UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DANIEL VENTURINI

UM ESTUDO EMPÍRICO SOBRE BREAKING CHANGES NO ECOSSISTEMA DO NPM

CAMPO MOURÃO

2020

DANIEL VENTURINI

UM ESTUDO EMPÍRICO SOBRE BREAKING CHANGES NO ECOSSISTEMA DO NPM

An Empirical Study of Breaking Changes in the NPM Ecosystem

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação do Curso de Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Orientador: Ivanilton Polato

Coorientador: Igor Scaliante Wiese

CAMPO MOURÃO

2020

DANIEL VENTURINI

UM ESTUDO EMPÍRICO SOBRE BREAKING CHANGES NO ECOSSISTEMA DO NPM

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação do Curso de Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 27/novembro/2020

Ivanilton Polato
Doutorado
UTFPR

Igor Scaliante Wiese
Doutorado
UTFPR

Marco Aurélio Graciotto Silva
Doutorado
UTFPR

Reginaldo Ré
Doutorado
UTFPR

CAMPO MOURÃO

2020

RESUMO

Contexto: Os pacotes hospedados no npm criam uma rede de dependências, na qual um pacote *cliente* utiliza algum recurso de um pacote *provedor*. Por vezes, os provedores introduzem *breaking changes*, que são alterações que podem causar defeitos nos clientes. Essas alterações deveriam ser publicadas apenas no nível *major* do Versionamento Semântico, mas quando são introduzidas nos níveis *minor* ou *patch*, podem causar defeitos inesperados nos clientes. **Objetivo:** Este trabalho propõe um estudo sobre *breaking changes* em níveis *minor* e *patch* no npm. Os objetivos são: (QP1) mensurar a ocorrência de *breaking changes*, (QP2) categorizar as modificações mais usuais causadoras de *breaking changes* e (QP3) analisar como os clientes se recuperam das *breaking changes*. **Método:** De uma amostra de clientes do npm foram restauradas as *releases* e instalada a última versão dos provedores que o cliente aceitava no momento da *release*. Em seguida, foram executados os *scripts* `npm install/test`. Para todas as *releases* que resultaram em erros, foram analisados os códigos e os repositórios dos clientes e dos provedores para verificar se o erro foi causado por um provedor, ou seja, uma *breaking change*. **Resultados:** (QP1) Ao todo, 13,9% das *releases* dos clientes são impactadas por *breaking changes* que estão ocorrendo ano após ano, e 54,9% das *releases* dos provedores com *breaking changes* possuem mais *commits* que as suas demais *releases*. (QP2) A maioria das *breaking changes* são corrigidas pelos provedores em níveis *patch* e são documentadas em 78,1% dos casos, principalmente em *issues*, o que faz com que a correção ocorra 3,3 vezes mais rápida. (QP3) Enquanto os provedores indiretos são os que mais introduzem as *breaking changes*, os clientes as corrigem em 39,1% dos casos e preferem realizar um *upgrade* na versão do provedor, mas sem alterar o *range*. **Conclusões:** As *breaking changes* realmente ocorrem em *releases minor* e *patch*. Enquanto estudos anteriores focaram em alterações de *APIs*, esse estudo utilizou os testes dos clientes para encontrar qualquer tipo de *breaking change*. Finalmente, apresentamos várias sugestões para os desenvolvedores melhorarem suas interações com o ecossistema do npm.

Palavras-chaves: npm. *Breaking change*. Versionamento Semântico. Gerenciamento de dependências.

ABSTRACT

Context: Packages hosted on npm create a dependency network, where *client* packages use some recource from *provider* packages. Occasionally, providers introduce breaking changes, which are changes that may cause defects on clients. These changes should be only introduced in *major* level of Semantic Versioning, but when introduced in *minor* or *patch* levels, these may cause issues on clients. **Objective:** This work proposes a study about breaking changes in minor and patch levels on npm. Our objectives are: (RQ1) to measure the breaking change occurrence, (RQ2) to show the most common breaking change types, and (RQ3) to analyze how clients recover themselves. **Method:** From a sample of clients from npm, we restored the releases and installed the latest version of providers that the client accepted in the release timestamp. Following, we executed the `npm install/test` scripts. All releases that raised an error were analyzed, and the client and providers code and repositories was verified to check whether the error was raised by a provider, characterizing a breaking change. **Results:** (RQ1) Altogether, 13.9% of client releases are impacted by breaking changes, which have happened year after year and 54.9% of provider releases with breaking changes have more commits than their other releases. (RQ2) The majority of breaking changes are fixed by providers in patch levels and are documented in 78.1% of cases, mainly on issues, causing the fix to be 3.3 time faster. (RQ3) While indirect providers are the ones that introduces the majority of breaking changes, clients fix these in 39.1% of cases and they prefer to do an upgrade on the provider's version without changing the range. **Conclusions:** Breaking changes do really happen in minor and patch releases. Previous studies focused only on API breaking changes, while this study used clients' tests to find any types of breaking changes. We presented several suggestions to developers to improve their interaction with the npm ecosystem.

Keywords: npm. Breaking change. Semantic Version. Dependency management.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O *Node Package Manager* (npm) é um gerenciador de pacotes para a linguagem JavaScript e um registro no qual os pacotes são publicados e armazenados. Lançado em 2009, seu principal objetivo é facilitar o compartilhamento de código escrito para o Node.js. Atualmente, o registro do npm ocupa a posição de maior repositório para uma dada linguagem, com mais de 1,4 milhões de pacotes.[1] O npm é um dos fatores que impulsionaram o JavaScript a se tornar um ecossistema completo, com pacotes, *frameworks*, aplicativos *web* entre outros (CHATZIDIMITRIOU et al., 2018). Além disso, 97% dos aplicativos *web* são oriundos do npm, de acordo com o próprio npm.[2]

O npm contém o maior número de dependências entre os diversos gerenciadores de pacotes para uma linguagem de programação (DECAN et al., 2016). Nesse cenário, o termo *provedor* refere-se ao pacote que fornece recursos para os seus *clientes*, que usufruem desses recursos. Como há muitas dependências no npm, quando algum provedor contém algum tipo de defeito, um grande número de clientes pode ser afetado. Para dimensionar a cadeia de dependências que os pacotes no npm possuem, considere o seguinte exemplo real. Quando o pacote left-pad foi removido do npm, essa remoção afetou milhares de outros pacotes em apenas 2,5 horas, gerando um erro quando o comando `npm install` era executado nesses pacotes. Até mesmo pacotes renomados como o babel e o atom foram atingidos e, por sua vez, propagaram esse defeito para seus clientes. Assim, problemas entre provedores e clientes realmente ocorrem no ecossistema do npm devido a essa cadeia de dependências entre os pacotes e por isso esse ecossistema foi escolhido como estudo de caso.

O npm utiliza um sistema de versionamento chamado *Versionamento Semântico*, que é composto basicamente de três dígitos (níveis) separados por um ponto (.) e que devem ser incrementados de acordo com as alterações que o desenvolvedor introduz em cada *release*. Esse sistema de versionamento permite que o desenvolvedor publique *releases* com *breaking changes*, novas funcionalidades e correções de erros, respectivamente nos níveis *major*, *minor* e *patch*, separadas uma das outras. As *breaking changes* são alterações nos pacotes provedores que os tornam incompatíveis com as suas versões anteriores (MØLLER; TORP, 2019), fazendo com que os provedores tenham um comportamento inesperado para os clientes. Os provedores podem introduzir *breaking changes* para evoluir seus códigos e fazer alterações importantes, entre outros motivos. Essas *breaking changes* deveriam ser introduzidas apenas em *releases* com nível *major* do Versionamento Semântico. Porém, o problema ocorre quando uma *breaking change* é introduzida em uma *release* de nível *minor* ou *patch*, ou seja, em *releases* que deveriam conter apenas novas funcionalidades e correções de erros, respectivamente. Quando isso ocorre, erros são introduzidos nos clientes, que são afetados por *breaking changes*.

[1] http://www.modulecounts.com
[2] https://blog.npmjs.org/post/180868064080

Este trabalho tem por objetivo realizar um estudo empírico sobre as *breaking changes* que foram introduzidas erroneamente em níveis *minor* e *patch* no ecossistema do npm. Além de quantificar as *breaking changes*, este trabalho apresenta uma categorização das *breaking changes* e uma análise acerca de como os clientes se recuperam das *breaking changes*. Para isso, foi utilizada uma amostra representativa dos clientes no npm e foram executados os testes de cada uma das *releases* em que haviam alterações nas versões dos provedores e *script* de testes válidos. Em seguida, foi realizada uma analise manual em cada *release* que resultou em erro para confirmar se o erro se tratava de uma *breaking change*. Por fim, foram analisados os repositórios dos provedores e dos clientes para recolher informações pertinentes a cada caso de *breaking change*, tais como *issues* e *pull-requests*.

Esse trabalho contribui como um estudo empírico sobre *breaking changes* no ecossistema do npm. Os resultados e as discussões apontam os principais casos de *breaking changes* e os métodos que os provedores podem utilizar para mitigar a introdução de *breaking changes*. Ainda, descrevemos técnicas que os clientes podem utilizar para gerenciar seus provedores, descobrirem e recuperarem-se com eficiência das *breaking changes*.

O Capítulo 2 apresenta o referencial teórico, incluindo os trabalhos relacionados. No Capítulo 3 são descritos três exemplos reais de *breaking changes* que foram identificados em nosso estudo. O Capítulo 4, além de detalhar a coleta dos dados que foram utilizados nessa pesquisa, contém a motivação e o método para cada uma das questões de pesquisa. Já o Capítulo 5 apresenta os resultados das questões de pesquisa e o Capítulo 6 discute esses resultados. No Capítulo 7 há a descrição dos fatores de ameças à validade interna, externa e de construção deste trabalho. Por fim, o Capítulo 8 apresenta as conclusões deste trabalho.

2 REFERENCIAL TEÓRICO

Neste capítulo apresentamos o referencial teórico utilizado neste trabalho. A Seção 2.1 discorre sobre *pacote*, *release* e *versão*. A Seção 2.2 distingue os termos *provedor* e *cliente*. A Seção 2.3 explica o que é *Versionamento Semântico* e o *SemVer* e como eles são utilizados no ecossistema do npm. A Seção 2.4 conceitua as *breaking changes* e a Seção 2.5 apresenta os trabalhos relacionados.

2.1 Pacote, *Release* e Versão

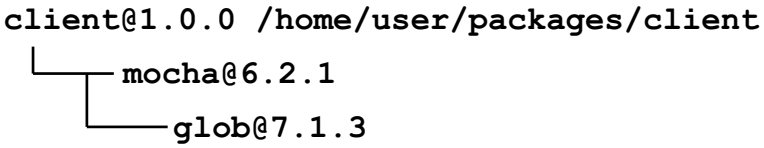
Neste trabalho, o termo *pacote* refere-se a um projeto de *software* hospedado no npm. Cada pacote contêm seu nome, seus arquivos e suas versões. O termo *release* designa o estado de um pacote em um determinado instante. Uma *release* é denotada por um conjunto de arquivos distintos das demais *releases*. O termo *versão* é utilizado para especificar uma *release*. Uma *versão* é uma *string* no padrão do *Versionamento Semântico* onde uma única versão identifica uma única *release* e é utilizada pelo npm no arquivo *package.json*. O arquivo *package.json* é o arquivo de configuração das *releases* que contém todas as informações pertinentes ao pacote naquela *release*, tais como a lista de arquivos, o endereço do repositório, mas principalmente, sua versão única e todos os provedores com as respectivas versões que o cliente aceita. Ainda, o *package.json* contém *scripts* que podem ser executados. Dois deles são os *scripts install* e *test*. Os provedores são instalados por meio do comando `npm install` e os testes executados através do comando `npm test`.

2.2 Provedor e Cliente

Provedor é aquele pacote que fornece recursos ao cliente, ou seja, contém interfaces públicas e recursos para acesso às suas funcionalidades. O termo *provedor* também pode ser conhecido como *biblioteca*. Por exemplo, um cliente executa o comando `npm install mocha`. Após a execução desse comando, o npm registra o mocha no *package.json* como uma dependência. Assim, o pacote mocha é um *provedor direto*, pois foi instalado diretamente pelo cliente. Entretanto, os provedores do mocha também são instalados, mas de forma indireta pelo npm. Estes outros pacotes são chamados de *provedores indiretos* do cliente, pois não dependem que o cliente instale-os manualmente com o comando `npm install`. A Figura 2.1 mostra que o pacote mocha é um provedor direto do client, pois o client requer o mocha para executar. Por sua vez, o pacote glob é um provedor direto do mocha e um provedor indireto do pacote client.

O *cliente* é aquele que está acessando as interfaces públicas do provedor. Quando uma *breaking change* é introduzida pelo provedor – direto ou indireto –, essa *breaking change* pode se manifestar em qualquer pacote nessa cadeia de dependências, principalmente no cliente. O cliente possui a responsabilidade de atualizar a versão de seus provedores no *package.json* quando esses publicam uma *release* com correções, enquanto o provedor possui a responsabilidade de indicar o

Figura 2.1. Pacote mocha como provedor direto e glob como indireto do client.



Fonte: Autoria própria

nível de compatibilidade que sua nova *release* está introduzindo (RAEMAEKERS et al., 2014), ou seja, se sua *release* é ou não retro-compatível, através do Versionamento Semântico. O cliente atualiza a versão do seu provedor alterando manualmente essa versão no *package.json* ou instalando outra versão do provedor, fazendo com que o npm atualize a versão no *package.json*. Já o provedor deve conhecer as regras do Versionamento Semântico e cuidar que as suas *releases* sejam versionadas corretamente.

2.3 Versionamento Semântico e *SemVer*

O Versionamento Semântico[1] é um padrão para versionamento de *releases* de um projeto que considera o tipo de alteração introduzida na *release*. As regras do Versionamento Semântico foram idealizadas por Tom Preston-Werner, criador do GitHub, que incentiva todos os desenvolvedores à utilizarem esse padrão, uma vez que suas regras são baseadas em práticas comuns já utilizadas em projetos (RAEMAEKERS et al., 2014). Embora sejam apenas um conjunto de regras e não são impostas pelo npm, são largamente utilizadas e incentivadas (DECAN; MENS, 2021). Uma *string* de versão no padrão do Versionamento Semântico possui os níveis *<MAJOR>.<MINOR>.<PATCH>*[2] que devem ser incrementados quando o desenvolvedor publicar uma *release*, de acordo com os seguintes critérios:

- *MAJOR*: deve ser incrementado quando a *release* introduz *breaking changes*; desobriga a retrocompatibilidade.
- *MINOR*: incrementado quando forem adicionadas melhorias/novas funcionalidades que mantenham a compatibilidade com as *releases* anteriores, ou seja, alterações retrocompatíveis; e
- *PATCH*: deve ser incrementado quando a *release* contém correção de *bugs*.

Dessa maneira, se um projeto contém a sua última *release* versionada como *2.1.0*, por exemplo, o seu nível *major* é o 2; o *minor*, 1; e o *patch*, 0. Ao publicar uma nova *release*, se essa contiver uma *breaking change*, então deverá ser publicada com a versão *3.0.0*; se for introduzida uma nova funcionalidade, *2.2.0*; se houver uma correção de *bugs*, *2.1.1*; e se introduzir os três tipos de alterações, o nível *major* deverá ser incrementado, uma vez que uma *breaking change* é a alteração que mais impacta o cliente.

