BREAK THE CODE? BREAKING CHANGES AND THEIR IMPACT ON SOFTWARE EVOLUTION

THESIS

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op woensdag 29 maart 2023 om 16:00 uur

door

LINA MARÍA OCHOA VENEGAS

geboren te Bogotá, Colombia

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

Voorzitter:	prof.dr. J.J. Lukkien
Promotoren:	prof.dr. J.J. Vinju
	prof.dr. M.G.J. van den Brand
Co-promotor:	dr. T.F. Degueule (Université de Bordeaux, CNRS, LaBRI)
Overige leden:	prof.dr. B. Baudry (KTH Royal Institute of Technology)
	dr. K. Blincoe (University of Auckland)
	prof.dr. A. van Deursen (Technische Universiteit Delft)
	prof.dr. T. Mens (Université de Mons)
	prof.dr. A. Serebrenik

Het onderzoek of ontwerp dat in dit thesis wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

The work in the thesis has been carried out under the auspices of the research school Institute for Programming research and Algorithmics (IPA). Thesis number 2023-02.

A catalogue record is available at the Eindhoven University of Technology (TU/e) Library.

ISBN 978-90-386-5688-5

Para *Claudia*, *Carlos* e *Isabella*.
Mi entonces, ahora y para siempre.

SUMMARY

Software seldom lives in isolation. Instead, projects dwell in software ecosystems where they depend on each other to favour reuse. Software projects have a dual role: (i) the *library* role when exposing a set of services to other projects, and; (ii) the *client* role when depending on other libraries to leverage their functionality.

As time passes, library developers introduce changes to include functional and extra-functional enhancements. Although changes aim at increasing the library's value, they might propagate to client projects resulting in broken code. Versioning schemes—such as semantic versioning—are often used to communicate the nature of introduced changes namely, changes that potentially break (or not) client code. Nevertheless, library developers still face a dilemma: whether to introduce changes at the cost of increasing the technical lag on their clients or even losing them; or avoid change at the cost of increasing technical debt. This thesis states that Breaking Changes (BCs) are not harmful by themselves. However, their impact should be first assessed, so developers can make informed decisions on their introduction and subsequent coping strategies.

In this thesis, we address the library-client co-evolution problem from the nounal and the verbal views. On the one hand, the *nounal* view allows us to empirically understand the nature of the library-client co-evolution phenomenon. In particular, we study (i) best practices to define dependencies as a way of preventing the propagation of BCs, and; (ii) syntactic BCs and their impact on client projects in relation with semantic versioning.

On the other hand, the *verbal* view encourages us to provide new processes, methods, and tools that can better support the library-client co-evolution process. Concretely, we introduce (i) the *static impact analysis* approach to detect BCs introduction and their impact on client code, and; (ii) the *static reverse dependency*

compatibility testing approach to perform static impact analysis as part of a pull-based development workflow. The former is implemented in Maracas, a static analysis tool for Java projects, and the latter is implemented in BreakBot, a GitHub bot that assists library evolution.

As the main conclusions of the thesis, we find that: (i) Practitioners do not widely follow best practices when defining dependencies. (ii) Libraries tend to comply with semantic versioning when introducing syntactic BCs; the adherence to such scheme has increased over time, and; only a few clients are impacted by these changes. (iii) Tooling to support software evolution is accurate, applicable, and relevant for pull-based development workflows.

SAMENVATTING

Softwareprojecten zijn vaak onderdeel van grote software ecosystemen met veel onderlinge afhankelijkheden, onder meer vanwege het hergebruik van software bibliotheken. Softwareprojecten hebben vaak een dubbele rol: (i) enerzijds dienen ze als bibliotheek, ze bieden functionaliteit aan andere projecten, en; (ii) anderzijds hebben ze zelf een klant rol, de software is afhankelijk van andere bibliotheken om de vereiste functionaliteit te implementeren.

In de loop van der tijd evolueren bibliotheken; software ontwikkelaars passen de bibliotheken aan om functionele en extra-functionele verbeteringen door te voeren. Het doel van deze veranderingen is om de bruikbaarheid van de bibliotheek te vergroten. Maar hebben deze wijzigingen tegelijkertijd ook effect op de projecten van de klanten en kunnen de veranderingen leiden tot broncode die niet meer voldoet of zelfs niet meer werkt, zogenaamde "brekende wijzigingen" of "Breaking Changes (BCs)". Versiebeheer—zoals semantisch versiebeheer—wordt vaak gebruikt om de aard van de geïntroduceerde veranderingen inzichtelijk te maken, namelijk veranderingen die mogelijk negatieve effecten op de broncode van de klant hebben. De ontwikkelaars van bibliotheken staan daarom voor een dilemma: moeten ze veranderingen introduceren ten koste van het veroorzaken van (grotere) veranderingen in de broncode van hun klanten of zelfs het kwijtraken van hun klanten; of het vermijden van de veranderingen ten koste van het verhogen van technische schuld ("technical debt"). Dit proefschrift laat zien dat een verandering die effect heeft op de broncode van de klanten op zichzelf geen ramp hoeft te zijn. Het effect van deze veranderingen moet echter eerst in kaart worden gebracht, zodat ontwikkelaars onderbouwde beslissingen kunnen nemen over het wel of niet introduceren van

veranderingen, de consequenties en de daarbij horende strategieën.

In dit proefschrift wordt het bibliotheek-klant co-evolutie probleem vanuit nominaal en verbaal perspectief bestudeerd. Enerzijds stelt het *nominale* perspectief ons in staat om de aard van het bibliotheek-klant co-evolutie fenomeen empirisch beter te begrijpen. In het bijzonder, bestuderen we (i) de meest effectieve huidige werkmethodes, ook wel best practices genoemd, om afhankelijkheden te definiëren om verspreiding van brekende wijzigingen te voorkomen, en; (ii) syntactisch brekende wijzigingen en het effect van deze wijzigingen op de software van klanten in relatie tot semantische versiebeheer.

Anderzijds spoort het *verbale* perspectief aan om nieuwe processen, methoden, en gereedschappen te ontwikkelen die het bibliotheek-klant co-evolutie proces beter kunnen ondersteunen. Daarom introduceren wij: (i) een statische impact analyse methode om brekende wijzigingen, en de effecten op software van de klant, te detecteren, en; (ii) een statische comptabiliteit test methode om omgekeerde afhankelijkheden te detecteren als onderdeel van een pull-gebaseerd ("pull-based") manier van werken. De eerste methode is geïmplementeerd in Maracas, een statisch analyse tool voor Java projecten, en de tweede methode is geïmplementeerd in BreakBot, een GitHub bot die bibliotheekevolutie ondersteunt.

De belangrijkste resultaten en conclusies van het proefschrift zijn: (i) gebruikers passen in het algemeen niet de best practices toe bij het definiëren van afhankelijkheden; (ii) bibliotheken gebruiken over het algemeen semantische versiebeheer wanneer er syntactisch brekende wijzigingen worden geïntroduceerd; dit wordt in toenemende mate toegepast, en; een beperkte aantal klanten heeft last van deze veranderingen; (iii) gereedschappen ("tools") om software evolutie te ondersteunen is accuraat, toepasbaar, en relevant voor pull-gebaseerd manier van werken.

ACKNOWLEDGEMENTS

This was the last chapter I wrote for this thesis. However (or consequently), I decided to place it at the beginning of the series. It appears in this order for a reason: It is here where I have the opportunity to, in the shape of small fragments, thank all the people that have either prepared me for or joined me in this journey. I warn you, the words won't be sufficient, meanings will be unfairly reduced to a few lines, and much to my regret, this is the best way I found to honor your lives into mine.

