada da seguinte forma: 4 das 26 bibliotecas favorecem suas próprias exceções, 20 das 26 optam por exceções fornecidas pela linguagem, e 2 das

26 usam exceções de outras fontes. Observa-se ainda que na versão atual todas bibliotecas lançam alguma exceção. Ao estudarmos os lançadores, percebemos que 15% contém exceções definidas por suas respectivas bibliotecas, 80% favorecem exceções entregues pelo Python e 5% dos lançadores usam exceções definidas por terceiros. Ou seja, nas versões atuais das bibliotecas, 85% dos lançadores usam exceções prontas que são providas pela linguagem ou por terceiros. Comparando com as versões iniciais, há um pequeno aumento na porcentagem de lançadores que usam exceções definidas pela própria aplicação (aumento de 12% para 15%).

Ao analisarmos cada linha da tabela, podemos analisar se as bibliotecas analisadas alteram suas estratégias de lançamento de exceções no período de evolução analisado. Nesta análise, percebe-se que 6 bibliotecas alteraram sua estratégia de lançamento no que se refere à preferência de categoria de tipos das exceções lançadas. Os projetos Moto, Docker-Py e Hypothesis inicialmente privilegiavam o uso de exceções definidas pela linguagem, mas passaram a priorizar suas próprias exceções. Em um comportamento inverso, durante a versão inicial das bibliotecas GitPython e Pika observava-se um uso maior de suas próprias exceções, mas nas versões atuais passaram a usar mais exceções cedidas por Python. Cabe salientar que nestas duas observações houve aumento no número de exceções em todas categorias de tipos usadas nessas bibliotecas; a mudança ocorreu apenas na preferência (i.e., na categoria que representa a maior porcentagem do total). Por fim, a biblioteca Scipy, que inicialmente utilizava em maior número exceções de terceiros, acabou alterando para exceções da biblioteca padrão de Python. Neste caso específico, houve uma redução no número de lançadores com exceções de terceiros acompanhado do aumento de lançadores nas outras categorias. Houve, portanto, a mudança do tipo da exceção usada em determinados lançadores.

Através destas observações, nota-se que, em grande parte, as bibliotecas escritas em Python priorizam exceções definidas pela própria linguagem e que ao longo da evolução tendem a não alterar a preferência adotada na versão inicial, visto que das 20 bibliotecas que iniciaram seus desenvolvimentos utilizando em maior número exceções definidas pelo Python, apenas três realizaram alterações (Moto, Docker-Py e Hypothesis).

As Tabelas 2 e 3 apresentam os dados que nos auxiliam a responder a segunda questão de pesquisa, apresentando os tipos de exceções mais recorrentes na fase inicial e atual das bibliotecas, respectivamente. As tabelas apresentam o número total de lançadores em todas as bibliotecas analisadas (coluna *#Lan.*), bem como o número de bibliotecas em que um determinado tipo foi usado (coluna *#Bib.*) e o nível que um determinado tipo de exceção está na árvore de tipos de exceções (coluna *Nível*). Neste caso, um valor maior do nível define um tipo mais específico e um valor menor define um tipo mais genérico. Quanto mais específico um tipo de exceção é, mais específica é a preocupação em detalhar qual tipo de erro foi encontrado.

Todos os tipos de exceção mais comumente usados nas versões iniciais das bibliotecas, aqueles apresentados na Tabela 2, são definidos pela biblioteca padrão de Python. Nós analisamos o quão específicos são esses tipos, isto é, o quão distante esses tipos estão do tipo de exceção base de Python, o tipo `BaseException`. Dentre os tipos mais comuns, o tipo `Exception`, que é o quarto mais comum nas versões iniciais, é o único que herda diretamente de `BaseException`. Durante a fase inicial das bibliotecas, as exceções mais recorrentes foram encontradas em 2514 lançadores, e, dentre esses, cerca

Tabela 2. Exceções Iniciais

Nome	#Lan.	#Bib.	Nível
ValueError	919	19	4
NotImplementedError	563	14	5
TypeError	509	11	4
Exception	254	12	2
RuntimeError	134	11	4
KeyError	47	9	5
ImportError	30	10	4
IndexError	26	6	5
IOError	13	6	5
AttributeError	19	4	4

Tabela 3. Exceções Atuais

Nome	#Lan.	#Bib.	Nível
ValueError	7567	25	4
TypeError	2108	23	4
NotImplementedError	1582	22	5
RuntimeError	477	20	4
Exception	409	17	2
AttributeError	173	19	4
ImportError	173	17	4
IndexError	161	17	5
KeyError	136	19	5
AssertionError	188	12	4

de 10% (254 lançadores) utilizam o tipo `Exception` que é um tipo bastante genérico. Já os tipos `ValueError`, `TypeError`, `ImportError`, `AttributeError` e `RuntimeError` estão três níveis abaixo do tipo `BaseException` enquanto os tipos `KeyError`, `NotImplementedError`, `IOError` e `IndexError` estão quatro níveis abaixo do tipo `BaseException`. Além disso, aproximadamente 90% das ocorrências estão divididas em quatro diferentes tipos: `ValueError`, `NotImplementedError`, `TypeError` e `Exception`. Entretanto, apenas dois desses tipos (`ValueError` e `NotImplementedError`) encontram-se em mais da metade das bibliotecas analisadas.