[1] https://semver.org
[2] A *string* pode especificar versões *beta, alpha, pre, releases* candidatas, entre outros, tal como *x.y.z-beta.0*

O *SemVer* é uma *string* de versionamento que especifica um intervalo de versões, ou um *range*. Com o *SemVer* é possível especificar quais são as *releases* que o cliente aceita do seu provedor. Há vários padrões de *range*[3] especificados pelo *SemVer*, mas os mais comuns, utilizados pelo npm, são:

- *X-Ranges (*)*: este *range* especifica para o npm que o cliente aceita qualquer nova *release* do provedor, até mesmo as *releases* com *breaking changes*;
- *Caret Ranges (^)*: este é o *range* mais comum e o padrão do npm. Com o *Caret Range*, o cliente especifica que o npm só deve descarregar novas *releases* do provedor que contenham novas funcionalidades ou/e correções de erros, mas que não contenham *breaking changes*, ou seja, o cliente aceita todas as *releases* das quais foram atualizadas os níveis *patch* ou *minor*;
- *Tilde Ranges (~)*: neste *range*, o cliente especifica para o npm que somente as *releases patch* do provedor são aceitas, ou seja, somente as *releases* que contêm alguma correção de erros.

O npm utiliza o padrão *SemVer* no arquivo *package.json* pois, através do *SemVer*, o cliente pode decidir o quão flexível ele é ao receber as futuras *releases* dos provedores (DECAN; MENS, 2021). Por exemplo, ao executar o `npm install express`, para instalar o provedor express por exemplo, o npm – além de descarregar esse provedor – irá registrar no *package.json* o nome do provedor com sua versão atual em modo *caret range*, de acordo com a Figura 2.2.

Figura 2.2. Modo como o npm registra no *package.json* a versão de uma dependência.

```
"author": "",
"license": "ISC",
"dependencies": {
  "express": "^4.17.1"
}
```

Fonte: Autoria própria

Com a informação do *range* do provedor no *package.json*, o npm sempre irá resolver o *range* e descarregar as *releases* mais recentes que são aceitas pelo *range SemVer* especificado pelo cliente. *Resolver o range* significa recuperar todas as versões do provedor e selecionar a mais recente que é aceita pela *string SemVer* especificada pelo cliente. Por padrão, o npm especifica o *Caret Range*, mas o cliente pode especificar outro *range* manualmente no *package.json* ou pode utilizar a opção `–save-exact` junto ao comando `npm install` para especificar a versão sem o *range*, tornando-se um *range steady*, e fazendo com que o npm sempre instale essa versão específica.

2.4 *Breaking Change*

Uma *breaking change* é uma alteração no provedor que produz defeitos nos clientes (RAEMAEKERS et al., 2014). As *breaking changes* surgem quando o provedor, que previamente era executado com

[3] https://github.com/npm/node-semver#ranges

sucesso pelo cliente, publica uma *release* que causa um erro ou um comportamento inesperado no cliente. Durante o desenvolvimento de *software*, os provedores precisam introduzir *breaking changes*, pois quando só há *releases* retrocompatíveis, o *software* perde muitas oportunidades de evolução (BOGART et al., 2015). Desta maneira, as *breaking changes* são importantes para a evolução de um *software*, e podem ser consideradas sinônimos de evolução. Exemplo disso é o Node.js, que publica uma *release* incrementando o nível *major* a cada 6 meses.[4] Introduzir *breaking changes* em níveis *major* permite que os *softwares* evoluam sem manter-se presos à versões anteriores.

Para evitar que os impactos de uma *breaking change* afetem os clientes, os provedores devem publicar suas *releases* com *breaking changes* sempre incrementando o nível *major* da sua nova versão, seguindo as regras do Versionamento Semântico. Dessa maneira, os clientes de versões prévias que especificaram os provedores com o *range caret* – *range* especificado por padrão pelo npm – ou o *range tilde*, não são afetados por uma *breaking change*, pois eles apenas aceitam *releases* com novas funcionalidades e correções de erros – *releases minor* e *patch*, respectivamente. Entretanto, uma *breaking change* pode ser introduzida erroneamente quando são atualizados os níveis *minor* ou *patch*. Dessa maneira, o problema das *breaking changes* está no fato dos provedores introduzirem em *releases minor* ou *patch*, o que não é esperado pelos clientes, resultando em defeitos nos clientes.

2.4.1 *Breaking change* induzida

Além de alterações que resultam em *breaking changes*, há algumas alterações que são geradas nos provedores mas que, nessa pesquisa, não serão consideradas como *breaking changes*, uma vez que esses casos não significam que o provedor contenha um erro ou não fazem parte do ecossistema do npm. Os erros do tipo *breaking change induzido* são erros que não são gerados por uma nova *release* do provedor e nem podem ser corrigidas em uma próxima *release* do provedor. Essas alterações são:

- Exclusão de uma *release*/provedor do npm: pelas regras do npm, uma *release* só pode ser removida até 72 horas após ter sido publicada.[5] Entretanto, quando uma *release* é removida do npm e o cliente especificou aquela *release*, isso gera um erro no *script install*. Assim, o erro é causado pelo npm que não consegue encontrar a *release*, e não pelo provedor. No caso do provedor ter sido removido do npm a situação é a mesma: um provedor só pode ser removido após 72 horas. Entretanto, anterior ao acontecimento do left-pad, os pacotes – e *releases* também – podiam ser removidos do npm em qualquer circunstância. Por isso, quando um erro foi causado pela falta de um pacote que foi removido do npm, não foi considerado como uma *breaking change*;
- Alterações em serviços externos: os provedores podem fazer uso de sistemas externos, tais como acesso à *APIs* de sites e sistemas, e recuperar dados. Entretanto, ao longo do tempo, naturalmente, essas *APIs* podem alterar seus dados, o que gera inconsistências em seus clientes.

[4] https://github.com/nodejs/node#release-types
[5] https://docs.npmjs.com/cli/unpublish#description

Mas esse tipo de erro não é considerado como uma *breaking change*, uma vez que esse caso não faz parte do ecossistema do npm e o provedor não tem controle sobre esses dados.

2.5 Trabalhos Relacionados

Esta Seção apresenta os trabalhos relacionados ao tema *breaking change*. Na Subseção 2.5.1 estão os trabalhos relacionados a estudos no ecossistema do npm. A Subseção 2.5.2 descreve os trabalhos sobre *breaking changes* em outros ecossistemas.

2.5.1 *Breaking changes* no npm

Bogart et al. (2015) apresentam um estudo sobre a estabilidade das dependências no ecossistema do npm e do CRAN. Os autores entrevistaram setes mantenedores de pacotes e questionaram os entrevistados sobre os motivas das mudanças em seus *softwares*. Os desenvolvedores afirmaram que o Versionamento Semântico deve ser usado corretamente para evitar problemas com atualizações de dependências. Em nosso trabalho, descobrimos que os provedores introduzem e publicam alterações em níveis incorretos do Versionamento Semântico. Essas publicações, usando um versionamento incorreto, introduzem erros nos pacotes clientes, encerrando a execução.

O trabalho de Kraaijeveld (2017) apresenta uma abordagem para detectar *breaking changes* em *APIs* em três pacotes provedores. O autor selecionou mil pacotes clientes para cada pacote provedor e realizou um *parser* nos códigos dos clientes e dos provedores. Assim, foi possível descobrir, entre duas versões de um provedor, se houve alteração em uma de suas *APIs*. O *parser* no código do cliente foi realizado para verificar se algum cliente utilizava alguma *API* do provedor que havia sido alterada. Ao todo, foram identificados de 9,8% a 25,8% de *releases* dos clientes haviam sido impactadas com alguma *breaking change* nas *APIs* dos provedores. Em nosso trabalho nós identificamos que 13,9% de *releases* dos pacotes clientes foram impactadas por algum tipo de *breaking change*. Porém, o método do autor é melhor do que o nosso para identificar *breaking changes* em *APIs*, uma vez que, em nosso trabalho, alterações nos argumentos de *APIs* podem não ter sido detectadas (Ver Seção 7).

Mezzetti et al. (2018) apresentam uma técnica chamada *teste de regressão de tipo* para verificar alteração no tipo dos objetos retornados de uma *API*. Essa técnica consiste em executar uma chamada para uma *API*, salvar o tipo do objeto retornado, executar novamente a *API*, salvar o tipo do objeto retornado nessa última execução e comparar ambos os tipos. A *release* do provedor é alterada entre as duas execuções para verificar se o tipo do objeto foi alterado entre as duas *releases* do provedor. Os autores escolheram os 12 provedores mais populares do npm para realizar as análises. Para todas as versões *major* de um provedor, foi executado as *APIs* de todas as *releases minor* e *patch* pertencentes àquela versão *major*. Então foi aplicada o teste de regressão nessas versões *minor* e *patch*. Foi constatado regressão no tipo dos objetos em 9,4% das *releases*. Nosso trabalho aborda qualquer tipo de *breaking changes* e nossa análise é tanto no pacote cliente quanto no pacote provedor, com um total de 13,9% de *releases* dos clientes afetadas com *breaking changes*.

O trabalho de Mujahid et al. (2020) foca na detecção de *releases* de provedores que induzem um erro na execução dos clientes. Os autores analisaram um total de *290k* pacotes do npm. Eles verificaram nos repositórios dos clientes todos os *commits* que alteraram o *package.json*. Para os provedores especificados com versões *range*, foi resolvido a versão do provedor com base na data do *commit*. Entre dois *commits*, os autores verificaram se a versão do provedor foi retrocedida, ou seja, se foi realizado um *downgrade*. Quando houve o *downgrade*, a versão atual e a prévia do provedor foi registrada como uma possível *release* com *breaking change*. Ou seja, se o pacote cliente realizou um *downgrade*, pode ter sido realizado por causa de uma *breaking change*. Finalmente, dez casos aleatórios de *downgrade* foram selecionados e os testes dos clientes foram executados com a *release* atual e a prévia do provedor e foi verificado se houve um erro na execução do cliente entre a versão atual e prévia. Enquanto os autores analisam a cobertura dos testes, nossa metodologia não faz essa análise. De maneira similar, usamos dados oriundos do npm e resolvemos as versões dos provedores, executando os testes dos clientes sempre que pelo menos uma versão dos provedores foi alterada.

Nosso trabalho se diferencia dos demais que estudaram *breaking changes* no ecossistema do npm pelo modo como detectamos as *breaking changes*, através do código do pacote cliente. Nosso trabalho encontrou vários tipos de *breaking changes*, uma vez que não focamos apenas em alterações de *API*. Por fim, todos os casos que geraram erros foram analisados manualmente para confirmar se eram ou não casos reais de *breaking changes*.

2.5.2 *Breaking changes* em outros ecossistemas

Brito (2018) estudaram 400 provedores no repositório do Maven por um total de 116 dias. Os pacotes provedores foram escolhidos pela popularidade medida no GitHub e, durante aquele período, os autores analisaram os *commits* que introduziram uma *breaking change* em uma *API*. Quando uma *breaking change* em algum *commit* foi detectada, eles questionaram o autor do *commit* sobre o motivo para introduzir aquela *breaking change*. Os autores observaram que 33,3% das *breaking changes* são inseridas para implementar novas funcionalidades e 29,8% para simplificar as *APIs*. Em nosso trabalho, focamos em encontrar as *breaking changes* através dos testes dos pacotes clientes.

O trabalho de Foo (2018) apresenta um estudo sobre *breaking changes* em *APIs* nos ecossistemas Maven, PyPI e RubyGems. O objetivo do trabalho de Foo (2018) é analisar uma ferramenta desenvolvida para esse propósito, que realiza automaticamente uma análise estática e eficiente para verificar se a atualização para uma versão de um provedor introduz uma *breaking change* de *API*. A ferramenta detecta *breaking changes* computando um *diff* entre o código de duas *releases* de um pacote provedor. Foram descobertas *breaking changes* em 26% dos pacotes provedores e a ferramenta conseguiu sugerir atualizações automaticamente para 10% dos pacotes. Nosso método vai além de descobrir *breaking changes* apenas em *APIs* mostrando que os pacotes clientes são impactados por *breaking changes*.

3 EXEMPLOS MOTIVACIONAIS

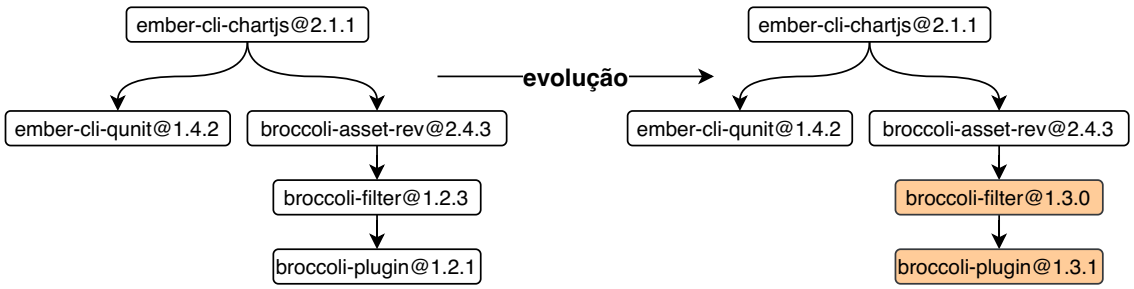
Breaking changes ocorrem de diversas maneiras no ecossistema do npm. Pelo fato do Javascript ser uma linguagem de tipagem dinâmica, o cliente pode ser afetado por muitos erros inesperados, o que comprometerá sua execução. Considere os seguintes casos de *breaking changes* que foram detectados nesse trabalho. O primeiro caso ocorreu no provedor optipng. Ao atualizar da *release 0.1.1* para *0.2.0*, o provedor introduziu uma alteração no nome de uma de suas *APIs* renomeando `OptiPng.getBinaryPath` para `OptiPng.getBinPath`,[1] conforme o *diff* do Código 3.1. Após publicar a *release 0.2.0*, todos os clientes que tinham acesso àquela *API* e atualizaram o optipng ou receberam a atualização automaticamente devido ao *range* passaram a receber um erro de execução, pois a *API* prévia não estava mais disponível. Os clientes que utilizavam o *range caret* apenas esperavam receber atualizações retrocompatíveis, mas receberam uma *breaking change*. Um erro de alteração do nome de *APIs* é um erro facilmente detectável, mas a *breaking change* introduzida em optipng@0.2.0 foi corrigida depois de 34 dias para a *release 0.3.1*.[2] No período que essa *breaking change* ficou sem correção, o provedor optipng recebeu *2,8k downloads* do npm.

```
- OptiPng.getBinaryPath = memoizeAsync(function (cb) {
+ OptiPng.getBinPath = memoizeAsync(function getBinPathAsync(cb) {
```

Código 3.1. *Breaking change* de alteração de API no pacote optipng.

O segundo exemplo de *breaking changes* ocorreu no pacote cliente ember-cli-chartjs. A *release 2.1.1* do cliente ember-cli-chartjs especifica dois provedores: ember-cli-qunit@^1.2.2 e broccoli-asset-rev@^2.4.2. O provedor broccoli-asset-rev especifica como seu provedor o pacote broccoli-filter@^1.2.2, que por sua vez especifica como seu provedor o pacote broccoli-plugin@^1.0.0. Essa árvore de dependências pode ser visualizada na Figura 3.1.

Figura 3.1. Evolução da árvore de dependências para o pacote cliente ember-cli-chartjs de quando foi publicada a *release 2.1.1* (esquerda) para quando foi executada nesse trabalho (direita).



Fonte: Autoria própria

[1] https://github.com/papandreou/node-optipng/commit/920216
[2] https://github.com/papandreou/node-optipng/commit/a155f2b

A *release* ember-cli-chartjs@2.1.1 foi publicada em Junho de 2016 e executou os testes com sucesso nas determinadas versões de seus provedores até essa data[3] (árvore esquerda da Figura 3.1). Porém, alguns meses antes, em Novembro de 2015, o provedor ember-cli-qunit publicou a *release 1.0.4* que introduziu uma correção de um erro. Essa *release* alterava o tipo de um objeto que era retornado,[4] como no diff do Código 3.2, e essa *release* executou com sucesso.

```
  // Skip if useLintTree === false.
  if (this.options['ember-cli-qunit'] && ... ) {
-     return tree;
+     // Fakes an empty broccoli tree
+     return { inputTree: tree, rebuild: function() { return []; } };
  }
```

Código 3.2. Alteração no tipo do objeto retornado em ember-cli-qunit@1.0.4.