Our story started in a rainy month. I call it rainy because in our land there are no seasons. Since then, we have shared other storms. "We have shared" as if this was the expected outcome. Today, three decades after that beginning, I understand that "being always there" was your constant decision. Mom, you have never let me down. Every time I have fallen, you have been there to reach out and help me stand. I hope life will allow me to love with the devotion you love, to face adversities with the braveness you face them. Thanks for showing me that joy and fun are not supposed to disappear with age. Dad, your calm speech hides tenderness and wisdom. You have taught me that one shall do what one loves, that it shall be done with commitment, and that surrender is never an option. Thanks for teaching me the language of silence and caress. To you two that always chose to be without saying, to you that have given everything without asking for anything, to you my absolute love, admiration, and gratitude. This achievement is as much yours as mine.

Isa, my little sister and friend, thanks for constantly challenging me, for showing me that the own voice should never be silenced, and for lending me yours when mine has failed. Thanks also for proving me wrong when I feared for your early decisions, for your betting on everything, and for not being afraid of anything. Thanks for making me see that fear should, under no

circumstance, define our limits. Knowing that I can always come to you when life turns serious fills my heart with happiness and relief. Tata, my daisy, it was you who taught me how to celebrate life, to sing our joys and sorrows to the sound of cumbias and mariachi. You instilled in us that crazy idea of betting on dreams and that, while life allows it, there will always be an excuse to toast and cheer. Uncle, you have the gift of sowing smiles in people's hearts, of not leaving the essentials unsaid. Thanks for the right word in the proper moment, for the jokes and the laughter, for The Beatles and music. Juancho and Caro, thanks for being part of the illusion of coming back home. Your brotherhood has always been complete (if you know what I mean). To all of you who I carry in my heart, to Nubia, Mari, Juanda, Roci, Joha, Alejo, Cami, to you whom I call family, from the deepest part of my soul, thanks.

Okan, sharing the present with you has been a beautiful surprise. A present that sounds like bağlama, blues, and boleros. You showed me the now when I wandered between the fears of the past and the uncertainties of the future. Thanks for the days filled with laughter, for the pul biber and the butter, for the peace that comes from tenderness and understanding, for propelling me to be a better version of myself. You were my company and shelter in this last chapter, and you are already part of the sketches of the next. Campi, already for a long time the word "friend" fell short. You have been to me bliss and shield. Disappointments have never been part of your language. Despite the fact that for years kilometers have stood between us, there has not been a day where I have not felt you close. For your unconditional and unshakable friendship, for warming up my soul, thank you. Dieguis, people should not get confused with that humor that is so yours. Beneath the surface, one can find a person with principles, an unquestioningly loyal friend. For taking me to discover the unknown, for the creamy lasagna, for your honesty and authenticity, for the late-night conversations, thank you. To Natis Torres who has never stopped being present and surprising me, to my life and childhood friends Natis Sofi, Dianis, Gabi, Juli, Mari, Lulu, Cata, and Ale, to you all thanks

for more than twenty years of being here. Seeing you will always make me feel at home.

Cami Sánchez, for years you have been a sister to me. We grew up together, we shared our families, and, after the ups and downs of life, we always found a way to come back to each other. I want to let you, Ceci, and Meji know that when the melancholy and the memories pay a visit, knowing you close has been a consolation. I find a deeper level of understanding in our conversations, an understanding that only comes to those who have adventured to leave that place they call home. Roquita, how much joy you have brought to my life. You taught me how to laugh more, and that the minute spent with a friend will always be well-invested. Recently, the discussions with Caro and you have been a source of enjoyment and inspiration. Thanks to you two for the banyat and the chocolate plumcake, for the fits of laughter, and for the rice pudding. In these foreign lands, you are the friendly sight, the heartfelt hug, the nearby family. To you all, thanks for being here today.

Verni, we gave the first steps of this adventure together. I thank you for years of the closest friendship. It seems that the road has finally ended and I remain here with an immense gratitude for the sincere affection, the learned lessons, and the new and old dreams. How much I learned from you. Thanks for who you were in my life and for all the beautiful memories, I will always keep them in my heart. Jesse and Hans, you do not know each other but I find so much in common between you both. My friends of life, contemplation, and perspective. In your advice, I always find a hidden truth and an authentic desire to have a heart a little bit more similar to yours. Anis, thanks for the words, your honesty, and your sweetness. You boosted my energy when it was missing. Juanpis, thanks for the sudden calls in moments when events have requested them, the thoughtful words, the pizza in Libery, and the landscape of the Po. George, thanks for your trust, the calculus classes, and the trip to Paris. Your friendships have been a constant in my life. Thanks for still being here.

Felipe (garotinho) and Mila, although at the beginning we were nothing more than strangers, you opened a space for me in your

routine, and you gave me smiles that were scarce at the time. Thanks for accompanying me in giving a difficult step, and for holding me tight while I was trembling. You must know that I keep a big affection and gratitude for you. Paola and Francesca, I am almost certain you don't know it, mainly because I never found the space to share it, but your friendships changed me deeply. La May, we reconnected after ten years in a remote city, more than 8,000 kilometers away from where we originally met. You taught me to question what has been learned, to unlearn the unnecessary, and to accept the other one and our destiny. Thanks for the deformed buñuelos and the arepitas, for the jogging techniques and the thoughtful bike rides, for showing me how to shift our limits. Franci, I told you once and I don't mind repeating it, you brought joy and confidence to my life. I won't ever forget the days you walked by my side, the tiramisú and the risotto during pandemic nights, the laughter just before the curfew, and the 60-kilometer bike ride to Den Bosch. You taught me to pay attention to details, raise my gaze and discover the infinite. Thanks for the friendship of gestures and feats that you gave me. Evelina, my chica of mimosas for breakfast and gin and tonics on Sunday afternoons. In lonely times, your cautious presence on the other side of the wall nourished me with company and comfort. Merve, Maria, and Hamza, my friends of letters and reflection. Thanks for being an affectionate and welcoming face in an unknown city.

Mazyar, we met during the pandemic, since then and until now we have been a team. The little giants. You have been to me a committed friend, a colleague that helps me see the objective from unforeseen angles. Thanks for the laughter, your trust, your peculiar logic (yes you ARE logic), the sour gummies and the teacakes, for showing me that professionalism can and should be a synonym for fun. David (cuatecito), I have found in you a loyal friend. The conversations about our similar cultures make me feel closer to home. Thanks for listening, for the good advice and the kind heart, for the chilaquiles and the homemade beer. Hope to have you around for longer. Gijs, do you remember the early morning of that Thursday when we were fighting against sleep

to get the work done? You did not need to be there, nevertheless, you accompanied me until we falsely believed we were done with the task. That day and many others you showed me that thing that is naturally forged when building a shared project. Thanks for being that sweet version of the Dutch culture, for not letting the other one down, for finding motivation in the people and not just in the labour. To you friends who taught me to find joy in doing what one loves, thank you.

Óscar, you shared your knowledge, taught me to appreciate this profession, and pushed me to follow this dream. Thank you because this decision started with you. Conchita, my history teacher. You inspired me to discover and tell new stories. You showed me that memory has the ability to save everything that the soul starts owning. My dear Paul, thanks for the late afternoon talks about work and life, happening just after the offices started getting emptied. Your humbleness, regardless of being the greatest of all the ones I have met, will always inspire me. Nico Jiménez, we met on street 72, in the black building at the corner of seventh avenue. While we toured Bogotá, you taught me all about the technical and humans aspects behind our labour. Jenny, you listened to me for hours following firsthand the events that were happening in my life. Thanks for always posing the right question. Your intelligence and your passion for what you do are inspiring to me. To you all, thanks for showing me the direction.