Fato similar foi observado na versão atual das bibliotecas: apenas um tipo comum nas versões iniciais não apareceu nos mais comuns das versões atuais. A diferença entre os tipos mais recorrentes durante as diferentes versões das bibliotecas foi a troca de `IOError` para `AssertionError`. Outrossim, observa-se que o número de lançadores que se relacionavam com as exceções contidas nas tabelas cresceu em aproximadamente 416% durante as duas fases. Posto isso, pode-se dizer que os tipos preferidos no início estenderam-se até a fase atual das bibliotecas, quando apenas `AssertionError` não se propagou em mais da metade das bibliotecas. Embora tenhamos observado algumas mudanças nas estratégias em termos de categorias de tipos, como discutido anteriormente, em relação aos tipos mais comuns não observamos mudanças nas estratégias de lançamento. Por fim, o fato de 97% das ocorrências conterem exceções que pertencem a níveis mais fundos na árvore de tipos (81% em três níveis abaixo e 16% quatro níveis abaixo do tipo `BaseException`) mostra que as bibliotecas preocuparam-se minimamente em lançar exceções com tipos mais específicos, provavelmente buscando tipos que indiquem mais claramente aos seus usuários quais erros ocorreram.

5. Trabalhos Relacionados

Os estudos de Ebert et al. [Ebert et al. 2015] e de Barbosa et al. [Barbosa et al. 2014] analisaram sistemas de software *open-source* e categorizaram falhas ocasionadas por defeitos no código de tratamento de exceções. Algumas destas categorias estão relacionadas ao lançamento de exceções, em particular ao lançamento de exceções com tipos muito genéricos. No nosso estudo, observamos que algumas das bibliotecas analisadas adotam a prática de lançar exceções muito genéricas desde as versões iniciais, mantendo esta prática até suas versões atuais. Similarmente ao nosso estudo, Sena et al. [Sena et al. 2016] realizaram um estudo empírico do tratamento de exceção em bibliotecas. Todavia, estes focaram em analisar os fluxos excepcionais, categorizar as estratégias de tratamento de

exceção e antipadrões de tratamento de exceções existentes bibliotecas Java, enquanto o presente estudo analisa as estratégias adotadas no lançamento de exceções em bibliotecas com código fonte Python. Já Osman et al. [Osman et al. 2017] apresentaram uma comparação entre a evolução do tratamento de exceções em bibliotecas e aplicações. Os autores também categorizaram as exceções e efetuaram uma análise sobre os lançadores em bibliotecas Java. Assim como em nosso estudo, estes autores também observaram que bibliotecas Java parecem dar preferência ao uso de tipos de exceção definidos pela própria linguagem ao invés de tipos próprios definidos pela biblioteca.

6. Conclusão

Nosso estudo observou que há uma preferência por tipos de exceção definidos pela biblioteca padrão de Python e que os tipos mais comumente usados estão em níveis mais profundos da árvore de tipos de exceção. Isto pode sugerir que o projeto da árvore de tipos da linguagem Python apresenta um bom vocabulário de erros para as necessidades das bibliotecas. Por outro lado, observamos que algumas bibliotecas ainda lançam exceções com tipos muito genéricos e pouco informativos sobre o erro encontrado, prática já reportada na literatura como sendo causa comum de falhas em sistemas de software. Isto pode indicar que o tratamento de exceções, especialmente o lançamento de exceções, ainda não é apropriado no projeto de algumas bibliotecas. Por fim, observamos que as preferências quanto ao lançamento de exceções adotadas na versão inicial não costumam mudar ao longo da evolução do projeto. Isto pode indicar que as decisões tomadas nas versões iniciais são difíceis de serem modificadas ao longo da evolução e, portanto, o tratamento de exceção deve ser bem projetado e implementado desde as versões iniciais das bibliotecas.

Referências

Barbosa, E. A., Garcia, A., and Barbosa, S. D. J. (2014). Categorizing Faults in Exception Handling: A Study of Open Source Projects. In *Proceedings of the XXVIII Brazilian Symposium on Software Engineering (SBES'14)*.

Ebert, F., Castor, F., and Serebrenik, A. (2015). An exploratory study on exception handling bugs in java programs. *Journal of Systems and Software*, 106:82–101.

Goodenough, J. B. (1975). Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*, pages 92–101.

McIlroy, M. D., Buxton, J., Naur, P., and Randell, B. (1968). Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering (ICSE'68)*, pages 88–98.

Osman, H., Chiş, A., Corrodi, C., Ghafari, M., and Nierstrasz, O. (2017). Exception evolution in long-lived java systems. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR'17)*, pages 302–311.

Sena, D., Coelho, R., Kulesza, U., and Bonifácio, R. (2016). Understanding the exception handling strategies of java libraries: An empirical study. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR'16)*, pages 212–222.