Aproximadamente três anos depois do ember-cli-qunit@1.0.4, em Agosto de 2018, a versão broccoli-plugin@1.3.1 foi publicada (Árvore de dependências à direita na Figura 3.1) e introduziu uma alteração em uma função de validação. Essa *release 1.3.1* introduziu uma melhoria na função `isPossibleNode` para consertar um erro que permitia a validação de objetos inválidos,[5] como no *diff* do Código 3.3.

```
  function isPossibleNode(node) {
-   return typeof node === 'string' ||
-     (node !== null && typeof node === 'object')
+   var type = typeof node;
+   if (node === null) {
+     return false;
+   } else if (type === 'string') {
    ...
+   } else {
+     return false;
+   }
  }
```

Código 3.3. broccoli-plugin@1.3.1 validation function enhanced.

Logo que broccoli-plugin@1.3.1 foi publicado uma *breaking change* se manifestou devido à alteração de objeto em ember-cli-qunit@1.0.4,[6] que havia sido publicado quase três anos antes. Essa *breaking change* foi introduzida entre os dois provedores e impactou o pacote cliente, resultando em erros nos seus casos de testes. Essa *breaking change* foi consertada em ember-cli-qunit@1.4.3 após 15 dias, quando o objeto foi alterado novamente.[7] Durante os 15 dias em que a *breaking change* não foi consertada, o pacote broccoli-plugin foi descarregado *384k* vezes do npm.

[3] https://travis-ci.org/github/busy-web/ember-cli- chartjs/builds/139575556
[4] https://github.com/ember-cli/ember-cli-qunit/commit/6fdfe7d
[5] https://github.com/broccolijs/broccoli-plugin/commit/3f9a42b
[6] https://github.com/broccolijs/broccoli-merge-trees/issues/65
[7] https://github.com/ember-cli/ember-cli- qunit/commit/59ca6ad

Nesse terceiro exemplo, podemos verificar como que o gerenciamento de dependências pode ser difícil para o pacote cliente. Mesmo quando o cliente se preocupa com as atualizações de seus provedores, alguma *breaking change* pode se manifestar. Por exemplo, o provedor jsdom@16 alterou uma de suas funcionalidades para não gerar mais propriedades compartilhadas entre os objetos[8] – já havia essa função em versões prévias do jsdom. Quando jsdom@16 foi publicado não havia nenhuma incompatibilidade com nenhum de seus provedores. Então, o pacote cliente delp.pt@1.0.35 especificou jsdom com uma versão específica, talvez para evitar possíveis *breaking changes* nas próximas releases do jsdom, ou seja, delp.pt não especificou um *range* de versões para jsdom e os casos de testes executaram com sucesso.

```
- exports.installInterfaces = window => {
+ exports.installInterfaces = (window, globalNames) => {
    // Install generated interface.
    for (const generatedInterface of Object.values(generatedInterfaces)) {
-     generatedInterface.install(window);
+     generatedInterface.install(window, globalNames);
  }
```

Código 3.4. jsdom@16.3 corrige a função `installInterface`

Quase 6 meses depois, foi publicado jsdom@16.3 que corrigiu um erro na função do jsdom@16, adicionando um parâmetro que estava faltando quando eram executados os seus provedores.[9] O *diff* do Código 3.4 contém essa correção. Aproximadamente dois meses após da publicação do jsdom@16.3, um dos provedores do jsdom, o pacote whatwg-url, publicou a *release 8.2*, que introduziu uma funcionalidade que é executada pela função `installInterfaces` do jsdom, a função do Código 3.4, e a execução ocorreu com sucesso. Entretanto, o cliente delp.pt usa o jsdom com um *range* específico e não recebeu a correção publicada em jsdom@16.3. Porém, como jsdom@16 especificou o seu provedor, o whatwg-url, com um *range caret*, o cliente delp.pt recebeu whatwg-url@8.2, que introduziu uma *breaking change* em jsdom@16. O provedor whatwg-url@8.2 requer o segundo parâmetro na função `installInterfaces`, mas esse parâmetro só foi adicionado no jsdom@16.3. Assim, por causa da falta de *range* no cliente e do *range* no jsdom, uma *breaking change*, que já havia sido corrigida oito meses atrás, se manifestou no pacote cliente. Essa *breaking change* ocorreu apenas no nosso estudo porque foi apagado o arquivo *package-lock.json* antes de instalar os provedores (Ver Seção 4.4). No cenário real, o cliente delp.pt utiliza o arquivo *package-lock.json* e a versão do whatwg-url está especificada nesse arquivo para que o *range* não seja aplicado pelo npm e outras versões do whatwg-url não sejam descarregadas. Portanto, o cliente delp.pt não foi impactado por essa *breaking change*.

[8] https://github.com/jsdom/jsdom/blob/master/Changelog.md#1600
[9] https://github.com/jsdom/jsdom/commit/e07aac9#diff-ead812607022ff7c6eca706fca8347ad

4 METODOLOGIA

Este trabalho propõe um estudo sobre as *breaking changes* em *releases minor* e *patch* e seus impactos no ecossistema do npm. Para isso, três questões de pesquisa foram elaboradas para direcionar nosso estudo. Apresentamos a seguir as questões de pesquisa e suas motivações, bem como os procedimentos para a coleta e amostragem do conjunto de dados.

4.1 Questões de Pesquisa

Apresentamos nessa seção as questões que norteiam nossa pesquisa e investigação.

QP1. Com que frequência *breaking changes* impactam os clientes?

Motivação: no ecossistema do npm, uma *release* que contenha um erro pode afetar uma grande quantidade de pacotes, uma vez que a rede de dependências do npm é relativamente densa (DECAN et al., 2016). Para evitar que *breaking changes* se manifestem nos clientes, os provedores introduzem as *breaking changes* em *releases major*, seguindo o padrão do Versionamento Semântico, e os clientes podem utilizar *strings SemVer* para aceitar apenas as versões *minor* e *patch* dos provedores – o padrão do npm. Entretanto, nem sempre o provedor é capaz de distinguir se suas alterações são ou não *breaking changes* (MEZZETTI et al., 2018), e muitas vezes as *breaking changes* são introduzidas sem que os provedores percebam. Estudos anteriores têm abordado *breaking changes* no ecossistema do npm (MUJAHID et al., 2020; MEZZETTI et al., 2018; MØLLER; TORP, 2019; KRAAIJEVELD, 2017), mas não estudaram a frequência e como se manifestam. Nesta QP, serão quantificadas as manifestações das *breaking changes* nos clientes.

Método: Quando os comandos `npm install` e `npm test` resultaram em erro, o nosso objetivo foi distinguir se o erro foi causado pelo provedor, caracterizando uma *breaking change*, ou se foi causado apenas pelo próprio cliente, não sendo uma *breaking change*. A primeira evidência é o *stack trace* gerado pelo npm quando ocorre um erro. O *stack trace* contém o tipo do erro, a localização exata do erro e o fluxo de execução no momento em que o erro ocorreu. Uma das principais informações são os provedores que estavam sendo executados no fluxo de execução. Se houve algum provedor no *stack trace*, provavelmente o erro se tratava de uma *breaking change*. Quando não houve algum provedor no *stack trace*, provavelmente o erro não se tratava de uma *breaking change*, mas ainda sim foram feitos os métodos descritos abaixo para confirmar se um erro era de fato uma *breaking change* ou não.

Para quantificar as *breaking changes*, foi necessário diferenciar entre um erro causado pelo próprio cliente, no qual não houve influência de nenhum provedor, e um erro causado por algum dos

provedores, sendo assim uma *breaking change*. Para realizar esta diferenciação, foram realizadas as seguintes heurísticas:

- **Alterações nos códigos**: foram realizadas algumas alterações nos códigos do cliente e do provedor para analisar o fluxo de execução até gerar o erro. Por exemplo, foram adicionadas chamadas para `console.trace()` para visualizar a pilha de execução até essa chamada. Também, a chamada para `console.log()` foi muito utilizada para verificar o conteúdo das variáveis em tempo de execução e suas tipagens. Isso tudo para verificar como as variáveis estavam se comportando e como estavam sendo alteradas pelos provedores e pelo próprio cliente.
- **Integração contínua ao GitHub:** sistemas de integração contínua[1] são sistemas que integram-se aos repositórios e disponibilizam tarefas automáticas para os desenvolvedores, tal como execução de testes. Essas integrações contínua desempenharam um papel fundamental na análise manual. Se o *status* do teste do *commit* da *release* do cliente nesses sistemas integrados estava como sucesso e em nosso estudo foi identificado como um erro, provavelmente o erro se tratava de uma *breaking change*. Isso pois o *commit* da *release* do cliente no sistema integrado era o mesmo *commit* executado em nosso estudo. Assim, apenas a versão dos provedores poderia ter sido alterada e causado o erro.
- ***Commits* do cliente:** foram analisados manualmente os *commits* do cliente a partir do *commit* da *release* para verificar se o cliente tentou consertar algo em seu código. Se sim, foram realizadas as alterações feitas pelo cliente para verificar se o erro foi consertado e concluir se o erro foi causado apenas pelo pacote cliente ou se foi causado por um dos pacotes provedores. Por exemplo, se um cliente atualizou um provedor e foi impactado por uma *breaking change*, nos próximos *commits* o cliente poderia realizar um *downgrade* na versão do provedor ou realizar uma alteração em seu código para se recuperar da *breaking change*. Os *commits* nomeadas como *"downgrade provider"*, *"fix break change"*, *"Bump tests and dependencies"* indicam que o cliente realizou alguma alteração para, provavelmente, se recuperar da *breaking change*.
- ***Issues/Pull-requests*:** se o erro é uma *breaking change*, outros clientes podem ter sido impactados e provavelmente já foi documentada em uma *issue* ou um *pull-request*. Através dos comentários das *issues*/*pull-requests* foi possível recuperar informações detalhadas sobre o erro, qual provedor introduziu e se foi consertada. *Issues* e *pull-requests* foram muito importantes e permitiram encontrar muitas informações, porque muitas *issues* e *pull-requests* referenciam outras, no mesmo projeto ou em projetos distintos, enquanto os desenvolvedores estão rastreando um erro (ZHANG, 2018).
- ***Releases* precedentes e posteriores do provedor:** essa foi uma etapa muito importante para detectar se um erro era uma *breaking change*. Se um erro é uma *breaking change*, as *releases* precedentes e posteriores do provedor poderiam consertar o erro. Nesse caso, foi desinstalada a *release* atual e instalada uma *release* anterior ou posterior àquela que causou o erro. Por

[1] https://strongloop.com/strongblog/node-js-travis-circle- codeship-compare/

fim, o *script* de teste foi reexecutado. Por exemplo, se um cliente especificou um provedor `p` como `{"p": "1.0.2"}` e esse provedor introduziu uma *breaking change* na *release*, por exemplo, `1.0.4`. Então foram instaladas as releases `p@1.0.2`, `p@1.0.3` e `p@1.0.5` para verificar se alguma dessas *releases* não introduziu ou consertou a *breaking change*. Assim, foi possível confirmar em qual *release* do provedor a *breaking change* foi introduzida/consertada.

Para todos os casos de erro confirmados como *breaking changes*, foram coletadas a data das *releases* dos provedores que introduziram as *breaking changes* para realizar uma análise temporal. O registro do npm está funcional desde 2010 e foi analisada a evolução das *breaking changes* ao longo do tempo.

Para entender quais características as *releases* com *breaking changes* diferem das demais *releases* sem *breaking changes*, foi recuperada, para cada *release* que introduziu a *breaking change*, a quantidade de *commits* que o provedor introduziu em todas as *release* pertencentes ao mesmo nível *major*. Por exemplo, para uma *breaking change* introduzida no provedor p@2.0.3, foi recuperada a quantidade de *commits* introduzida no *range* p@2.x.y, ou seja, p@2.0.0, p@2.0.1, p@2.0.2, p@2.0.3 e assim por diante. Então, foi calculada a mediana dos *commits* introduzidos em cada *release* nesse *range major* para verificar se a *breaking change* na *release* do provedor foi influenciada pela quantidade de *commits*. Entretanto, três provedores foram removidos desta análise pois os seus repositórios são compartilhados com outros pacotes, o que tornou inviável a análise dessa quantidade de *commits* entre duas *releases*, uma vez que seria analisado *commits* dos outros pacotes também. Esses provedores são @types/node, @types/lodash e babel-preset-es2015.

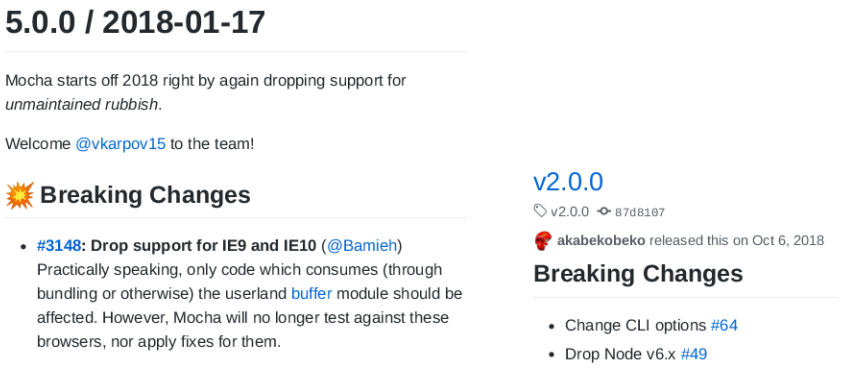
QP2. Como os provedores introduzem *breaking changes* em uma *release*?

Motivação: Pesquisas anteriores apresentam estudos sobre *breaking changes* no ecossistema do npm. Entretanto, pelo fato do *Javascript* ser dinâmico, esses estudos focaram apenas nas alterações de *APIs*, tais como as remoções/renomeações, alterações na lista de parâmetros e alterações no tipo de retorno. Esses estudos foram realizados por Kraaijeveld (2017) e Mezzetti et al. (2018) e não verificaram *breaking changes* além das relacionadas às *APIs*. Porém, podem haver outros tipos de *breaking changes* no ecossistema do npm além das alterações em *API*. Por causa da falta de informação, muitas *breaking changes* introduzidas poderiam ser facilmente evitadas. Por isso, categorizar as *breaking changes* ajudará os desenvolvedores a atentar-se para as *breaking changes* mais comuns, assim produzindo códigos menos favoráveis a ocorrência das mesmas.

Método: O objetivo da análise manual é descobrir o motivo que originou uma *breaking change*, ou seja, qual foi a alteração que o provedor realizou e que causou a *breaking change*, para que seja possível agrupa-las por suas similaridades. Após descobrir em qual versão do provedor a *breaking change* foi introduzida, foi realizada uma análise manual no repositório do provedor para descobrir a exata alteração no código que originou a *breaking change*. As seguintes técnicas foram usadas:

- **Arquivos de alterações:** os arquivos de registros de alterações, comumente nomeados por *CHANGELOG.md* ou *HISTORY.md*, contêm as descrições das principais alterações em cada *releases* do projeto. Uma das informações mais relevantes nestes arquivos são as descrições de *breaking changes*. Por exemplo, a versão *5.0.0* do pacote Mocha contém uma *breaking change* que foi documentada no *CHANGELOG.md*[2] de acordo com a Figura 4.2(a). Outro tipo de documentação equivalente são as *releases-notes*, como pode ser visualizado na Figura 4.2(b) como o pacote wpxml2md documentou *breaking changes* nas suas *releases-notes*.[3]
- **Ferramentas de *diff*:** foram utilizadas ferramentas que realizam o *diff* entre duas *releases* de um pacote. Um *diff* entre duas *releases* exibe todas as alterações que foram realizadas de uma *release* para outra. Com isso, foi verificado o que foi adicionado e removido do código do provedor – até mesmo do cliente – em um determinado intervalo de versões.
- ***Commits* dos provedores:** foram analisadas os *commits* do provedor que introduziram a *breaking change* para verificar exatamente a sua evolução em detalhes. Foram verificados no repositório do provedor os *commits* anterior e posterior ao *commit* da *release* com *breaking change* para verificar exatamente em qual *commit* a *breaking change* foi introduzida.