From CWI I thank Aiko, Bikkie, Bert, Davy, Esteban, Felipe, Jouke, Irma, Muriel, Nikos, Pablo, Riemer, Rodin, Remko, Rob, Sussane, Thomas van Binsbergen, Tijs, Tim, and Ulyana for the smiles and conversations. You were part of the first half of this road. From TU/e I thank Agnes, Alexander, Erik Scheffers, Hossain, Jacob, Kees, Lars, Loek, Nathan, Maurice, Michel, Rick, Samar, Satrio, Tukaram, Weslley, and Zahra for the cozy working environment. Working with you has been a source of joy. Sangeeth, Nan, and Priyanka, our processes coincided in this last stage. The nervous motivational discussions about our Ph.D. made evident our humanity and vulnerability, that, despite and thanks to it, we are not alone. Jean-Rémy, Harold, Tom Verhoeff,

Eleni, Juliana Alves, Thomas Thüm, the teams from L'Aquila and the VUB, thanks for contributing to my research and learning process. To my Ph.D. committee, thanks for the time you took to carefully review this thesis and share useful questions and feedback. I also thank Cor, Dani, Erik Takke, Guillermo, Karina, Kasra, Niels, and Nora for inspiring me, for making me question my own methods, for renewing the energy and enthusiasm that sometimes might get lost with time. In particular, I thank Erik for teaching me that we should always "expect the best from people and the worst from their circumstances".

In 2015, three strangers crossed paths in an event hall in a hotel in Pittsburgh. Life would make them meet again two years later, this time in a building of red frames and green sunshades. Jurgen the story followed with an interview in winter and an offer in spring. I am not sure if I ever told you but that offer meant to me the happiness and fear of starting over. Thanks to that, I am, almost six years later, still writing that same story. From you I learned that a person is never too important to lose their kindness, that mastery comes with experience, and that it is okay to not always be fine. Thomas, you were the other person in that hall. I will always admire your intelligence and the (almost obsessive) thoroughness you use to do your job. Thanks for your commitment and every single invested minute, for believing, for pushing and challenging me, and for generously sharing your knowledge. When times were difficult you offered me your hand without even waiting for a call. In this journey, you have been to me not only a guide but foremost a friend. For your dedication, and endless patience, thank you. Mark, I met you at a later stage. You believed in me when I was doubting. I confess that it was your faith the one that propelled me to do what I considered impossible, to reach what I thought unreachable. Sometimes we only need someone to believe in us to start shining. You were that someone in my life. You have been like a father and guardian to me and many people that have had the fortune to work with you. I was lucky to have the three of you as mentors. For the invested time, guidance, and respect, my sincere and deepest gratitude.

With these words, I close up this chapter, a chapter that lasted almost six years. I know that during these times, we did not always smiled (although there were several occasions). However, I can say without hesitation that each minute was well lived, that every step was well taken. Life will take its course, new goals will come, maybe we will meet again, maybe we will say goodbye, but to wherever the route takes us, your imprint will remain forever in my heart. To all of you who have been part of this journey and who have shaped who I am, again and from the depths of my heart, THANK YOU.

AGRADECIMIENTOS

Fue este el último capítulo que escribí para este libro y, sin embargo (o en consecuencia), decidí situarlo al comienzo de la serie. Aparece en este orden por una razón: es aquí donde tengo la oportunidad de, a modo de pequeños fragmentos, agradecer a esas personas que bien me prepararon para este camino o me acompañaron en su recorrido. Advierto de antemano que las palabras no serán suficientes, que los significados serán reducidos injustamente a unas pocas líneas, y que, muy a su pesar, es esta mi manera de honrar sus vidas en la mía.

Nuestra historia comenzó en un mes de lluvias. De lluvias porque en nuestra tierra no hay estaciones. Desde entonces y hasta hoy hemos compartido otros temporales. "Compartido" como si eso fuera lo esperado. Hoy, tres décadas después de ese comienzo, entiendo que el "siempre estar" fue su elección constante. Mamá jamás me has faltado. Cada vez que he caído has estado allí para tenderme la mano y levantarme. Que la vida me permita amar con la intensidad y entrega con que tú amas, que me permita enfrentar la adversidad con la valentía con que tú la enfrentas. Gracias por mostrarme que la alegría y la diversión no se pierden con los años. Papá, cuánta ternura y sabiduría esconde tu hablar pausado. Me enseñaste que uno hace lo que ama, y que lo hace uno con entrega, y que uno no se rinde porque rendirse no es una opción. Gracias por enseñarme ese lenguaje de silencios y caricias. A ustedes que eligieron siempre estar sin decirlo, a ustedes que lo han dado todo sin pedir nada, para ustedes mi absoluto amor, admiración y gratitud. Este logro es tan suyo como mío.

Isa, mi hermanita y amiga, gracias por nunca dejar de retarme, por mostrarme que la propia voz no debe ser silenciada, y por prestarme la tuya cuando la mía faltó. Gracias por mostrarme cuán equivocada estaba cuando temí por tus decisiones tem-

pranas, por tu apostarle a todo y no temerle a nada. Gracias por enseñarme que el miedo no debe, bajo ninguna circunstancia, definir nuestros límites. Saber que puedo volver a ti siempre que la vida se torne seria me llena el corazón de alegría y alivio. Tata, mi margarita, fuiste tú quien me enseñó a celebrar la vida, a cantar las penas y alegrías al son de cumbias y mariachi. Inculcaste en todos esa tendencia casi loca de apostarle a los sueños, y de que mientras la vida lo permita siempre habrá una excusa para brindar y vestir de gala. Tío, qué don el tuyo de sembrar sonrisas en los corazones, de no dejar lo importante sin ser dicho. Gracias por la palabra indicada en el momento oportuno, por las ocurrencias y carcajadas, por los Beatles y la música. Juancho y Caro, gracias por ser parte de esa ilusión de volver a casa. Su hermandad siempre ha sido completa, nunca a medias, si entienden a los que me refiero. A todos ustedes que atesoro y admiro, a Nubia, Mari, Juanda, Roci, Joha, Alejo, Cami, a ustedes a quien llamo familia, desde lo más profundo de mi alma, gracias.

Okan, qué linda sorpresa ha sido compartir el presente contigo. Un presente que suena a bağlama, blues y boleros. Me mostraste el ahora cuando deambulaba entre los miedos del pasado y la incertidumbre del porvenir. Gracias por los días llenos de risas, por el pul biber y la mantequilla, por la paz que trae la dulzura y el entendimiento, por impulsarme a ser una mejor versión de mí. Fuiste mi compañía y abrigo en este último capítulo y eres ya parte de la ilusión del siguiente. Campi, desde hace mucho la palabra "amiga" se quedó corta. Tu amistad ha sido para mí, dicha y refugio. Las decepciones nunca han tenido cabida contigo. A pesar de que por años los kilómetros se han interpuesto entre las dos, no ha habido un solo día en el que no te haya sentido cerca. Por tu amistad incondicional e inquebrantable, por calentar mi alma cuando hacía frío, gracias. Dieguis, que no se deje confundir la gente con ese humor que es tan tuyo. Bajo la superficie encuentra uno una persona de principios, un amigo incondicional. Por llevarme a conocer lo desconocido, por la lasagna "cremosita", por tu honestidad y autenticidad, por las conversaciones de madrugada, gracias. A Natis Torres que no ha dejado de estar presente y sorprenderme, a mis amigas de la

vida y de la infancia, Natis Sofi, Dianis, Gabi, Juli, Mari, Lulu, Cata y Ale, a ustedes gracias por seguir estando después de más de veinte años. Verlas siempre me hará sentir en casa.

Cami Sánchez, has sido desde hace años una hermana para mí. Crecimos juntas, compartimos nuestras familias y siempre volvimos a la otra después de los vaivenes de la vida. Quiero hacerte saber a ti, a Ceci y a Meji que cuando la melancolía y las memorias visitan saberlas cerca ha sido un consuelo. Son aquí mi terreno conocido. Encuentro en nuestras conversaciones una capa adicional de entendimiento, que sólo el que se aventura lejos puede entender. Roquita, cuánta alegría has traído. Me enseñaste a reír más, que el minuto con un amigo siempre será bien invertido. La tuya ha sido una amistad constante. Recientemente, el tiempo y las conversaciones con Caro y contigo han sido fuente de goce e inspiración. Gracias a los dos por el banyat y el plumcake de chocolate, los ataques de risa y el arroz con leche. Son ustedes aquí, en estas tierras extranjeras, la mirada amiga, el abrazo sentido, la familia cercana. A los cinco, gracias por estar hoy.

Verni, los primeros pasos de esta aventura los dimos juntos. A ti te agradezco años de la más estrecha amistad. Parece que finalmente ha terminado el trayecto y hoy me queda una inmensa gratitud por el cariño sincero, las lecciones aprendidas y los viejos y nuevos sueños. Cuánto aprendí de ti. Gracias por quién fuiste en mi vida y por las lindas memorias, siempre las atesoraré en mi corazón. Jesse y Hans, los dos no se conocen y, sin embargo, tienen ustedes para mí tanto en común. Mis amigos de la vida, de reflexiones y perspectiva. En su abrazo y consejo siempre encuentro una verdad oculta y un deseo sincero de tener el corazón un poquito más como el suyo. Anis, gracias por las palabras, tu honestidad y tu dulzura. Recargaste mi energía cuando carecía de ella. Juanpis, gracias por las llamadas repentinas cuando los eventos lo han requerido, por las palabras sentidas, por la pizza en Libery y el paisaje del Po. George, gracias por tu confianza, las clases de cálculo y el viaje a París. Su amistad ha sido una constante en mi vida, gracias por seguir estando.

Felipe (garotinho) y Mila, a pesar de que en un comienzo no éramos más que extraños, abrieron ustedes un espacio para mí en su rutina, me regalaron sonrisas que para cuando nos conocimos escaseaban. Gracias por acompañarme a dar un paso difícil, por sostenerme mientras tambaleaba. Sepan que guardo por ustedes un inmenso cariño y gratitud. Paola y Francesca, probablemente no lo sepan porque nunca busqué la oportunidad para decirlo, pero su amistad me cambió profundamente. La May, nos reencontramos después de diez años en una ciudad remota a más de 8.000 kilómetros de donde nos conocimos. Tú me enseñaste a cuestionar lo aprendido, a desaprender lo innecesario y sobre todo a aceptar al otro y al destino. Gracias por los buñuelos deformes y las arepitas, por las técnicas de jogging y los viajes contemplativos en bicicleta, por mostrarme cómo desdibujar los límites. Franci, te lo dije una vez y te lo repito hoy, trajiste a mi vida dicha y confianza. No olvidaré los días que caminaste a mi lado, el tiramisú y el risotto en noches de pandemia, las risas antes del toque de queda, el paseo en bicicleta de 60 kilómetros a Den Bosch. Me enseñaste a fijarme en los detalles, a subir la mirada y descubrir el infinito. Gracias por la amistad de gestos y locuras que me regalaste. Evelina, mi chica de mañanas con mimosa y domingos de gin and tonics. En épocas solitarias tu presencia cautelosa al otro lado del muro me brindó compañía y bienestar. Merve, Maria y Hamza mis amigos de letras y reflexión. Gracias por ser un rostro amigo en una ciudad desconocida.

Mazyar, nos conocimos en pandemia, desde entonces y hasta ahora hemos sido un equipo. Los pequeños gigantes. Has sido para mí un amigo entregado y comprometido, un colega que me hace ver el objetivo desde rincones que no había contemplado. Gracias por las risas, por tu confianza, por tu tan peculiar lógica (sí, ERES lógica), por las gomitas ácidas y los teacakes, por mostrarme que profesionalismo puede y debe ser sinónimo de diversión. David (cuatecito) en tí he encontrado a un amigo leal. Las charlas sobre nuestras culturas tan cercanas me hacen sentir en casa. Gracias por escuchar, por el buen consejo y un corazón bondadoso, por los chilaquiles y la cerveza hecha en casa. Espero

poder seguir teniéndote cerca. Gijs, ¿recuerdas esa madrugada del jueves luchando contra el sueño para terminar el trabajo? Tú no tenías que estar ahí, pero me acompañaste hasta que falsamente creímos terminar con la tarea. Ese y muchos otros días me mostraste eso que naturalmente se forja cuando se construye un proyecto compartido. Gracias por ser esa versión dulce de la cultura holandesa, por no fallarle al otro, por encontrar motivación en las personas y no sólo en la labor. A ustedes amigos que me enseñaron a encontrar el goce en hacer lo que uno ama, a ustedes gracias.

Óscar, compartiste tu conocimiento, me enseñaste a apreciar esta labor y me empujaste a seguir el sueño. Gracias porque esta decisión empezó contigo. Conchita, mi profesora de historia. Me inspiraste a descubrir y contar historias, me mostraste que la memoria tiene la habilidad de guardar todo aquello que uno apropia. Mi querido Paul, agradezco las charlas sobre el trabajo y la vida al final de la tarde cuando las oficinas se desocupaban. Tu humildad, muy a pesar de ser el más grande de todos los que he conocido, siempre me inspirará. Nico Jiménez, nos conocimos en la 72, en un edificio negro justo en la esquina de la séptima. Mientras recorríamos Bogotá, me enseñaste el aspecto técnico y humano que se esconde tras nuestra labor. Jenny, me escuchaste por horas siguiendo de primera mano lo que acontecía en la vida. Escuchaste con atención mi historia y formulaste siempre la pregunta precisa. Tu inteligencia y la pasión por lo que haces son una fuente de inspiración para mí. A ustedes maestros de la vida, gracias.

Del CWI agradezco a Aiko, Bikkie, Bert, Davy, Esteban, Felipe, Jouke, Irma, Muriel, Nikos, Pablo, Riemer, Rodin, Remko, Rob, Sussane, Thomas van Binsbergen, Tijs, Tim y Ulyana por las risas y conversaciones. Ustedes fueron parte de la primera mitad de este trayecto. De TU/e agradezco a Agnes, Alexander, Erik Scheffers, Hossain, Jacob, Kees, Lars, Loek, Nathan, Maurice, Michel, Rick, Samar, Satrio, Tukaram, Weslley y Zahra por un ambiente acogedor. Trabajar con ustedes ha sido una fuente de alegría. Sangeeth, Nan y Priyanka, nuestros procesos coincidieron en esta última etapa. Las nerviosas charlas de motivación del doctor-

ado nos recordaron nuestra humanidad y vulnerabilidad, que a pesar de y gracias a ellas no estamos solos en el recorrido. Jean-Rémy, Harold, Tom Verhoeff, Eleni, Juliana Alves, Thomas Thüm, y los equipos de L'Aquila y de VUB, gracias por contribuir en el aprendizaje de mi labor. Cada uno de ustedes me dejó una lección. A mi comité doctoral, gracias por el tiempo que tomaron para revisar esta tésis y compartir su retroalimentación y preguntas valiosas. También agradezco a Cor, Dani, Erik Takke, Guillermo, Karina, Kasra, Niels y Nora por inspirarme, por hacerme cuestionar mis propios métodos, por renovar en mí esa energía y entusiasmo que a veces se pierde con los años. En particular, agradezco a Erik por enseñarme a "esperar lo mejor de las personas y lo peor de sus circunstancias".