Figura 4.1. Exemplo de *break changes* documentadas em *changelogs* e *relese-notes*



(a) (b)

Fonte: Autoria própria

Também buscamos em *issues* e *pull-requests* por comentários indicando as causas das *breaking changes*. Após descobrir as alterações que introduziram as *breaking changes*, foram analisadas e agrupadas cada uma dessas alterações em categorias mais genéricas possíveis. Por exemplo, todas as alterações relacionadas com mudança no tipo de variáveis foram agrupadas em uma categoria chamada *Alteração de tipo de objeto*. Também foi analisado o nível do Versionamento Semântico em que a *breaking change* foi introduzida pelo provedor e consertada pelo provedor ou cliente, bem como o local onde as *breaking changes* foram documentadas (*issues/pull-requests/changelogs*).

[2] https://github.com/mochajs/mocha/blob/master/CHANGELOG.md#500–2018-01-17
[3] https://github.com/akabekobeko/npm-wpxml2md/releases/tag/v2.0.0

QP3. Como os clientes se recuperam das *breaking changes*?

Motivação: Uma *breaking change* pode impactar um pacote cliente através de uma atualização *implícita* ou *explícita* de seu pacote provedor. Uma atualização implícita ocorre quando o cliente especificou o seu provedor como um *range* de versões no *package.json*. Então o npm descarrega automaticamente a nova *release* do provedor. Já uma atualização explícita ocorre quando o cliente atualiza manualmente a versão do provedor no *package.json* e o npm descarrega a nova versão especificada pelo cliente. Após uma *breaking change*, o cliente pode se recuperar realizando uma alteração no seu código, aguardando uma nova *release* do provedor que venha a consertar a *breaking change* ou o cliente pode realizar um *downgrade/upgrade* na versão do provedor.

As *breaking changes* podem ser introduzidas pelo provedor *direto* ou *indireto*, uma vez que os clientes dependem de poucos provedores diretos mas dependem de muitos provedores indiretos (DECAN et al., 2019). Mesmo quando o cliente tem poucos provedores diretos, muitos provedores indiretos podem propagar *breaking changes*. Quando uma *breaking change* se manifesta nos pacotes clientes, esses devem se recuperar uma vez que eles precisam executar sem erros e, também, eles podem ser provedores de outros pacotes nessa árvore de dependências. Portanto, uma *breaking change* pode ser continuamente propagada enquanto não for consertada por nenhum dos pacotes. Até mesmo quando as *breaking changes* podem ser consertadas atualizando-se para uma nova versão do provedor, os pacotes clientes precisam resolver manualmente as incompatibilidades que ainda existem (FOO, 2018). Então, entender o comportamento da manifestação das *breaking changes* pode ajudar os desenvolvedores a compreenderem quais são as maneiras mais efetivas e rápidas para se recuperar das *breaking changes*.

Método: Todas as informações usadas para responder esta questão de pesquisa foram recuperadas dos repositórios dos pacotes clientes. Foram procuradas nesses repositórios informações sobre o erro e como os cliente se recuperaram da *breaking change*. As seguintes informações foram analisadas:

- ***Commits:*** foram analisados manualmente os próximos *commits* no repositório do cliente a partir da data da *release* que contém a *breaking change*. Foram analisados principalmente os *commits* que alteraram o *package.json* para verificar se o cliente realizou um *downgrade/upgrade* ou se o cliente removeu ou substituiu o provedor.
- ***Changelogs:*** o cliente pode mencionar nos *changelogs* e nas *release-notes* como foi realizada a recuperação da *breaking change*, principalmente se o cliente realizou um *downgrade/upgrade* na versão do provedor. Também, se o cliente consertou a *breaking change* diretamente no seu código, provavelmente há essa informação nos *changelogs*. Ao todo, 48% dos repositórios dos clientes continham *changelog* ou *release-notes*.
- ***Pull-requests/Issues:*** foram procurados *pull-requests* e *issues* no repositório do cliente que deveria consertar, ou conter informações sobre a *breaking change*. *Pull-requests/issues* nomeadas como *Update provider*, *Fix provider errors*, *Fix tests* indicavam a presença de alguma alteração que foi realizada devido às *breaking changes*.

Para cada caso de *breaking change* foi recuperada a árvore de dependências do cliente até o provedor que introduziu a *breaking change*. Por exemplo, em nosso segundo exemplo motivacional (Capítulo 3) foi recuperada a árvore de dependências a partir do cliente até broccoli-asset-rev→broccoli-filter→broccoli-plugin (Figura 3.1). Com isso foi analisada a quantidade de *breaking changes* introduzida por provedores diretos e indiretos.

Também foram investigados os dados sobre quando a *breaking change* foi introduzida, consertada, qual pacote consertou e como foi realizada a correção. Assim foi analisado o tempo que as *breaking changes* levaram para serem consertadas e quais são as principais maneiras como os clientes se recuperam das *breaking changes*. Ainda foram analisados os casos onde o provedor consertou a *breaking change* e, ainda assim, o cliente realizou um *upgrade/downgrade* da versão do provedor. Por fim, foi verificado como as versões dos provedores foram alteradas pelos clientes e como a documentação da *breaking change* influenciou na velocidade com que as *breaking changes* foram consertadas.

4.2 Coleta do Conjunto de Dados

O conjunto de dados utilizado neste trabalho foi extraído do registro do npm, do qual foram recuperados os arquivos de metadados *package.json* de 1.233.944 pacotes publicados no período de 20 de Dezembro de 2010 até 01 de Abril de 2020. Os principais dados recuperados no *package.json* são os *timestamp* de cada uma das *releases* dos pacotes, os provedores que os pacotes clientes continham em cada *release* e suas respectivas versões. A Figura 4.2 exibe as informações do pacote buffer-includes[4] que podem ser recuperadas de seu *package.json*.

Figura 4.2. Informações que serão recuperadas do *package.json* para validar um pacote.

Fonte: Autoria própria

[4] http://registry.npmjs.org/buffer-includes

Foram excluídos desse conjunto de dados todos os pacotes que não continham nenhum provedor, pois quando um pacote não contém provedores não há como ser impactado por *breaking changes*. Dessa maneira, o conjunto de dados final foi reduzido para um total de 987.595 pacotes clientes do npm. Por fim, para cada *release* de todos os clientes, foram resolvidas as versões de todos os provedores com base no *timestamp* dessa *release*. Ou seja, foi resolvido o *range* do provedor, que o cliente especificou naquela *release*, para a maior versão aceita no momento da *release* do cliente. Desse modo, foi determinada qual a versão do provedor o cliente utilizava no momento da publicação da *release*.

4.3 Amostra do Conjunto de Dados

Para gerar a amostra utilizada neste trabalho, foram verificados dois requisitos nos *987k* pacotes clientes: 1) possuir um *script* de teste não vazio e diferente do *script* padrão de teste do npm: `{"test": "Error: no test specified"}` (488.805 conferem); e 2) possuir a *url* do repositório (410.433 conferem). Esses dois requisitos foram analisados na última *release* disponível do pacote. Então, foi recuperada uma amostra representativa com 95% de confiança e 5% de margem de erro. Dentre os 410.433 pacotes clientes restantes, a amostra resultou em 384 pacotes para serem analisados neste trabalho. Esses 384 pacotes foram recuperados de forma aleatória.

Foi realizada uma verificação manual nos repositórios do pacotes com menos de 4 *releases* (130 dentre 384) para verificar se cada pacote não era um *toy package*, ou seja, um pacote que não foi criado para ser um projeto real, apenas um teste no npm, no GitHub ou algo do tipo. Um pacote foi classificado como *toy package* e substituído por outro que foi sorteado seguindo os dois requisitos acima citados.

4.4 Metodologia para Detecção de *Breaking Changes*

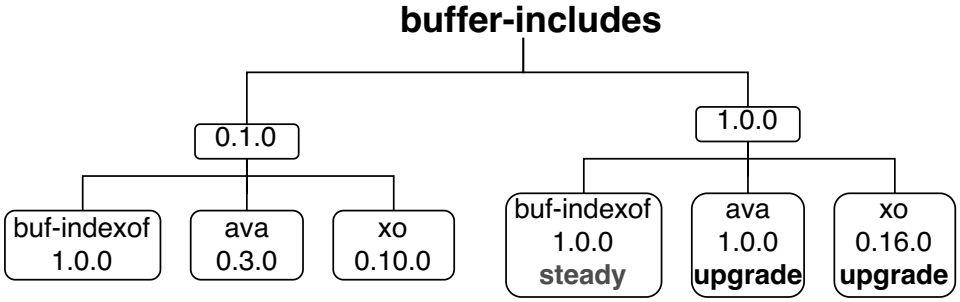
Para este trabalho, desenvolvemos uma ferramenta chamada BCDetect[5] disponível no GitHub sob a licença MIT e escrita na linguagem Python. A BCDetect lê um arquivo de entrada, clona o repositório listado nesse arquivo, restaura cada uma das *releases* do repositório e, para cada *release*, altera a versão de todas as dependências no `package.json`. Então, a BCDetect configura o sistema operacional para a versão do Node.js que deveria ser executado, de acordo com a data da *release*. Em seguida, executa os comandos `npm install/test` e salva os resultados.

Primeiramente, a BCDetect clona o repositório do respectivo cliente – todos os clientes estavam hospedados no Github – e cria uma estrutura de dados para armazenar as informações sobre o cliente. Nessa estrutura, cada *release* do cliente contém todos os provedores com suas versões resolvidas e o tipo de atualização que os provedores realizaram desde a última *release* do cliente: `steady` significa que o provedor não publicou nenhuma *release* aceita desde a última release do cliente; `upgrade` significa que o provedor publicou uma nova *release* aceita desde a última *release*

[5] https://github.com/danielventurini/bcdetect

do cliente; e quando não há nenhuma dessas informações, o provedor foi inserido no *package.json* nesta *release*. Essa estrutura básica está representada na Figura 4.3, construída a partir dos dados do cliente buffer-includes da Figura 4.2.

Figura 4.3. Estrutura de dados para representar o cliente buffer-includes.



Fonte: Autoria própria

Os testes de cada *release* do cliente foram executados toda vez que havia pelo menos um provedor com uma nova *release* (*upgrade*) ou um provedor havia sido adicionado naquela *release* do cliente. Após clonado o repositório, foi executado o comando `git checkout` para todas as *releases* do cliente – *release* por *release* – com a respectiva *tag* da *release*, fazendo com que todos os arquivos sejam restaurados para exatamente os mesmos arquivos do momento em que o cliente publicou a *release*. Uma *tag* é uma referência a um ponto importante e específico do repositório, que geralmente são as *releases*. Para *releases* que o desenvolvedor não criou uma *tag*, o *checkout* foi realizado usando o *timestamp* da respectiva *release*. Em seguida, o arquivo *package-lock.json*[6] foi excluído, pois esse arquivo altera o comportamento do comando `npm install` – a partir do npm@5 – fazendo com que o npm instale as versões dos provedores de acordo com o *package-lock.json*, e não de acordo com o *package.json*. Em sequência, todos provedores no *package.json* e suas respectivas versões foram adicionados apenas no campo *dependencies* do *package.json* e os demais campos foram removidos,[7] uma vez que para executar os testes, ambos os provedores são requeridos.

O Node.js é o ambiente de execução para os pacotes JavaScript e a cada 6 meses uma nova *release major* é publicada.[8] Por isso, antes de executar os testes da *release* do cliente, a versão do Node.js precisa ser alterada. O chaveamento das *releases* do Node.js é necessário pois as *releases major* do Node.js não são retrocompatíveis, ou seja, um pacote que executa com sucesso na versão *0.x* do Node.js, por exemplo, provavelmente não executaria com sucesso na versão *8.x*. Para cada *release* do cliente que teria seus testes executados, a versão do Node.js foi selecionada de duas maneiras: 1) do campo `engines->node` no *package.json*, que permite o desenvolvedor especificar a versão do Node.js; e 2) através do *timestamp* da *release* do cliente, foi possível identificar a última *release* do Node.js disponível,[9] ou seja, qual era a *release* máxima do Node.js que os testes dos cliente foram executados no momento da *release* do cliente. Assim, os testes do cliente foram executados em todas

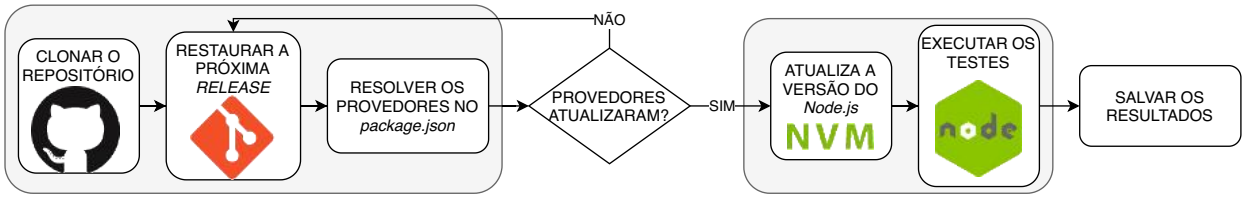
[6] https://docs.npmjs.com/files/package-lock.json
[7] campos para dependências no *package.json*, tais como o *peerDependencies*, *optionalDependencies* e o *globalDependencies*
[8] https://github.com/nodejs/node#release-types
[9] https://nodejs.org/en/download/releases

as versões *major* do Node.js, da versão mais atual, pelo *timestamp* da *release* do cliente, até a versão *major* mais antiga, ou até o teste executar com sucesso. Para o cliente da Figura 4.3, a sua *release* *0.1.0* possui o *timestamp* como *2015-10-28*, e a última *release* do Node.js disponível até esta data é a *4.2.1*. Assim, os testes dessa *release* do cliente seriam executados com as *releases 4.x, 3.x, 2.x, 1.x, 0.x* do Node.js. Ao atualizar a *release* do Node.js, a versão do npm é atualizada também. Isso é necessário pois é o npm que executa os *scripts install* e *test*. Após executar o *install* e o *test*, foram salvas as seguintes informações:

- versão do cliente;
- se houve alteração na versão aceita de alguns dos provedores;
- os códigos da execução do `npm install` e `npm test` – sucesso ou erro;
- a versão do Node.js que deveria ser executado com base na data da *release*; e
- a versão do Node.js que os testes do cliente executaram com sucesso.

Os passos resumidos das operações em cada cliente juntamente com a verificação de alteração dos provedores e execução dos testes das *releases* se encontram na Figura 4.4.

Figura 4.4. Etapas para clonar, restaurar e execução os testes das *releases* de um cliente



Fonte: Autoria própria

Dos 384 clientes que tiveram seus testes executados, em 33 não foi possível executar o comando `npm install/test` para nenhuma de suas *releases*. Desses 33 clientes, 15 não possuíam algum dos arquivos necessários para os testes; 11 continham *scripts* de teste inválidos em todas as suas *releases*, tal como `{"test": "no tests"}`; 4 haviam listados alguns arquivos no *.gitignore* – arquivo utilizado pelo git para ignorar arquivos no repositório –, mas que eram necessários para a execução dos testes; 2 necessitaram de configurações específicas em banco de dados e não foi possível realizá-las; e 1 cliente requeria uma variável de ambiente para acessar um determinado site. Dessa forma, os 33 clientes foram substituídos em um novo sorteio seguindo o mesmo critério do sorteio anterior citado na Seção 4.3, totalizando 384 clientes com 5957 *releases* que foram utilizados no estudo.

Um pacote de replicação incluindo a amostra utilizada neste trabalho, as ferramentas, *scripts* e todos os casos de *breaking changes* está disponível no *Zenodo*.[10]

[10] https://doi.org/10.5281/zenodo.4284035

5 RESULTADOS

Este capítulo apresenta as principais descobertas sobre os dados de cada questão de pesquisa. A Subseção 5.1 descreve os resultados sobre a execução dos *scripts* de testes das *releases* dos pacotes clientes. Esses dados foram analisados nas questões de pesquisa. As Subseções 5.2, 5.3 e 5.4 contém as descobertas para a primeira, segunda e terceira questões de pesquisa, respectivamente.