En 2015, tres extraños cruzaron su camino en la sala de eventos de un hotel en Pittsburgh. La vida volvería a reencontrarlos dos años después en un edificio de marcos rojos y parasoles verdes. Jurgen, la historia continuó con esa entrevista corta en invierno y una invitación en primavera. No sé si algún día supiste o si me tomé la molestia de decirte, pero esa invitación significó para mí la alegría y el temor de iniciar de nuevo. Gracias porque fue por ese "sí" que hoy, casi seis años después, sigo escribiendo esa historia. De ti aprendí que nunca se es lo suficientemente grande para perder la dulzura, que la maestría viene con la experiencia, y que está bien no siempre estarlo. Thomas, fuiste la tercera persona de este cuento. Siempre admiraré tu inteligencia y esa rigurosidad (casi obsesiva) con la que haces tu trabajo. Gracias por la entrega y cada minuto invertido, por creer, por compartir conmigo tu saber sin ningún recelo. Cuando los tiempos fueron difíciles ofreciste tu mano incluso antes de escuchar el llamado. Fuiste para mí en este camino no sólo una guía, sino sobre todo un amigo. Por tu entrega e infinita paciencia, gracias. Mark, a ti te encontré un poco más tarde. Creíste en mí en un momento en el que dudaba. Confieso hoy que fue tu fe la que me impulsó a hacer lo imposible, a alcanzar lo inalcanzable. A veces sólo se necesita que alguien crea en el otro para brillar, tú fuiste ese alguien en mi vida. Has sido como un padre y protector para muchos de los que hemos tenido la suerte de trabajar contigo. Fui afortunada al

tenerlos a los tres como mentores. Por el tiempo, por los consejos, por el respeto, mi más sentida y profunda gratitud.

Con estas palabras finalmente cierro este capítulo, un capítulo que duró casi seis años. Sé que en este tiempo no todo fueron sonrisas (aunque sí que hubo bastantes). Puedo, sin embargo, decir con toda certeza que cada minuto fue bien vivido, que cada paso estuvo bien dado. La vida seguirá su curso, vendrán nuevos objetivos, quizás nos reencontraremos, nos despediremos, pero hacia donde sea que nos lleve el camino, su huella permanecerá por siempre en mi corazón. A todos ustedes que han hecho parte de mi vida y han moldeado quien soy hoy, nuevamente y desde lo más profundo de mi corazón, GRACIAS.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ADT	Algebraic Data Type
API	Application Programming Interface
AST	Abstract Syntax Tree
BENEVOL	Belgium-Netherlands Software Evolution Workshop
BC	Breaking Change
CAU	Conditional API Usage
CD	Continuous Development
CI	Continuous Integration
CNRS	Centre National de la Recherche Scientifique
CWI	Centrum Wiskunde & Informatica
ICSE	International Conference on Software Engineering
IDE	Integrated Development Environment
IoC	Inversion of Control
IPA	Institute for Programming research and Algorithmics
JAR	Java ARchives
JDK	Java Development Kit
JLS	Java Language Specification
JNI	Java Native Interface
JPMS	Java Platform Module System
JVM	Java Virtual Machine
KOSP	KT OSGi Service Platform
LSP	Liskov Substitution Principle
MCR	Maven Central Repository
MDD	Maven Dependency Dataset
MDG	Maven Dependency Graph
NIER	New Ideas and Emerging Results
OSGi	Open Service Gateway Initiative
OSS	Open-source Software
POM	Project Object Model
PR	Pull Request

RDCT Reverse Dependency Compatibility Testing
REST REpresentational State Transfer
SET Software Engineering and Technology
TSE IEEE Transactions on Software Engineering
TU/e Eindhoven University of Technology
VCS Version Control System

Part I

THE ORIGIN

On the content of this thesis and its background

1

INTRODUCTION

Designing software for reuse and change has been an abiding quest in software engineering. Parnas introduced the principle of *information hiding* as a means to achieve this goal. The principle states that implementation details and changeable information should be hidden and encapsulated [126, 127]. On the one hand, the application of the information hiding principle opens the door to software modularisation—that is, software is split up into modules that offer a set of well-defined services via an interface [19]. Modules can then favour reuse by establishing *dependencies* among each other. On the other hand, information hiding facilitates maintenance by allowing developers to make changes to the implementation code without impacting the interface. By avoiding the introduction of such changes in the interface, developers prevent ripple effects that might impact other modules.

Software modularisation can be applied at different scales, in particular, to software projects and software ecosystems. In this thesis, we define a *software project* as a collection of source or binary code together with additional documentation and metadata that addresses a specific problem. The functionality offered by a software project can be split into modules, each one addressing a different functional aspect. Moreover, software projects seldom live in isolation. Instead, they coexist with other projects in software ecosystems. We define a *software ecosystem* as a collection of software projects that evolve together [102]. Modularisation can also be applied to software ecosystems where software projects act as modules. In a software ecosystem, a software project has a

dual role: the *library* role when it offers a set of services to other projects via a well-defined interface, and; the *client* role when it depends on other projects to leverage their functionality.

The environment where software dwells and its underlying requirements can change with time. This change results in a chain of software modifications that Lehman coined *software evolution* [96]. Software evolution is a phenomenon that materializes after executing a set of design and development processes that lead to a modified piece of software. Introduced changes can impact not only the concrete implementation of a module but also its interface.

Regardless of how well-maintained dependencies are, modular systems face the risk of propelling a ripple effect where changes to a module's interface propagate to client code in a backwards-incompatible fashion. Such changes are known as *breaking changes*. Changes are said to be backwards-incompatible if they raise syntactic or semantic errors in client code. The adverse effects of breaking changes can even be amplified by poorly managed dependencies (*i.e.*, dependencies that do not follow best practices) among modules [105].

For decades, some software communities and scientific literature have assumed that breaking changes are inherently harmful [18, 68]. In fact, several software communities and projects such as Eclipse and the Java programming language itself [18], have gone as far as to forbid the introduction of any breaking change to guarantee backwards compatibility. However, contrary to aiming at damaging client code, software evolves to increase its offered value; the introduction of new features, bug fixes, security patches, refactorings, and other extra-functional enhancements are common triggers of software evolution [12, 20]. Nevertheless, these improvements might come at the cost of increasing the technical lag [55, 150] on their client projects or even losing them. *Technical lag* refers to the lag between the deployment of a new library release and a client project that takes no action to upgrade [55]. That is, the release of the library used by the client gets outdated [150]. If the client keeps stagnant avoiding the upgrade of its dependencies and code, technical debt builds up.

Technical debt represents the costs a software project incurs when a good enough solution is shipped instead of providing the full set of expected features [29]. For instance, in the case of software evolution, a maintainer can refrain from refactoring its Application Programming Interface (API) for the sake of compatibility, which, in turn, increases maintenance costs in the future [18].

On that account, we claim that breaking changes are not harmful *per se*. Some of them won't ever impact client code. Even if they do, they might bring benefits that can be leveraged by client projects. Nevertheless, they must not come unannounced, otherwise, the trust client projects have put in the evolving module might be threatened. To increase trust, transparency and communication must be reinforced. Concretely, knowing when and where breaking changes are to be expected is needed to inform about the stability of the interface and the effort required to update to a newer release [23]. Different mechanisms such as versioning schemes, as well as code mechanisms such as annotations and naming conventions are used to share this information. However, developers need first to be aware of all sorts of introduced breaking changes—which is an undecidable problem—and be disciplined enough to label them accordingly. To help with the former, there are some static tools that automatically detect the introduction of breaking changes in the code. Unfortunately, they are not able to detect their impact on client projects. A more accurate method to automatically detect breaking changes introduction and their impact on client code is, thus, needed.

In this thesis, we address the so-called library-client co-evolution problem where breaking changes introduced in a library propagate to client code. We study the problem from the *nounal view* that aims at understanding the nature of the library-client co-evolution phenomenon, and; from the *verbal view* that aims at providing new processes, methods, and tools that can better support the software maintenance process [98]. In the nounal view, we study (i) best practices to define dependencies as a way of preventing the propagation of breaking changes, and; (ii) syntactic breaking changes and their impact on client projects in relation with semantic versioning. In the verbal view, primarily

motivated with the statistical-backed findings provided in the nounal view, we introduce the static impact analysis approach and its implementation in Maracas, and the static reverse dependency compatibility testing and its implementation in BreakBot. Maracas is a static analysis tool that detects syntactic breaking changes between two versions of a Java library and their impact on client code. This library is the main engine providing information to BreakBot. The latter is a GitHub bot that assists library evolution by reporting insights into syntactic breaking changes introduction and their impact on client code.

The remainder of the chapter is structured as follows. Section 1.1 introduces the core concepts needed to understand the contributions of the thesis. In Section 1.2, we introduce the thesis problem statement including the underlying motivation and the used methodology. Section 1.3 dives into the research questions and methods to address the thesis problem. Section 1.4 gives general information about the context in which this thesis was developed. In Section 1.5, we present the thesis contributions in terms of produced software and datasets. Lastly, Section 1.6 describes the origin of the chapters included in this thesis.

1.1 Background

In this section, we introduce the main concepts needed to understand the contributions of this thesis. In particular, we present an overview of *software modularisation* and how it manifests at different scales. We then explore how modular systems undergo *software maintenance* processes and how the execution of such processes leads to *software evolution*. We also discuss how changes introduced in these processes can yield *breaking changes*, which are *backwards-incompatible* by nature. We wrap up the section by describing some mechanisms (*e.g.*, versioning and naming conventions) used to signal code prone to be impacted by breaking changes.

Software Modularisation

The time-honoured principle of *information hiding* [126] states that implementation details must be hidden from other parts of the system [19]. Consequently, information hiding propels the principle of software *modularisation*, which states that software should be split into modules that expose functionality via a well-defined interface and keep information details hidden [19]. Modules can reuse and leverage the functionality provided by others by depending on them.

These design principles can be applied at different scales, *e.g.*, at the software project and software ecosystem scales. On the one hand, a *software project* is a set of source code files and additional metadata and documentation that can be built, run, and debugged. A software project can be modularised by splitting it into modules (*e.g.*, classes, components). On the other hand, software projects can act themselves as modules within a software ecosystem. There are diverse definitions for software ecosystem, many of them addressing social and business perspectives [104]. However, we stick to the one provided by Lungu et al., which considers software projects—our subjects of study—as first-class citizens: "A *software ecosystem* is a collection of software projects which are developed and evolve together in the same environment." [102] A software ecosystem is usually built around a shared project, programming language, package manager, and/or community [18]. To act as modules, software projects define an interface known as an API, which exposes a set of services that can later be used by other projects within the ecosystem. Dependent projects are known as *client* projects (refer to as *clients* henceforth), whilst dependee projects are known as *library* projects (refer to as *libraries* henceforth).

Software Evolution & Maintenance

In a software ecosystem, the evolution of a software project triggers changes in others due to existing dependencies. Figure 1.1 shows an overview of how library-client co-evolution manifests.

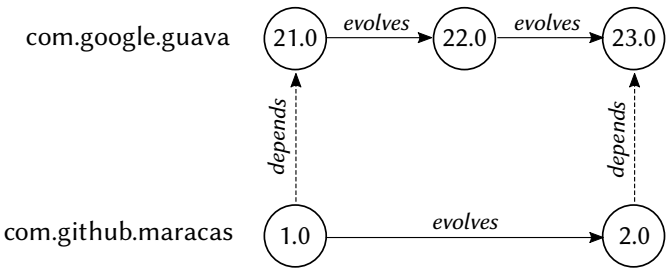


Figure 1.1: Library-client co-evolution

A client project release depends on a specific release of a library. A *software release* is a versioned distribution of a software project. Each release is packed not only with code but also with metadata that is later used to resolve dependencies towards other project releases. The metadata varies depending on the ecosystem where the project inhabits, but, in general, it contains the *name* that identifies the project; the *specific version* of the release, and; the list of *dependencies* that point to other required project releases [2]. At some point, library developers adapt their libraries. The new library distributions containing all the introduced changes result in new releases. If clients aim at leveraging these changes, they need to bear some of the costs of upgrading their own code and potentially the ones of their own clients—if any.

To provide a definition of software evolution we need to start by referring to software maintenance. Software maintenance is defined in *The IEEE Standard for Software Maintenance* as the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment." [103] The term *maintenance* on its own refers to the process of preserving something in appropriate condition or state, avoiding its deterioration. However, as pointed out by Lehman, software cannot deteriorate spontaneously [95]. Changes introduced to maintain software are directed towards fixing existing faults (*i.e.*, corrective maintenance), adapting to a changing environment (*i.e.*, adaptive maintenance), or improving existing code (*i.e.*, perfective maintenance) [103].

The difference between software evolution and software maintenance is then fuzzy [24]. Actually, some researchers and practitioners use the terms interchangeably [25, 152, 154], even though there is indeed a semantic difference [7]. To avoid confusion and overlapping definitions, we define *software maintenance* as the *process* of modifying a software project to preserve or increase its value, and; *software evolution* as the *phenomenon* that results from executing a collection of evolutionary processes resulting in some sort of software modification. Software maintenance is, in this regard, an evolutionary process that impacts how software evolution manifests [80].

Backwards Compatibility

Library evolution can introduce two different types of changes: breaking and non-breaking changes. *Breaking Changes (BCs)* are said to be backwards incompatible—that is, when introduced, they can potentially break client code. On the contrary, *non-breaking changes* are considered backwards compatible, meaning that client projects upgrading to newer releases of the library cannot be negatively impacted by the changes [43]. BCs are language-specific, meaning that the kind of BCs present in one programming language might differ from other. To generalize the types of BCs that we can get regardless of the programming language, we introduce the following simile: the API of a library is like a language with its own *syntax* and *semantics*. BCs can either be *syntactic* if they impact the form of the API (*e.g.*, removing a method, changing the parameter list of a function), or; *semantic* if they change the behaviour of certain API members (e.g. changing the returned value of a method, adding or removing side effects from a function).

In Java—the language subject of study of this thesis—*syntactic* BCs, can happen at different levels [40], namely: (i) at the source level—BCs are detected at compile time in the form of compilation errors, and; (ii) at the binary level—BCs are detected at linking time in the form of linkage errors. Different BC types can appear at both levels, for instance, removing a public method in Java

is considered both a binary and source incompatible change. However, in Java, each set of BCs—as classified by levels [40]—is independent of the other set, and even though they intersect, none is a superset of the other [70]. Additionally, *semantic* BCs are detected at run time in the form of unexpected behaviour—for instance, after executing unit tests.

Various mechanisms, such as code annotations and versioning and naming conventions, have been created to help developers announce the introduction of different types of changes. For instance, unstable interfaces or implementation code that should not be used by clients can be signaled with annotations such as `@Deprecated`, or other annotations like Google's `@Beta` and Apache's `@Internal`; or by means of using naming conventions on packages, classes, and other members of a software project (*e.g.*, *internal* or *experimental*). Moreover, developers rely on versioning conventions—such as semantic versioning—to communicate the change introduced between releases of the same library.