5.1 Resultados preliminares: execução dos testes dos clientes

Após os comandos `npm install/test` executarem em pelo menos uma *release* de todos os 384 clientes, 203 resultaram em erros para alguma de suas *releases*. Analisando os resultados por *releases*, de todas as 5957 *releases*, foram executadas os testes de um total de 3230 *releases*, enquanto que 2727 *releases* não tiveram seus testes executados pois não haviam alterações nas *releases* dos provedores, ou não continham algum *script* de teste válido. Após o término da execução, 1954 *releases* resultaram em sucesso, enquanto que 1276 *releases* resultaram em erros no `script install/test`. A Tabela 5.1 contém os resultados prévios sem que os clientes que resultaram em erro fossem analisados, ou seja, são dados apenas da execução dos testes dos clientes.

Tabela 5.1. Resultado da execução dos testes nas *releases* de cada cliente, por clientes e por *releases*.

	Clientes	*Releases*
Total	384	5957
Não executado	–	2727
Executado	384	3230
Sucesso	181	1954
Erro	203	1276

Fonte: Autoria própria

Os resultados apresentados na Tabela 5.1 são a base para a análise e geração dos resultados em cada uma das questões de pesquisas. A partir das execuções dos testes que resultaram em erro, foi aplicada a metodologia da primeira questão de pesquisa (Subseção 4.1) para encontrar quais casos de erros se referem à casos reais de *breaking changes*. Esses casos reais foram analisados em cada uma das questões de pesquisa.

5.2 QP1: Frequência de Breaking Changes

Nesta Seção, encontram-se os resultados da análise manual realizada para responder a primeira questão de pesquisa: “Com que frequência *breaking changes* impactam nos pacotes clientes?”. Apresentamos os dados relacionados à quantificação das *breaking changes*.

Descoberta #1: 11,7% dos pacotes clientes e 13,9% das *releases* dos clientes são impactados por *breaking changes*

De todos os 384 pacotes clientes, as *breaking changes* se manifestaram em pelo menos uma *release* em 45 pacotes (11,7%). De todas as 3230 *releases* dos clientes que foram executadas, 1276 resultaram em erro e foram analisadas manualmente. Após a análise, foi detectado que, de todas as executadas, em 479 *releases* (14,8%) o erro foi introduzido unicamente pelos pacotes cliente, não caracterizando uma *breaking change*, uma vez que os provedores não influenciaram nesses erros. Mas, em 450 *releases* (13,9%), o erro foi introduzido pelos provedores, caracterizando uma *breaking change*. Em 86 *releases* (2,7%) não foi possível identificar qual pacote introduziu o erro.

Em 261 *releases* (8,1%) houve um tipo especial de erro: *breaking change induzida*. Essas *releases* utilizavam um provedor que recuperava dados de serviços externos (por exemplo, a *API* do Twitter), e, naturalmente, esses dados se alteraram ao longo do tempo. Então, o provedor, por não ser o mantenedor dos dados, não tinha como alterar esses dados para se recuperar desse erro. Esses casos não foram considerados casos de *breaking changes*, uma vez que esses dados não fazem parte do ecossistema do npm. Ainda, casos onde os provedores foram removidos do npm também foram considerados como *breaking changes* induzidas. A Tabela 5.2 apresenta os resultados da análise por *releases*.

Tabela 5.2. Resultado da análise das *releases*

Resultados		**(#)**	**(%)**
Sucesso		1.954	60,5
Erros	Erros do cliente	479	14,8
	Breaking changes	450	13,9
	Breaking changes induzidas	261	8,1
	Erros não identificados	86	2,7

Fonte: Autoria própria

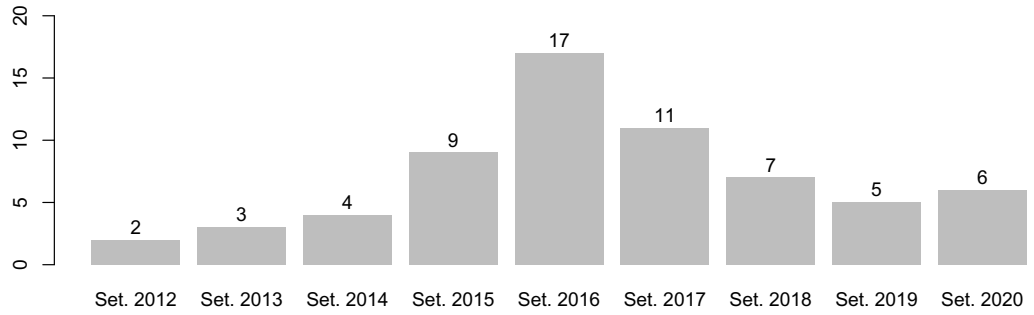
Descoberta #2: 51 pacotes provedores introduziram 64 casos de *breaking changes*

Cada caso de *breaking change* é um erro introduzido por uma *release* do provedor. Enquanto 47 provedores introduziram uma única *release* com *breaking changes*, 4 provedores introduziram 2 *releases* com *breaking changes*, cada um deles. Ou seja, cada um desses 4 provedores possui 2 *releases* que introduzem *breaking changes*. Assim, foram detectados 55 casos únicos introduzidos pelos provedores. Entretanto, alguns desses 55 casos impactaram mais de um cliente da amostra. Uma vez que esses provedores são usados por vários clientes, uma única *release* com *breaking change* impactou mais de um cliente. Por exemplo, o caso de *breaking change* da categoria *Versão de provedores incompatíveis* (Seção 5.3) impactou 6 clientes diferentes, ou seja, um único caso no provedor impactou mais de um cliente. Deste modo, um total de 64 casos de *breaking changes* foram manifestados nos pacotes clientes.

Descoberta #3: Frequência de ocorrências das *breaking changes*

A base de dados dos pacotes clientes utilizada nesse estudo se estende de Dezembro de 2010 à Abril de 2020. O primeiro caso de *breaking change* detectado na análise manual foi introduzido pelo provedor em Novembro de 2011, e o último caso, em Setembro de 2020. Nesse período, outros 62 casos de *breaking changes* ocorreram. A Figura 5.1 apresenta a quantidade de *breaking changes* introduzidas ao longo dos anos, ano após ano.

Figura 5.1. Número de casos de *breaking changes* ao longo dos anos



Fonte: Autoria própria

O fenômeno das *breaking changes* teve um crescimento, de Setembro de 2011 até Setembro de 2016 e de Setembro de 2019 até Setembro de 2020. A maior taxa de crescimento ocorreu no período de Setembro de 2014 até Setembro de 2015, período no qual as *breaking changes* aumentaram de 4 casos em 2014 para 9 casos em 2015, um aumento de 125%, seguido do período 2015-2016, onde as *breaking changes* quase dobraram em número de ocorrências. A manifestação das *breaking changes* decresceu de Setembro de 2016 até Setembro de 2019.

Descoberta #4: 54,9% das *releases* dos provedores com *breaking changes* possuem mais *commits* do que as outras *releases* no mesmo nível *major*

As *releases* dos provedores com *breaking changes* possuem uma característica interessante: mais da metade delas possuem mais *commits* do que a mediana dos *commits* em outras *releases* que não introduziram *breaking changes* no mesmo nível *major* do respectivo provedor. Ou seja, foi detectado que, no mesmo nível *major*, o provedor introduz mais *commits* na *release* que introduz uma *breaking change* do que nas *releases* que não introduzem *breaking changes*. Ainda, 42,3% dessas *releases* possui mais de 50% de *commits* do que a mediana das demais *releases* no mesmo nível *major*. Ou seja, a cada dois *commit* em uma *release* que não possui *breaking change*, há três *commits* na *release* que introduz *breaking changes*. Esses dados podem indicar que quando os provedores realizam mais *commits* do que a mediana de suas releases, podem introduzir mais alterações do que as presumidas pelos clientes. Embora estudos futuros sejam necessários, nesses casos, as alterações introduzidas em uma *release* poderiam ser segmentadas, permitindo uma melhor inspeção pelos clientes a cada *release*.

Do outro lado, aproximadamente um terço das *releases* dos provedores com *breaking changes* (29,2%) possuem menos *commits* do que a mediana das outras *releases* no mesmo nível *major*. Dessas

releases, 57,1% introduziram apenas um *commit* na *release* e foi esse o *commit* que introduziu a *breaking change*.

Destaques da QP1: 11,7% dos pacotes clientes e 13,9% das *releases* foram impactados por *breaking changes*. O fenômeno das *breaking changes* têm ocorrido ao longo dos anos e mais da metade das *releases* dos provedores que introduzem *breaking changes* recebem mais *commits* do que as outras *releases* no mesmo nível *major*.

5.3 QP2: Erros introduzidos pelos provedores

Nesta Seção, estão os resultados e a análise dos dados relacionados à segunda questão de pesquisa: "Como os pacotes provedores introduzem *breaking changes* em uma *release*?".

Descoberta #5: Categorias de *Breaking changes*

Ao todo, foram descobertos 64 casos de *breaking changes* distribuídas em 45 clientes. Todos esses casos foram agrupados em 8 categorias das quais se combinavam os casos semelhantes, e um segundo pesquisador os validaram. A Tabela 5.3 apresenta cada uma dessas categorias, bem como a quantidade de pacotes e a quantidade de *releases* que cada categoria impactou.

Tabela 5.3. Categorias dos casos de *breaking changes*

Categoria	**Casos**		***Releases***	
	(#)	**(%)**	**(#)**	**(%)**
Alteração de funcionalidade	25	39,1	101	22,5
Provedores incompatíveis	15	23,4	64	14,2
Alteração de tipo de objeto	9	14,1	213	47,3
Objeto indefinido	5	7,8	28	6,2
Código incorreto	5	7,8	14	3,1
Código desatualizado	2	3,1	24	5,3
Renomeação de função	2	3,1	2	0,5
Arquivo não encontrado	1	1,6	4	0,9
Total	**64**		**450**	

Fonte: Autoria própria

A seguir, encontra-se uma descrição sobre cada categoria e um exemplo que foi encontrado na análise manual sobre como os provedores introduziram os erros da categoria.

- **Alteração de funcionalidade**: os casos dessa categoria foram os que mais impactaram os clientes. Essa categoria contém os casos de *breaking change* no qual os provedores possuíam um determinado comportamento, mas alteraram algumas de suas regras/funcionalidades e impactaram os seus clientes. Não foi uma simples alteração no código, mas sim uma alteração em regras no qual os clientes tinham como sólida. Por exemplo, a *release* request@2.17.0 –

essa *release* foi removida do npm, mas a alteração se manteve – introduziu uma alteração no código,[1] como pode ser visto no Código 5.1.

```
  debug('emitting complete', self.uri.href)
+ if(response.body == undefined && !self._json) {
+   response.body = "";
+ }
  self.emit('complete', response, response.body)
```

Código 5.1. Exemplo da categoria *Alteração de funcionalidade*

Nesse caso, o request adiciona uma *string* vazia ao invés de manter *undefined* no corpo de uma requisição. Esse caso do request ocorreu porque os pacotes provedores evoluem independentemente dos clientes (FOO, 2018). Essa alteração na regra do request reflete em uma evolução do pacote, mas os clientes não esperavam essa alteração e confiavam que o corpo da resposta fosse retornado como *undefined*, por isso os clientes sofreram um erro.

- **Provedores incompatíveis**: nessa categoria, há um provedor direto A e um provedor indireto B envolvido, o qual alterou o seu código, o que não gerou um erro, mas provocou um comportamento inesperado no provedor A. Ou seja, o provedor B passou a ser incompatível com o provedor A. Nessa categoria, nenhum dos provedores contém um erro, mas sim uma incompatibilidade. Um exemplo disso ocorreu com os pacotes babel-eslint e escope, sendo o pacote escope é um provedor indireto do babel-eslint.

```
  }
-   },
-   visitClass: {
+ }, {
+   key: 'visitClass',
      value: function visitClass(node) {
```

Código 5.2. Exemplo da categoria *Provedores incompatíveis*

A *release* escope@3.4 realizou uma alteração no seu código, conforme o Código 5.2, mas que não reflete em um erro. Essa alteração impactou diretamente o pacote babel-eslint, mesmo o pacote escope não sendo um provedor direto do babel-eslint e não ter introduzido um erro.[2] Com isso, houve uma incompatibilidade entre os provedores e essa incompatibilidade precisou ser corrigida pelo babel-eslint. Essa *breaking change* durou apenas um dia, mas durante esse período o babel-eslint foi descarregado *80k* vezes do npm.

- **Alteração de tipo de objeto**: em linguagens fortemente tipadas essa categoria de *breaking change* não ocorre, mas no Javascript representam um tipo de *breaking change* que, por muitas vezes, pode nem se manifestar no cliente. Neste trabalho, foram detectados 9 casos (14,1%) nos quais os provedores alteraram o tipo de algum objeto.

[1] https://github.com/request/request/commit/d05b6ba
[2] https://github.com/estools/escope/issues/99#issuecomment-178151491

```
- this.sockets = [];
+ this.sockets = {};
  this.nsps = {};
    this.connect Buffer = [];
  }
  var socket = nsp.add(this, function() {
-   self.sockets.push(socket);
+   self.sockets[socket.id] = socket;
    self.nsps[nsp.name] = socket;
```

Código 5.3. Exemplo da categoria *Alteração de tipo de objeto*

No Código 5.3 o provedor socket.io alterou um *array* para *object*.[3] Anteriormente, os clientes iteravam nesse *array*, mas após essa alteração os clientes foram impactados por uma *breaking change*. Mesmo sendo uma simples alteração, muitos clientes do socket.io foram impactados, até mesmo o pacote karma,[4] um utilitário para testes em *browsers*, que foi forçado a alterar seu código[5] e publicar a *relese* karma@0.13.19. Enquanto essa *breaking change* permaneceu por apenas um dia, o karma recebeu *146k downloads* do npm.

- **Objeto indefinido**: por vezes, os códigos podem estar todos corretos, mas os provedores tentam acessar uma variável que não existe. Esta categoria de *breaking change* representa os casos no qual os provedores tentaram obter acesso à alguma variável/objeto, mas que não existiam.

```
+ app.options = app.options || {};
  app.options.babel = app.options.babel || {};
  app.options.babel.plugins = app.options.babel.plugins || [];
```

Código 5.4. Exemplo da categoria de *Objeto Indefinido*

Esse tipo de erro surgiu na *release* ember-cli-htmlbars-inline-precompile@0.1.3, na qual o desenvolvedor tenta acessar uma variável que não estava disponível. Mas, assim como o desenvolvedor já havia feito com as demais variáveis do Código 5.4, uma simples alteração no código[6] foi o suficiente.