In particular, *semantic versioning* (`semver` for short) is a versioning convention used to inform about the nature of the changes introduced in a library release [130]. A version in `semver` is defined as `major.minor.patch-qualifier`, where `major`, `minor`, and `patch` are version numbers, and `qualifier` is an alphanumeric identifier. According to `semver`, a change in the `major` number announces the introduction of BCs; changes in the `minor` number announce the introduction of a new feature in a backwards-compatible manner; changes in the `patch` number signal the introduction of backwards-compatible bug fixes, and; the `qualifier` labels the project with information about the pre-release or build metadata. In addition, there are *initial development* releases identified by the use of a zero in the major number (*i.e.*, `0.minor.patch-qualifier`). These releases must be considered unstable, meaning that BCs might appear at any time during the development process [130]. Nevertheless, developers do not always stick to this scheme and its semantics, making the interpretation of changes in version numbers imprecise. For instance, developers might introduce BCs in minor or patch releases [135], or treat initial development versions as stable and mature [33].

1.2 PROBLEM STATEMENT

After introducing the main concepts and background needed to understand the contributions of this thesis, we now present the thesis problem statement. We introduce first the motivation and the problem addressed in this manuscript. We then describe the methodology we followed to address the given problem.

MOTIVATION BCs introduced in library interfaces potentially impact client code. The effects of such changes are amplified by poorly managed dependencies among software projects. These circumstances result in the so-called *library-client co-evolution problem*.

Library-client co-evolution problem. Lack of understanding of co-evolution between library and client projects in a given software ecosystem and support for their evolutionary processes: the evolution of a library might negatively impact client code. Evidence about this impact is required, so developers can decide when, where, and how changes should be introduced.

On the one hand, poorly managed dependencies might result in clients depending on an unstable API. For instance, (i) depending on concrete implementations rather than interfaces; (ii) not providing a specific tested version or version range when defining a dependency, and; (iii) keeping unneeded dependencies as part of the project are some known behaviours of unhealthy dependency management (*cf.* Chapter 2). Knowing which concrete best practices are advised to manage dependencies might significantly decrease the impact of library evolution on client projects in diverse ecosystems [16].

On the other hand, library developers need to evolve their libraries to preserve or increase the libraries' value. However, introduced changes might threaten clients even in scenarios where dependencies are well-maintained. These threats are perceived as an additional cost that hampers the development process of the client project itself. The latter might opt for incurring the

costs of upgrading to the new release of the library or dropping its use if they consider it too costly. That is why some software communities have deemed BCs harmful [68]. This line of thought has resulted in the misconception that stability is equivalent to stagnation [18, 21, 109]. We claim instead that stability is a state in which software does not introduce BCs *that break client code* (*cf.* Chapter 3). However, if a BC does impact a client, the change must be identified and communicated to avoid damage and mistrust in the project [16]. Knowing *when* and *where* BCs are introduced is thus essential to improve the communication among libraries and clients. How many known clients will a BC impact? How significant is the impact, if any, in terms of affected client code? In particular, the answers to these questions support library maintainers when deciding whether and how to introduce a BC (*cf.* Chapter 5), and; clients in deciding whether to upgrade to a new release of a library or which library to use.

In this thesis, we address the library-client co-evolution problem. In particular, we dive into the introduction and impact of BCs, and the way dependencies should be defined to reduce their propagation.

METHODOLOGY Lehman suggested two different views to study the software evolution phenomenon, namely the *nounal* and the *verbal views*. The *nounal view* (evolution as a noun) also known as the *knowledge-seeking* view [148, 149], studies the *nature* of the phenomenon, contributing to the understanding of software evolution by studying its causes, properties, and impact, among others. In contrast, the *verbal view* (evolution as a verb) otherwise called the *solution-seeking* view [148, 149], focuses on the *means* of the phenomenon—that is, on developing and improving the processes, methods, and activities used to support software evolution [98]. Findings in the nounal view motivate studies in the verbal view by providing empirical evidence that support the development of new software evolution approaches. These two views are used as the backbone of this thesis to both motivate, plan, and structure all included research contributions.

Regarding the *nounal view*, we aim first at gaining knowledge on how dependencies are established in software ecosystems, and which practices can hinder software maintenance when defining such dependencies in a software project. Once dependency handling is better understood, it is important to dig into the code to describe how software evolution happens in the wild. In particular, we aim at identifying what is the type and frequency of BCs and their actual impact on projects within a software ecosystem. Methods and techniques coming from the empirical software engineering field are used to provide answers to our inquiries.

Knowledge discovered in the nounal view of the research provides statistical evidence to motivate the development of new approaches in the *verbal view*, which can better support the library-client co-evolution processes. In particular, with the creation of adequate methods and tools, we can help library developers in understanding and foreseeing the impact that introduced BCs have on clients before releasing them.

1.3 RESEARCH QUESTIONS

Given the problem statement and methodology introduced in the previous section, we now discuss the research questions that help us address the *library-client co-evolution problem*. Concretely, we present three research questions that target (i) dependency management best practices (**Q1**); (ii) semantic versioning, breaking changes, and their impact on client code (**Q2**), and; (iii) a new method and tool to offer library evolution assistance (**Q3**). We introduce each question with its corresponding *motivation* and employed *method* to address it.

Research Question 1: Dependency Management Best Practices

MOTIVATION Starting with the nounal view of the study, we first analyse how dependencies are defined between the library and client projects. Dependencies can usually be declared in various manners, for instance, they can be declared at different

levels of granularity (*e.g.*, package or component level), scopes (*e.g.*, compile, runtime, test), *etc*. However, this freedom creates confusion among developers; it is not clear which is the best alternative to follow. We aim, therefore, at identifying best practices when defining such relationships, measuring the impact of these best practices in the project, and evaluating to what extent these practices are being followed. This brings us to the definition of the first research question of the thesis.

Q1: What dependency management best practices are advised and followed and what observable effect do they have on software projects?

METHOD To answer **Q1**, we use the Open Service Gateway Initiative (OSGi) framework as the subject of study [6]. OSGi is a module system and service framework, designed to support modular development in Java. First, to identify dependency management best practices suggested by experts and practitioners, we conduct a systematic review of gray literature. We then perform an empirical study to analyse the use of such best practices in a corpus of OSGi bundles. Finally, we modify the studied bundles, so they follow a subset of best practices, and we register their observable effects in terms of size and performance.

Research Question 2: Semantic Versioning, Breaking Changes & Impact Analysis

MOTIVATION Once best practices of dependency management and their implications are identified, it is time to zoom into the library-client co-evolution problem itself. How does this co-evolution manifest in terms of BCs? Are BCs communicated by libraries? What is the real impact of these changes on clients? These and other related questions are of foremost importance when unveiling the magnitude of the library-client co-evolution problem. Moreover, the means used by library developers to communicate change or instability must be considered. Changes

are not pernicious by themselves but they should not come unannounced. The use of versioning schemes (*e.g.*, `semver`)—which also happens to be a dependency management best practice (*cf.* Chapter 2)—and code-level mechanisms (*e.g.*, annotations, naming conventions) must be considered in the analysis to correctly diagnose the severity of BCs in software ecosystems.

Q2: What is the real impact of BCs on clients?