- **Código incorreto**: este caso de *breaking change* ocorreu quando os provedores escrevem um código semanticamente incorreto, gerando um erro na sua execução e afetando os clientes. Em linguagens compiladas, esse tipo de erro seria facilmente identificado em tempo de compilação, mas no Javascript esse tipo de erro apenas se manifesta em tempo de execução. Foi o que ocorreu com a *release* front-matter@0.2.0.Apesar de ser um erro facilmente detectável e corrigível, como o provedor fez[7] no Código 5.5, o provedor front-matter aguardou aproximadamente um

[3] https://github.com/socketio/socket.io/commit/b73d9be
[4] https://github.com/socketio/socket.io/issues/2368
[5] https://github.com/karma-runner/karma/commit/3ab78d6
[6] https://github.com/ember-cli/ember-cli-htmlbars-inline-precompile/pull/5/commits/b3faf959
[7] https://github.com/jxson/front-matter/commit/f16fc01

ano para corrigir e publicar a *release* front-matter@0.2.2. Durante esse período, front-matter foi descarregado do npm 366 vezes.

```
  const separators = [ '---', '= yaml =']
- const pattern = pattern = '^('
+ const pattern = '^('
    + '((= yaml =)|(---))'
```

Código 5.5. Exemplo da categoria *Código incorreto*

- **Código desatualizado**: nessa categoria um provedor A atualiza o seu provedor B explicitamente – alterando manualmente o *range* do seu provedor no *package.json* –, mas o provedor A não atualiza o seu código para executar de acordo com a nova versão do provedor B. Consequentemente, uma *breaking change* é introduzida nos pacotes clientes. Além de um caso de atualização explícita, foi detectado um caso no qual um provedor A especificou um provedor B com o *X-range (>=)*, e ao longo do tempo o provedor B publicou uma *release major* que introduziu *breaking changes*. Apesar do provedor A especificar um *X-range*, A não atentou-se para a atualização implícita de B com uma *breaking change*, e essa *breaking change* foi cascateada para os clientes.
- **Renomeação de função**: as *breaking changes* relacionadas à esta categoria são facilmente detectáveis. Quando a mensagem de erro do Node.js é exibida como *TypeError: var is not a function*, provavelmente trata-se de uma determinada função que não esta disponível, ou seja, foi removida ou renomeada. Todas os dois casos nesta categoria ocorreram porque a função foi renomeada. O primeiro caso está descrito nos exemplos motivacionais (Capítulo 3).

```
- RedisClient.prototype.send_command = function (command, args, callback) {
-     var args_copy, arg, prefix_keys;
+ RedisClient.prototype.internal_send_command = function (command, args, callback) {
+     var arg, prefix_keys;
```

Código 5.6. Exemplo da categoria *Renomeação de função*

O segundo caso ocorreu quando a *release* redis@2.6.0-1 renomeou uma função, como no Código 5.6.[8] Entretanto, essa função era apenas utilizada internamente pelo pacote fakeredis,[9] que foi impactado pela *breaking change*. Essa *breaking change* foi corrigida no pacote cliente fakeredis@1.0.3 que realizou um *downgrade* para a versão redis@2.6.0-0.[10] Em cinco dias nos quais a *breaking change* não foi corrigida, o fakeredis foi descarregado *2,3k* vezes do npm.

- **Arquivo não encontrado**: os casos de *breaking change* relacionados à esta categoria são aqueles no qual os desenvolvedores realizam um acesso a um arquivo que não existe. O arquivo requerido pode não existir ou não estar disponível, uma vez que, referenciado no arquivo *.npmignore* – arquivo utilizado pelo npm para ignorar arquivos durante o processo de

[8] https://github.com/NodeRedis/node_redis/commit/861749f
[9] https://github.com/NodeRedis/node-redis/issues/1030#issuecomment-205379483
[10] https://github.com/hdachev/fakeredis/commit/01d1e99

publicação –, o arquivo existe mas não está disponível. Entretanto, o único caso de *breaking change* dessa categoria ocorreu pois o provedor referenciou o *index.js* ao *.npmignore*.

Os casos apresentados em cada categoria foram categorizados na categoria mais expressiva. Por exemplo, para casos onde um objeto foi alterado para que toda uma funcionalidade seja alterada, foi classificado como *Alteração de funcionalidade*, e não como *Alteração de tipo de objeto*.

Descoberta #6: Nível do Versionamento Semântico em que as *breaking changes* são introduzidas

De todos os 64 casos de *breaking changes*, 3 casos foram introduzidos no nível *major*; 26 casos, no *minor*; 28 casos, no *patch*; e 5 casos, pré-*releases*, como pode ser visto na Tabela 5.4. Foram analisados apenas os casos de *breaking changes* em *releases minor* e *patch*, mas houve 3 casos em que a *breaking change* foi introduzida em uma *release major*: em dois casos os provedores escreveram códigos errôneos na *release major* e o cliente a estava usando – como no caso do jsdom@16 (Cap. 3) – e o terceiro caso ocorreu quando um provedor A, que depende de um provedor B, aceitou a *release* com *breaking change* através do *X-range (>=)* do provedor B, mas não tratou essa *breaking change*.

Tabela 5.4. *Breaking changes* por nível do Versionamento Semântico

Níveis	**(#)**	**(%)**
Major	3	4,7
Minor	28	43,75
Patch	28	43,75
Pré-*release*	5	7,8

Fonte: Autoria própria

Pré-*releases* antecedem suas *releases* estáveis, mas não são consideradas estáveis; qualquer alteração não-retro compatível deve ser resolvida até a versão estável ser publicada.[11] Foram detectadas *breaking changes* em 5 casos de pré-*releases*, mas esses provedores cascatearam essas *breaking changes* para as *releases* estáveis. Por exemplo, a pré-*release* redis@2.6.0-1, descrita na categoria *Renomeação de função* da Seção 5.3, alterou o nome de uma função, e essa alteração não-retro compatível deveria ser corrigida até a *release* estável redis@2.6.0, mas essa alteração foi propagada para a *release* estável, impactando e execução do cliente com uma *breaking change*. Esse exemplo mostra como os provedores não estão devidamente orientados sobre as regras do Versionamento Semântico.

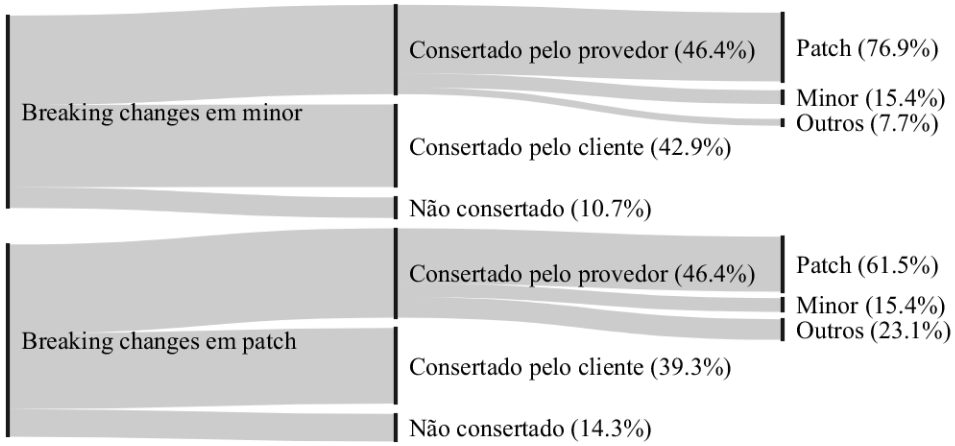
Descoberta #7: Provedores introduzem a mesma quantidade de *breaking changes* em *releases minor* e *patch*

Os provedores introduzem *breaking changes* em níveis não-*major* em dois cenários: 1) em *releases minor*, quando os provedores tinha a intenção de publicar novas funcionalidades retro-compatíveis,

[11] https://semver.org/#spec-item-9

mas essas funcionalidades tornaram-se *breaking changes*; e 2) em *releases patch*, quando os provedores tinha o propósito de publicar correções de erros. Nos casos das *releases patch*, os provedores estavam realmente tentando publicar uma correção de erro, mas essas alterações podem não ter consertado o erro anterior, mas sim introduzido mais erros. Para casos em *releases minor*, essas novas funcionalidades podem ser casos reais de *breaking changes* dos quais os provedores têm a responsabilidade de corrigir, ou podem ser novas funcionalidades que os clientes precisam se adaptar. Para analisar esse impasse, foi observado qual pacote corrigiu a *breaking change* e para qual nível a nova *release* foi publicada. A Figura 5.2 apresenta esses dados.

Figura 5.2. As *breaking changes* corrigidas pelos pacotes providores e clientes e o respectivo nível do Versionamento Semântico que a *release* de correção foi publicada



Fonte: Autoria própria

A Figura 5.2 mostra que metade das *breaking changes* introduzidas em *releases minor* são corrigidas pelos pacotes clientes. Ou seja, os provedores introduziram uma funcionalidade retro-compatível que tornou-se uma *breaking change*, mas foram os pacotes clientes quem precisaram se adaptar. Mas, quando os provedores corrigem as *breaking changes* introduzidas em *releases minor*, em 76,9% dos casos a correção é publicada em uma *release patch* e, de acordo com o Versionamento Semântico, essa *release* tinha a finalidade de corrigir um erro. Portanto, os provedores assumem que, de fato, uma *breaking change* foi introduzida e são eles os responsáveis pela correção.

Breaking changes introduzidas em *releases patch* deveriam conter apenas correções de erros, mas os provedores introduziram *breaking changes*. Essas *breaking changes* em *releases patch* são corrigidas pelos provedores em 46,4% das vezes. Ainda, em 61,5% das vezes os provedores publicam a correção em uma *release patch*, significando que houve a tentativa de corrigir algo.

Descoberta #8: 78,1% das *breaking changes* possuem pelo menos uma documentação

Quando uma *breaking change* se manifesta nos pacotes clientes, ela deve ser documentada em *issues* ou *pull-requests*. Essas funcionalidades integradas aos repositórios dos pacotes permitem aos clientes notificarem os provedores sobre erros ou questões acerca do próprio provedor. Também, as *breaking*

changes podem ser documentadas em *changelogs*, tanto quando a *breaking change* é introduzida ou quando a mesma é corrigida. As *changelogs* criadas pelos provedores contêm os detalhes das alterações que os provedores introduzem em uma determinada *release*.

Tabela 5.5. Como cada *breaking change* foi documentada

Documentação	**(#)**	**(%)**
Issue	32	64
Pull-request	22	44
Changelog de introdução	23	46
Changelog de correção	16	32

Fonte: Autoria própria

A Tabela 5.5 mostra que as *breaking changes* são documentadas em 55 casos (78,1%). Dos casos documentados, 70% possuem mais de um tipo de documentação. Por exemplo, o provedor foi notificado de uma *breaking change* através de uma *issue*, corrigiu-a e a documentou em seu *changelog*. Nesse caso houveram duas documentações. Esse é um número positivo para os clientes, uma vez que quando a mesma *breaking change* se manifesta em vários clientes, mas já foi previamente documentada, torna-se fácil para os clientes se recuperarem. Por fim, em 14 casos (21,9%) a *breaking change* não foi documentada.

Destaques da QP2: Os defeitos mais comuns nos pacotes provedores que causam *breaking changes* são aqueles relacionados com uma alteração de funcionalidade, quando dois provedores se tornam incompatíveis e alterações nos tipos de objetos. Os provedores introduzem *breaking changes* em níveis *minor* e *patch* na mesma frequência. A maioria das *breaking changes* corrigidas pelos provedores são publicada em *releases patch*. As *breaking changes* são documentadas em 78,1% dos casos e a principal funcionalidade usada para documentação são as *issues*.

5.4 QP3: Recuperação dos clientes

Esta Seção apresenta os resultados da terceira questão de pesquisa: "Como os pacotes clientes se recuperam das *breaking changes*?".

Descoberta #9: A maioria das *breaking changes* são introduzidas pelos provedores indiretos

As *breaking changes* também podem ser introduzidas pelos provedores indiretos e propagadas para os clientes. A Tabela 5.6 apresenta a profundidade/nível da árvore de dependência do cliente em que os provedores que introduziram as *breaking changes* estavam. Apenas 42,2% das *breaking changes* são introduzidas pelos provedores que o cliente realmente instalou e usa no seu código, ou seja, os provedores diretos que estão referenciados no *package.json* do cliente. Esses provedores estão no primeiro nível da árvore de dependências.

Um fato interessante é que as *breaking changes* introduzidas pelos provedores indiretos, aqueles que estão nos níveis dois, três e quatro, representam 57,8% dos casos. Seis casos estão no

Tabela 5.6. O quão profundo os provedores que introduziram a *breaking change* estão do cliente na árvore de dependências

Nível	(#)	(%)
1	27	42,2
2	30	46,9
>3	7	10,9

Fonte: Autoria própria

terceiro nível e um único caso está no quarto nível. Esses provedores não são instalados pelos clientes, mas eles são descarregados pelo npm quando os provedores diretos são instalado. Nesses casos, a *breaking change* podem ser totalmente desconhecida para os pacotes clientes, uma vez que os clientes podem não ter conhecimento sobre esses provedores indiretos.

Descoberta #10: *Breaking changes* que são corrigidas pelos provedores sendo clientes

Um pacote provedor é um pacote que outros pacotes dependem. Entretanto, quando o provedor possui algum provedor, o primeiro provedor pode ser considerado como um cliente do segundo provedor. Por exemplo, na seguinte árvore de dependências `cliente`→`prov1`→`prov2` – o `cliente` seria o pacote cliente que teve seus testes executados neste trabalho – o `cliente` tem como seu provedor o pacote `prov1`, e `prov1` tem o pacote `prov2` como seu provedor, que é um provedor indireto do pacote `cliente`. Então, quando a *breaking change* é introduzida pelo `prov2` e se manifesta em `prov1`, o `prov1` pode ser considerado como um pacote cliente de `prov2` uma vez que na árvore de dependências o `prov2` é um provedor direto de `prov1`. A partir desta perspectiva, os próximos resultados são apresentados considerando o pacote cliente como aqueles que são clientes do pacote que introduziu a *breaking changes* – exceto nos casos classificados como *Provedores incompatíveis* (Subseção 5.3), porque ambos os provedores devem ser considerados como provedores de `cliente`.

Nesse contexto, a Tabela 5.7 apresenta quais pacotes corrigiram cada caso de *breaking change*. A maioria das *breaking changes* foram consertados pelos pacotes provedores. Uma vez que são eles que introduziram a *breaking change*, teoricamente, esse era o comportamento esperado. Os pacotes clientes – aqueles que tiveram seus testes executados neste trabalho – recuperaram-se das *breaking changes* em 20,3% dos casos. Por último, os provedores que são considerados como clientes do provedor que realmente introduziu a *breaking change*, recuperam-se das *breaking changes* em 18,8% dos casos. Isso é ótima observação para os pacotes clientes, pois os provedores como clientes corrigem as *breaking changes* muitas vezes. Então, quando a *breaking change* não é corrigida pelo provedor que a introduziu, outro provedor, que é o cliente do provedor que introduziu a *breaking change*, pode consertar a *breaking change*.

Entretanto, considerando os cliente e os provedores como clientes, as *breaking changes* são consertadas pelos clientes em 39,1% dos casos. Esse valor indica que por várias vezes os clientes têm a tarefa de se recuperarem das *breaking changes*.

Tabela 5.7. Pacotes que consertaram a *breaking change*

Consertado por	**(#)**	**(%)**
Provedor	32	50
Cliente	13	20,3
Provedor como cliente	12	18,8
Não consertado	7	10,9

Fonte: Autoria própria

Descoberta #11: Provedores como clientes são os pacotes que corrigem as *breaking changes* mais rapidamente

Quando uma *breaking change* é introduzida, ela deve ser corrigida pelos provedores que a introduziram, pelos provedores como clientes ou pelos pacotes clientes. A Tabela 5.8 apresenta o tempo que cada pacote gastou para consertar os casos de *breaking changes*. Em geral, as *breaking changes* são consertadas em 7 dias pelo provedor que as introduziram. Mesmo considerando esse curto período de tempo, um grande número de clientes diretos e indiretos podem ser afetados pelas *breaking change* enquanto elas não são corrigidas.

Tabela 5.8. Mediana dos dias gastos para consertar as *breaking changes*

Consertador por	**Dias**
Provedor	6
Cliente	34
Provedor como cliente	4
Total	7

Fonte: Autoria própria

No geral, os pacotes que mais rápido corrigem as *breaking changes* são os provedores como clientes. Na verdade, a uma *breaking change* somente existe quando ela se manifesta nos pacotes clientes, portanto, os provedores como clientes são os primeiros pacotes a serem afetados pelas *breaking changes* e são os que realmente precisam se recuperar. Considerando os provedores e os provedores como clientes, as *breaking changes* são corrigidas em uma mediana de 5 dias por esses provedores, enquanto que os clientes gastam 34 dias para se recuperarem das *breaking chnages*. Então, os pacotes que devem corrigir as *breaking changes* são os provedores e os provedores como clientes, uma vez que os clientes demoram para se recuperarem e, de acordo com a Tabela 5.7, os provedores e os provedores como clientes são os que corrigem a maioria das *breaking change* (78,8%). Finalmente, as *breaking changes* indiretas – aquelas corrigidas pelos provedores como clientes – são consertadas 1,5 vezes mais rápidas do que as *breaking changes* diretas.

Descoberta #12: Como os clientes se recuperam das *breaking changes*

A Tabela 5.9 apresenta os dados sobre as formas que os clientes se recuperam das *breaking changes*. Em 48 casos os clientes alteraram a versão de seus provedores – mesmo quando os provedores

consertam as *breaking changes*. Apesar de ser a maneira mais fácil de se recuperaram das *breaking changes*, na maioria dos casos (71,4%), ao invés de realizarem um *downgrade*, os pacotes clientes realizam um *upgrade* na versão de seus provedores. A partir desse resultado, foi verificado todos os casos onde os clientes e os provedores como clientes consertaram a *breaking change* apenas alterando a versão do provedor antes do provedor ter consertado a *breaking change*. Foi observado que os clientes e provedores como clientes realizam um *upgrade* em 12 (52,2%) casos de 23. Ou seja, mais da metade dos casos do qual os clientes e os provedores como clientes consertaram a *breaking change*, os pacotes provedores já tinham *releases* mais atuais, mas esses clientes não estavam usando essas novas *releases*.

Tabela 5.9. Como os pacotes clientes alteram a versão dos provedores após uma *breaking change*

Pacote	**Total**	***Upgr.***	***Downg.***	**Troca**	**Remoção**
Cliente	28	71,4%	21,4%	3,6%	3,6%
Provedor como Cliente	20	45%	50%	5%	—

Fonte: Autoria própria

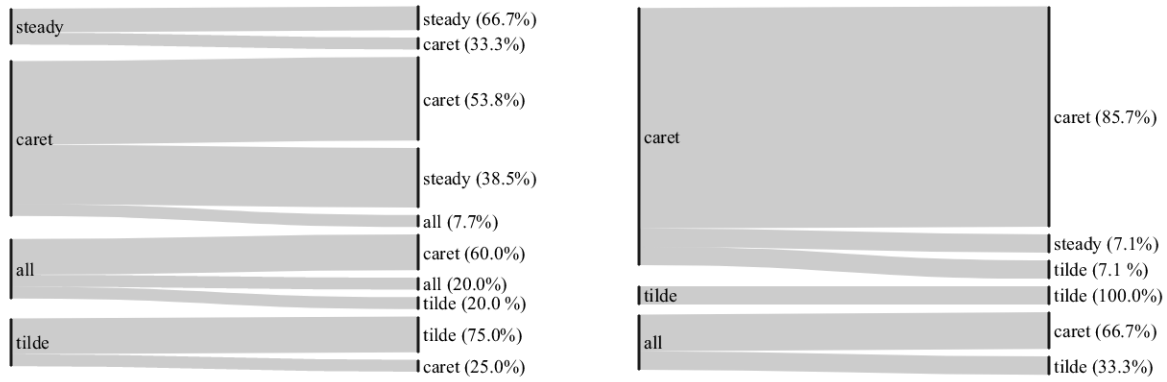
O número de *downgrades* nos provedores como clientes pode explicar o motivo pelo qual corrigem as *breaking changes* mais rápido que outros pacotes. Uma vez que os provedores como clientes também são provedores, eles devem consertar as *breaking changes* o mais rápido possível, assim evitando a propagação das *breaking changes*. Portanto, realizar um *downgrade* para uma *release* estável do provedor, como ocorreu na metade dos casos, é a maneira mais rápida de se recuperar das *breaking changes*. Finalmente, em uma pequena proporção, os provedores são removidos ou trocados após a introdução das *breaking changes*. Isso mostra o quão dependente os pacotes cliente podem ser dos seus provedores.

Descoberta #13: Os clientes geralmente alteram a versão do provedor, mas para o mesmo *range*

Geralmente, os clientes alteram a versão de seus provedores. Isso garante que o npm não irá instalar as versões prévias do provedor que são aceitas pelo *range*. Entretanto, quando uma *breaking change* se manifesta, é praticamente obrigatório atualizar a versão do provedor. A Figura 5.3 apresenta quando os clientes e os provedores como clientes atualizaram a versão dos seus provedores.

Foi verificado que os provedores como clientes nunca utilizam a versão de seus provedores como *steady*, ou seja, sem o *range*. Quando uma *breaking change* é manifestada nos provedores como clientes, eles sempre estão usando os seus provedores com um *range*. Entretanto, apenas em um único caso o provedor como cliente alterou o *range* de um *caret* para um *steady*. Mas, quando os clientes estão usando um *range caret* e são impactados por uma *breaking change*, em 38,5% dos casos, eles realizam um *downgrade* na versão do provedor alterando para um *range steady*. Essa é a maneira

Figura 5.3. A versão dos provedores alteradas pelos clientes e pelos provedores como clientes



(a) Clientes (b) Provedores como clientes

Fonte: Autoria própria

mais rápida e fácil de se recuperar de uma *breaking change*, mas o empecilho está no fato de se utilizar uma versão antiga do provedor.

A maioria das *breaking changes* são introduzidas quando os clientes e provedores como clientes estão usando o *range caret*. Esse é o *range* padrão que o npm insere no *package.json* quando os provedores são instalado. Em mais da metade dos casos, esses clientes alteram a versão dos provedores para outro *range caret*. Ainda, os *X-range*, ou *range all*, são os menos usados e os apenas uma minoria das alterações de *range* são convertidas para um *X-range*.

Os clientes e provedores como clientes tendem a manter o mesmo *range*, apenas atualizando esse *range*. Em 60,5% dos casos, o tipo do *range* (*X-range*, *range caret*, *range tilde* ou *steady*) é mantido, mas é realizado um *downgrade/upgrade* no *range* do provedor. Por exemplo, um cliente especifica o provedor como p@^1.2.0 e recebe uma *breaking change* em p@1.3.2. Se o provedor corrige o erro, o cliente irá atualizar para, por exemplo, p@^1.4.0, mas não irá mudar para outro tipo de *range*, tal como para um *range tilde*.

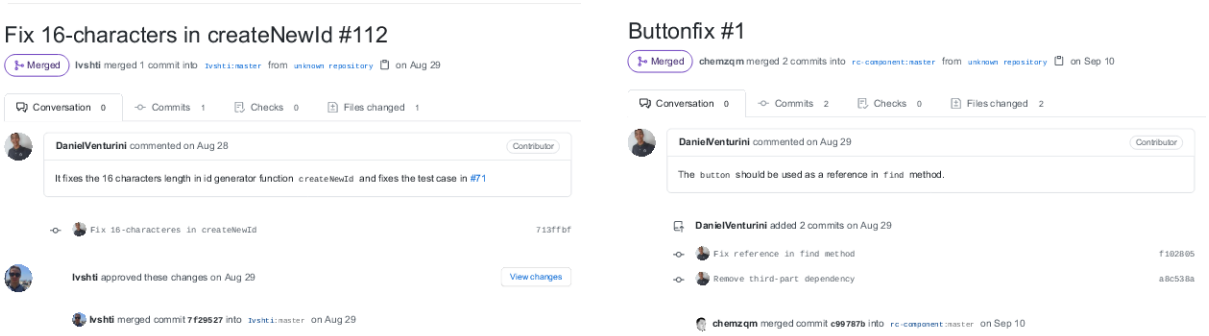
Descoberta #14: As *breaking changes* documentadas são corrigidas 3,3 vezes mais rápidas do que as *breaking changes* não documentadas

Breaking changes podem ser documentadas em *issues*, *pull-requests* ou *changelogs*. Tais documentações ocorrem em 78,1% dos casos. A documentação afeta diretamente o tempo gasto para corrigir uma *breaking change*. As *breaking changes* que possuem pelo menos um tipo de documentação são corrigidas, em mediana, 3 dias após serem introduzidas. Em oposição, quando as *breaking changes* não são documentadas elas recebem uma correção somente após 10 dias. Então, a documentação é muito importante para os clientes e os provedores quando eles estão tentando rastrear e corrigir as *breaking changes*

Descoberta #15: Contribuições com a comunidade

Esse trabalho estuda o surgimento de *breaking changes* nos pacotes provedores e as suas manifestações nos pacotes clientes. Todos os casos encontrados foram confirmados como casos reais de *breaking changes*, dos quais a grande maioria já havia sido resolvidos pelos provedores e pelos clientes. Entretanto, alguns casos não haviam sido corrigidos e, a partir da análise manual, os pesquisadores deste trabalho as descobriram e as corrigiram. Esse trabalho contribuiu com a comunidade provendo quatro *pull-requests*. A Figura 5.5(a) apresenta um *pull-request* criado no repositório do pacote linvodb3[12] e a Figura 5.5(b), no repositório do pacote rc-minidialog.[13]

Figura 5.4. *Pull-requests* criados nos repositórios



(a) *Pull-request* em linvodb3 (b) *Pull-request* em rc-minidialog

Fonte: Autoria própria

Destaques da QP3: Os pacotes clientes, incluindo os clientes diretos e os provedores como clientes, corrigem as *breaking changes* em 39,1% dos casos. Os casos de *breaking changes* introduzidas pelos provedores indiretos são os mais comuns. Também, os provedores corrigem as *breaking changes* mais rápidos do que os clientes, e os clientes preferem realizar um *upgrade* a um *downgrade* na versão do provedor. Finalmente, o *range* dos provedores podem ser alterados após uma *breaking changes*, mas, no geral, os clientes não alteram o tipo do *range*. As *breaking changes* documentadas são corrigidas 3,3 vezes mais rápido do que as demais.

[12] https://github.com/Ivshti/linvodb3/pull/112
[13] https://github.com/rc-component/dialog/pull/1

6 DISCUSSÕES

O principal resultado deste trabalho é que realmente os pacotes provedores introduzem *breaking changes* em *releases minor* e *patch*. Essas alterações, que se tornaram *breaking changes*, deveriam ser introduzidas apenas em *releases major* para manter todas as *releases minor*/*patch* retro compatíveis, seguindo o Versionamento Semântico. A Subseção 6.1 discute o gerenciamento de dependências na perspectiva do cliente. A Subseção 6.2 apresenta algumas práticas que podem ser seguidas pelos pacotes *clientes* e *provedores* para evitar/mitigar o impacto das *breaking changes*. Finalmente, a Subseção 6.3 discute alguns aspectos gerais sobre *breaking changes* no ecossistema do npm.

6.1 Gerenciamento de dependências

O gerenciamento de dependências é uma tarefa difícil para os pacotes clientes. Os clientes devem estar consciente sobre as versões de seus provedores e suas atualizações. Não é uma boa prática definir o *range* dos provedores como *X-range* ou como um *range* específico – quando não há *range*, especificado com a opção `–save-exact` no processo de instalação. Foi verificado que esses são os tipos de *range* menos comum (Descoberta #6). Ainda, o *range* específico é tão prejudicial quanto o *X-range*, uma vez que ele impede os clientes de serem atualizados com as novas *releases* dos provedores, enquanto que o *X-range* provavelmente irá introduzir uma *breaking change* no futuro. Portanto, foi verificado que algumas vezes os clientes estão utilizando versões desatualizadas dos provedores (Descoberta #12).

Para facilitar o gerenciamento de dependências, os clientes podem utilizar atualizadores automáticos de dependências no GitHub para receberem *pull-requests* quando há alguma nova versão dos provedores. Exemplos são o Snyk e o Dependabot. Esses automatizadores verificam continuamente por novas versões dos provedores que contenham correções de erros/vulnerabilidades e novas funcionalidades, de acordo com o *range* especificado pelos clientes. Esses automatizadores exibem o *changelog* de cada nova versão e criam *pull-requests* nos repositórios dos clientes atualizando seus *package.json*. O trabalho de Mirhosseini e Parnin (2017) mostra que os clientes que utilizam automatizadores como esses atualizam suas dependências 1,6 vezes mais rápidos do que se fossem atualizados manualmente.

Finalmente, os pacotes clientes devem usar métodos alternativos para se recuperarem das *breaking changes* ao invés de simplesmente removerem/trocarem o provedor. Foi observada uma pequena porcentagem de clientes que removeram ou trocaram o provedor após uma *breaking change* (Descoberta #12). Essa baixa porcentagem é devido ao fato de que os clientes usam várias funcionalidades/recursos dos pacotes provedores e ter que remover/trocar o provedor pode ser uma tarefa muito custosa (ALRUBAYE; MKAOUER, 2018). Consequentemente, os clientes podem realizar um *downgrade* temporário para uma versão específica do provedor e verificar se isso impede a

manifestação da *breaking change*. Então, o cliente deve verificar continuamente por novas *releases* do seu provedor. Ainda, quando os clientes forem atualizar a versão de um provedor, os clientes devem verificar os *changelogs* dos provedores por alterações significativas para evitar alterações inesperadas. De fato, como verificado nesse trabalho, a maioria das *breaking changes* possuem documentação em *changelogs*, *issues*, *pull-requests* (Descoberta #8). Finalmente, os pacotes clientes devem sempre usar e incluir o arquivo *package-lock.json* para evitar atualização implícita inesperada nos provedores. Como no terceiro exemplo motivacional (Capítulo 3), quando havia o *package-lock.json*, o cliente não era impactado pela *breaking change*, mas quando o *package-lock.json* era removido, isso permitia que o npm atualizasse a versão dos provedores, introduzindo a *breaking change*.

6.2 Melhores práticas

Uma ferramenta muito útil que pode ser utilizada por desenvolvedores são os *linters*. Um *linter* realiza uma análise estática no código do desenvolvedor. Para linguagens dinâmicas como o Javascript, os *linters* podem ser muito úteis para evitar alguns tipos de erros (TóMASDóTTIR et al., 2017). Para Javascript, os *linters* mais utilizados são jslint, jshint e standard. Vários casos de *breaking changes* detectados e categorizadas nesse trabalho (Seção 5.3) poderiam ser evitados se os provedores usassem *linters*. As *breaking changes* classificadas como *Código incorreto* e *Renomeação de função* são facilmente capturadas por esses *linters*. Tómasdóttir et al. (2017) e Tómasdóttir et al. (2020) mostraram que um dos principais motivos que os desenvolvedores usam *linters* são para prevenir erros.

Entretanto, devido à natureza dinâmica do Javascript, *linters* não são capazes de verificar propriedades herdadas de objetos, então os erros classificados como *Alteração de funcionalidade*, *Alteração do tipo de objeto* e *Objeto indefinido* – ainda, *Renomeação de função* em funções das propriedades de objetos – não são capturadas pelos *linters*. Portanto, os desenvolvedores devem criar casos de testes para descobrirem esses erros em tempo de execução. Há vários *frameworks* disponíveis, tal como o mocha, chai, ava que realizam essas tarefas. Esses testes devem ser executados em integração contínua cada vez que os desenvolvedores realizarem um *commit* e enviarem ao repositório. Para isso, há várias ferramentas disponíveis, tal como o Travis, Jenkins, Drone CI e Codefresh. Ao utilizar *linters* e integração contínua, os desenvolvedores poderão descobrir a maioria dos casos de erros.

Os provedores devem estar conscientes sobre a quantidade de *commits* introduzida em cada *release*. Foi observado que mais da metade das *releases* dos provedores com *breaking changes* têm mais *commits* do que a mediana das *release* sem *breaking change* no mesmo nível *major* (Descoberta #4). Essas *releases* com *breaking changes* introduzem muitas alterações e isso pode ser difícil gerenciar todas essas alterações em apenas uma *release*: podem ser introduzidas mais alterações do que o planejado. Ou seja, algumas *releases* deveriam ser segmentada em duas ou mais *releases*. Portanto, os provedores devem manter as *releases* o mais concisas possível, introduzindo apenas *commits* consistentes, restringindo o número de *commits*.

Finalmente, os provedores – e qualquer outro pacote do ecossistema do npm – devem manter um *changelog*. Com um *changelog* consistente muitos erros poderiam ser facilmente rastreados e consertados. Foi observado que as correções em *breaking changes* que haviam sido documentadas em *changelogs*, *issues* ou *pull-requests* ocorrem 3,3 vezes mais rápido do que as *breaking changes* não documentadas (Descoberta #14). Assim, os provedores podem criar *templates* para as *issues* e *pull-requests* para permitir outros desenvolvedores a especificar detalhes mais consistentes e precisos nessas *issues* e *pull-requests* para facilitar o rastreamento das *breaking changes*.

6.3 Ocorrência das *breaking changes* e o Versionamento Semântico

Os casos *breaking changes* estão ocorrendo ao longo do tempo (Descoberta #3). Desta maneira, os desenvolvedores devem estar conscientes desse fenômeno e utilizar das técnicas/ferramentas disponíveis para mitigar o surgimento das *breaking changes*. Detectamos que a maioria das *breaking changes* são introduzidas por provedores indiretos, ou seja, são provedores que estão a partir do segundo nível na árvore de dependências. Esses provedores não são instalados diretamente pelos clientes, mas eles são instalados quando um provedor direto é instalado pelo npm. O trabalho de Decan et al. (2019) mostra que metade dos clientes no ecossistema do npm têm pelo menos 22 dependências transitivas (provedores indiretos), e um quarto possui pelo menos 95 dependências transitivas, em 2016. Esse grande e crescente número potencializa o surgimento de *breaking changes* pelos provedores diretos e indiretos.

Por fim, o Versionamento Semântico é apenas uma politica da qual os provedores podem decidir usar ou não (DECAN; MENS, 2021). Se os provedores não seguem as regras do Versionamento Semântico, muitas erros podem ser introduzidos. Foi observado que todas as *breaking changes* introduzidas em *pre-releases* foram propagadas para as *releases* estáveis (Descoberta #6). Portanto, os provedores podem estar desorientados acerca do Versionamento Semântico, mesmo sendo um conjunto simples de regras. Ainda, as regras do Versionamento Semântico descrevem as *breaking changes* apenas como *Alterações incompatíveis de APIs*,[1] mas foi verificado vários outros tipos de *breaking changes* que os provedores podem introduzir (Descoberta #5); alterações de *API* é apenas um subconjunto de *breaking changes*.

[1] https://semver.org/#summary

7 AMEAÇAS À VALIDADE

Neste capítulo descrevemos os detalhes sobre as circunstâncias que podem fazer com que os resultados variar.

Ameaças à validade interna

Quando uma *breaking change* era detectada, foi verificado o tipo da alteração que o provedor introduziu e causou a *breaking change* e elas foram categorizadas da maneira mais genérica possível. Entretanto, alguns casos se encaixariam muito bem em mais de uma categoria. Por exemplo, um provedor que alterou o tipo de objeto para alterar toda uma funcionalidade. Esse caso de *breaking change* poderia ser classificado como *Alteração de comportamento* e *Alteração do tipo de objeto*. Casos como esse foram categorizados na categoria mais expressiva da alteração. Nesse caso, o objeto foi alterado por causa de uma alteração no comportamento do pacote. Então, a categoria mais apropriada foi selecionada como *Alteração de comportamento*.

Os casos de erros do tipo *breaking change induzida* são aqueles que os pacotes usam dados de sites e *APIs*. Esses dados alteram-se ao longo do tempo (Ver Subseção 2.4.1) e os pacotes não têm controle sobre esses dados. As *releases* impactadas por *breaking changes* induzidas representam 9,3% das *releases* do cliente. Nesses casos, não foi verificado se havia ou não *breaking change*, uma vez que não foi possível executar os casos de testes. Então, quase 10% das *releases* dos clientes poderiam ser impactadas por *breaking change*, mas não foram analisadas.

Ameaças à validade externa

Os pacotes clientes foram selecionados de forma aleatória com uma variedade de características em termos da quantidade de *releases*, provedores e tamanhos. Por isso, a amostra é diversa e pode ser facilmente generalizada. Além disso, o método utilizado neste trabalho pode ser replicado para outros ecossistemas porque outros gerenciadores de pacotes funcionam da mesma maneira que o npm. Ainda, vários desses ecossistemas utilizam o Versionamento Semântico para gerenciar as dependências e a análise nos repositórios hospedados no GitHub não é exclusiva para o ecossistema do npm.

Porém, uma vez que a base de dados é exclusiva do npm, os resultados estão muito relacionados com esse ecossistema e podem não ser generalizados com fidelidade para os outros ecossistemas. O ecossistema do npm possui muitas diferenças substanciais dos outros ecossistemas (BOGART et al., 2016).

Ameaças à validade de construção

O método para detectar *breaking changes* realizou uma análise tão completa quanto possível no cliente e no provedor. Foram analisados os casos em que os testes do cliente falhou, indicando um real caso de *breaking change*. Foi analisado o código do cliente e do provedor, os *commits*, *changelogs*, *issues*, *pull-requests* e *logs* de serviços integrados.

Porém, a análise só foi realizada quando um teste falhou. Se um cliente usava uma *release* do provedor que continha uma *breaking change*, mas o cliente não realizava uma chamada para a função que introduzia a *breaking change*, então a *breaking change* não se manifestou no cliente e não gerou erro. Não havia como identificar *breaking change* nesses casos, pois só houve a análise quando houve um erro no teste do cliente. Assim, um fator importante para detectar *breaking changes* é a qualidade do *script* de teste do cliente, mas não foi analisada em nosso estudo.

Muitas das *breaking changes* relacionadas às *APIs* não foram detectadas. Basicamente, apenas foram detectadas casos no qual o provedor alterou o nome de uma *API*. Outros casos não foram identificados porque o Javascript permite realizar chamadas para *APIs* com qualquer número de parâmetros. Por exemplo, se um provedor removeu ou adicionou mais parâmetros à sua *API*, o cliente ainda era capaz de realizar uma chamada para essa *API* com os mesmos parâmetros que eram passados nas chamadas anteriores. Isso ocorre porque o Javascript insere todos os parâmetros dentro de uma variável chamada `arguments`[1] e não se preocupa com a quantidade de parâmetros.

Por último, para cada *release*, foram restaurados os arquivos da *release* utilizando a *tag* da respectiva *release* que o desenvolvedor criou. Foram listados todas as *tags* do repositório e foi realizado um *checkout* com a *tag* específica. Entretanto, para as *releases* que o desenvolvedor não criou uma *tag*, o *checkout* foi realizado utilizando o *timestamp* da *release* no *package.json*. Devido a necessidade de realizar o *checkout* usando o *timestamp*, foi verificado a confiabilidade dos *timestamps*. Em repositórios que haviam *tags*, 94% dos casos a *tag* e o *timestamp* correspondiam o mesmo *commit*.

[1] https://eloquentjavascript.net/03 functions.html#p kzCivbonMM

8 CONCLUSÕES

O reuso de código é um aspecto essencial no desenvolvimento de *software*, principalmente no ecossistema do npm, onde os pacotes criam uma imensa rede de dependências. Entretanto as *breaking changes* são uma consequência inevitável da facilidade que o reuso de código traz. *Breaking changes* e seus impactos são estudados na literatura em vários ecossistemas de *softwares* (BRITO, 2018; MØLLER; TORP, 2019; BOGART et al., 2016; YI et al., 2013). Poucos estudos examinaram as *breaking changes* no ecossistema do npm da perspectiva do pacote cliente, ou seja, executaram os casos de testes dos clientes para verificar o impacto das *breaking changes* (BOGART et al., 2015; MEZZETTI et al., 2018; MUJAHID et al., 2020). Nesse trabalho foi analisado as *breaking changes* no ecossistema do npm na perspectiva do pacote cliente e do pacote provedor.

Da perspectiva dos pacotes provedores foi analisado o quanto de *breaking changes* são introduzidas que impactaram os seus pacotes clientes. Foi concluído que 11,7% dos clientes são impactados por *breaking changes*. Também, foram analisados os erros mais comuns realizados pelos provedores e que causam as *breaking changes*. Foi verificado que o principal erro dos provedores é alterar seus comportamentos quando os clientes não esperam por essas alterações. Descobrimos que algumas alterações entre dois provedores podem os tornar incompatíveis e gerar uma *breaking change*. Finalmente, concluímos que o os casos de *breaking changes* está ocorrendo ao longo dos anos.

Da perspectiva dos pacotes cliente, analisados como eles se recuperaram das *breaking changes*. Quando uma *breaking change* é introduzida, os clientes se recuperam de várias maneiras. Observados que os pacotes clientes se recuperam das *breaking changes* em 39,1% dos casos realizando, principalmente, um *upgrade* na versão dos provedores. Esse método é o mais rápido e simples para se recuperarem, mas os clientes demoram mais tempo para se recuperarem de uma *breaking change* que os provedores. Ainda, as *breaking changes* são introduzidas principalmente por provedores indiretos. Os pacotes clientes, ao se recuperarem das *breaking changes*, preferem manter o *range*, apenas atualizando a versão de seus provedores. Finalmente, quando as *breaking changes* são documentadas, isso permite a recuperação dos clientes de maneira mais rápida.

Esse estudo contribui apresentando uma análise empírica sobre *breaking changes* em *releases minor* e *patch* no ecossistema do npm. Foram realizadas várias análises aprofundadas nos repositórios dos clientes e dos provedores para garantir que todos os casos de *breaking changes* são casos reais. As categorias criadas fornecem informações sobre os principais erros e os desenvolvedores podem usá-las como um guia para evitar tais erros em seus códigos. Finalmente, apresentamos várias sugestões sobre como os clientes e os provedores podem melhorar a qualidade de seus desenvolvimentos, prevenindo as *breaking changes*. Ainda, *pull-requests* foram criados nos repositórios dos pacotes corrigindo-os de erros e *breaking changes*.

Para trabalhos futuros, alguns pontos podem ser melhorados. Primeiramente, para detectar as *breaking changes* é necessário que o código de teste do cliente seja suficiente e eficiente para cobrir

o código do provedor e garantir que mais casos de *breaking changes* sejam detectadas. O trabalho de Mujahid et al. (2020) detectou que os testes dos clientes apresentam cobertura, em mediana, de 22% dos pacotes provedores. Também é importante detectar em qual *branch* o cliente publicou uma determinada *release* antes que os casos de testes da *release* sejam executados, pois nem todas as *releases* necessariamente são publicadas diretamente da *branch master* do repositório. Outro aspecto importante é sobre os casos em que o teste resulta em sucesso. Nesses casos, é interessante verificar se o teste foi realmente executado com sucesso e se o teste é um caso de teste real. Por exemplo, o desenvolvedor poderia escrever um *script* de teste como `{"test": "exit 1"}`. Nós detectamos que um dos fatores que influencia o surgimento das *breaking changes* é a quantidade de *commits* que é introduzida em uma *release*. Porém, podem haver outros fatores que provocam o fenômeno das *breaking changes*. Finalmente, a categorização dos casos de *breaking changes* podem ser hierarquizados para representar problemas mais genéricos. Por exemplo, as categorias *Alteração de tipo de objeto*, *Objeto indefinido*, *Renomeação de função* e *Arquivos não-encontrados* poderiam ser subcategorias de uma categoria *Violações de API*.

REFERÊNCIAS

ALRUBAYE, Hussein; MKAOUER, Mohamed Wiem. **Automating the Detection of Third-Party Java Library Migration at the Function Level**. In: *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*. Markham, Ontario, Canada: [s.n.], 2018. (CASCON'18), p. 60–71.

BOGART, Christopher; KäSTNER, Christian; HERBSLEB, James; THUNG, Ferdian. **How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems**. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Seattle, WA, USA: [s.n.], 2016. (FSE 2016), p. 109–120.

BOGART, Christopher; KÃ¤STNER, Christian; HERBSLEB, James. **When It Breaks, It Breaks: How Ecosystem Developers Reason about the Stability of Dependencies**. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. Lincoln, NE, USA: [s.n.], 2015. p. 86–89.

BRITO, Aline *et all.* **Why and how Java developers break APIs**. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Campobasso, Mulise, Italy: [s.n.], 2018. p. 255–265.

CHATZIDIMITRIOU, Kyriakos C.; PAPAMICHAIL, Michail D.; DIAMANTOPOULOS, Themistoklis; TSAPANOS, Michail; SYMEONIDIS, Andreas L. **Npm-miner: An Infrastructure for Measuring the Quality of the Npm Registry**. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. Gothenburg, Sweden: ACM, 2018. (MSR '18), p. 42–45.

DECAN, Alexandre; MENS, Tom. **What Do Package Dependencies Tell Us About Semantic Versioning?** *IEEE Transactions on Software Engineering*, v. 47, n. 6, p. 1226–1240, 2021.

DECAN, Alexandre; MENS, Tom; CLAES, Maelick. **On the Topology of Package Dependency Networks: A Comparison of Three Programming Language Ecosystems**. In: *Proccedings of the 10th European Conference on Software Architecture Workshops*. New York, NY, USA: ACM, 2016. (ECSAW '16, v. 21), p. 1–4.

DECAN, Alexandre; MENS, Tom; GROSJEAN, Philippe. **An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems**. *Empirical Software Engineer*, Kluwer Academic Publishers, USA, v. 24, n. 1, p. 381–416, Feb. 2019. ISSN 1382-3256.

FOO, Darius *et all.* **Efficient Static Checking of Library Updates**. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 2018. (ESEC/FSE 2018), p. 791–796.

KRAAIJEVELD, Michel. ***Detecting Breaking Changes in JavaScript APIs***. Dissertação (mathesis) — Dept. Soft. Tech., Delft University of Technology, Delft, Netherlands, September 2017.

MEZZETTI, Gianluca; ANDERS, Møller; MARTIN, Toldam Torp. **Type Regression Testing to Detect Breaking Changes in Node.js Libraries**. In: *Proc. 32nd European Conference on Object-Oriented Programming (ECOOP)*. Amsterdam, Netherlands: [s.n.], 2018. v. 7, p. 1–24.

MIRHOSSEINI, Samim; PARNIN, Chris. **Can automated pull requests encourage software developers to upgrade out-of-date dependencies?** In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Urbana-Champaign, Illinois: [s.n.], 2017. p. 84–94.

MØLLER, Anders; TORP, Martin Toldam. Model-based testing of breaking changes in node.js libraries. In: ***Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering***. Tallinn, Estonia: ACM, 2019. (ESEC/FSE 2019), p. 409–419. ISBN 978-1-4503-5572-8.

MUJAHID, Suhaib; ABDALKAREEM, Rabe; SHIHAB, Emad; MCINTOSH, Shane. **Using Others' Tests to Identify Breaking Updates**. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, 2020. (MSR '20), p. 466–476. ISBN 9781450375177.

RAEMAEKERS, Steven; DEURSEN, Arie van; VISSER, Joost. **Semantic Versioning versus Breaking Changes: A Study of the Maven Repository**. In: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. [S.l.: s.n.], 2014. p. 215–224.

TóMASDóTTIR, Kristín Fjóla; ANICHE, Maurício; DEURSEN, Arie Van. **The Adoption of JavaScript Linters in Practice: A Case Study on ESLint**. *IEEE Transactions on Software Engineering*, v. 46, n. 8, p. 863–891, 2020.

TóMASDóTTIR, Kristín Fjóla; ANICHE, Mauricio; DEURSEN, Arie van. **Why and how JavaScript developers use linters**. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2017. p. 578–589.

YI, Jooyong; QI, Dawei; TAN, Shin Hwei; ROYCHOUDHURY, Abhik. **Expressing and Checking Intended Changes via Software Change Contracts**. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2013. (ISSTA 2013), p. 1–11.

ZHANG, Yang *et all*. **Within-ecosystem Issue Linking: A Large-scale Study of Rails**. In: *Proceedings of the 7th International Workshop on Software Mining*. New York, NY, USA: ACM, 2018. (SoftwareMining 2018), p. 12–19.