METHOD Our starting point to explore **Q2** is the study by Raemaekers, Deursen, and Visser, entitled *Semantic Versioning and Impact of Breaking Changes in the Maven Repository* and published in The Journal of Systems and Software in 2017 [135]. This study investigates whether library developers use `semver` to signal BCs introduction, and how these BCs impact client projects in terms of compilation errors. Given the relevance of this research and the closeness to our own inquiries, we conduct an external and differentiated replication study of this empirical work [101]. We also develop a new tool, MARACAS, to address some of the limitations of the original protocol. Descriptive and inferential statistics are later used to analyse the data and provide a robust answer to our research question.

Research Question 3: Library Evolution Assistance

MOTIVATION Once the previous two research questions have been answered, we get enough information to feed back into the software evolution process and focus on the verbal view. Library developers are constantly facing the following dilemma: whether to introduce changes to preserve or increase the library's value at the cost of breaking client code; or avoid the risk and confront the consequences of immobility and technical debt [27, 55]. In this regard, we aim at helping library developers to understand and anticipate BCs that might be harmful to clients [65]. Tools that support the identification of BCs before a new release is

distributed in an ecosystem are required to help stakeholders make evidence-backed decisions.

Q3: How to assist library evolution?

METHOD To answer **Q3**, we develop a new approach that statically analyses the introduction of BCs in a set of commits, and their impact on relevant clients. The result of the analysis is fed back to developers via an enriched code review report. The approach is implemented in our prototype BreakBot. We evaluate the accuracy of the tool by reporting the precision and recall obtained for two synthetic projects. Then, to validate the usefulness of the approach, we gather a corpus of relevant library pull requests and use BreakBot to generate impact analysis reports. A survey is then distributed to gather the opinion and feedback of library maintainers. Results are then qualitatively analysed.

1.4 THESIS CONTEXT

The research enclosed in this thesis was partially executed under the umbrella of the Crossminer project,[1] funded by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement no. 732223. The main goal behind this project was to support developers on the adoption of Open-source Software (OSS) given a set of required standards in terms of quality, maturity, activity of development, and user support. Both risks and benefits should be exposed to help developers and other stakeholders make informed decisions. The contribution of the project was twofold: (i) deliver an open-source platform that supports the analysis of OSS, and; (ii) facilitate the knowledge extraction from OSS repositories. To do so, diverse information sources were considered *e.g.*, source code, source code repositories, communication channels (*e.g.*, forums, mail-

1 https://www.crossminer.org/

ing lists), bug tracking systems, and other relevant metadata (*e.g.*, licenses, dependency definitions). As part of the Centrum Wiskunde & Informatica (CWI) team, our work package was directed towards analysing source code and dependencies, and extracting actionable knowledge from them.

The outcome of this initial work set the foundations of the ALIEN (Usage-Driven Software Library Evolution) project, proposed by Thomas Degueule and funded by the French National Research Agency through the grant ANR-21-CE25-0007. The main goal of the project is "to investigate how library maintainers can better understand and anticipate the impact of their changes on their clients and ecosystems to help them make informed and responsible decisions about the development and evolution of their library." As main contributions, the project aims at: (i) understanding software libraries usage; (ii) performing impact analysis of software evolution, and; (iii) augmenting development tools with usage-driven feedback. I am one of the main collaborators of the project as a Ph.D. student affiliated to the Software Engineering and Technology (SET) group at TU/e.

1.5 ARTEFACTS

In the context of this thesis, a set of *software* artefacts and *datasets* were produced. Such assets are listed in this section and pointers to the locations where they were originally published are provided to guarantee the repeatability, reproducibility, and replicability of our results. To ease the searching process and avoid having all assets spread around, we gather such assets as part of a Zenodo repository that can be found at https://zenodo.org/record/7466409/.

Software

OSGI ANALYSIS PROJECT The *OSGi Analysis Project*[2] was developed to help answer **Q1**. It was built in the context of the

2 https://github.com/crossminer/osgi-analysis-rascal/

CROSSMINER project, using the Rascal meta-programming language [84]. The goal of the project is to extract and model relevant information from bundles metadata (*aka.*, Manifest files) and Java bytecode to, then, compute a set of metrics that help us perform a dependency analysis. In particular, it helps verify if dependency management best practices are being followed by bundles.

MARACAS IN RASCAL Initially, the MARACAS[3] project was developed in the Rascal meta-programming language under the umbrella of the CROSSMINER project. It was used to address **Q2**. The project aims at (i) modeling Java bytecode facts by using Rascal M3 models [10]; (ii) computing the list of BCs between two versions of a Java ARchives (JAR) file by relying on `japicmp`[4], and; (iii) identifying client locations that are impacted by the aforementioned BCs.

MARACAS IN JAVA To better integrate MARACAS with underlying Java libraries and improve its performance, the MARACAS project was migrated to Java.[5] Its development, this time, was done in the context of the ALIEN project. The new MARACAS project was defined as a source code and bytecode analysis framework used to analyse the library-client co-evolution in Java ecosystems. At the time of writing this thesis, MARACAS relied both on `japicmp` and the Spoon framework [128] to perform the required analyses. Additionally, the project was extended with (i) a REST API that exposes MARACAS capabilities to analyse libraries and clients, and; (ii) source code forge connectors that enable the analysis of remote OSS projects. This extension was included to address **Q3**.

BREAKBOT BREAKBOT[6] was developed to address **Q3**. It was built at Centre National de la Recherche Scientifique (CNRS) in the context of the ALIEN project. Its main contributor is Leonard

3 https://github.com/crossminer/maracas/
4 https://siom79.github.io/japicmp/
5 https://github.com/alien-tools/maracas/
6 https://github.com/alien-tools/breakbot/

Rizzo. BreakBot's main goal is to identify the introduction of BCs in Java libraries hosted on GitHub and their impact on client projects. A report is generated with insightful information to help library maintainers evolve their projects.

Datasets

OSGi sources and corpus A set of OSGi authorized resources and Eclipse bundles were included to answer **Q1**[7]. The Zenodo repository contains: (i) the OSGi resources and the result of the systematic review, and; (ii) seven OSGi corpora used during the empirical evaluation to address **Q1**. It includes a *control* corpus with 372 original Eclipse bundles and six *transformed* corpora, which are the output of transforming the original corpus to address OSGi best practices.

Breaking bad? dataset & analysis A dedicated pipeline[8] was created to generate required datasets and analyses to answer **Q2**. The Zenodo repository contains: (i) the Java code, R scripts, and SQL queries to get relevant information from two different corpora, namely the Maven Dependency Dataset (MDD) and the Maven Dependency Graph (MDG); (ii) the resulting datasets, and; (iii) a set of Jupyter notebooks written in R, which analyse the extracted data and help us draw conclusions on our research inquiries.

BreakBot dataset This dataset[9] contains information of breaking and impactful Pull Requests (PRs) of popular Java libraries hosted on GitHub. This dataset is used to address **Q3**. It was also later used to simulate interesting PRs in the BreakBot playground. BreakBot reports were generated for these subjects.

7 https://github.com/msr18-osgi-artifacts/msr18-osgi-artifacts/
8 https://zenodo.org/record/5221840/
9 https://zenodo.org/record/7475823

This thesis is a compilation of multiple research publications that have been organized into chapters following a common research line and narrative. Each publication has been slightly adapted with respect to its original version based on the thesis format and additional comments performed by the Ph.D. committee members. Beware that each chapter was built as an independent article that by nature needs to be self-contained, leading to some redundancy across the manuscript.

The research done for this thesis led to five peer-reviewed publications. I am the main author of four of them and they are the result of a collaboration with Thomas Degueule (co-promotor), Jurgen J. Vinju (promotor), and Jean-Rémy Falleri. These four publications are the ones that give origin to the chapters contained in this document. The fifth publication was a collaboration with Phuong Nguyen, Juri Di Rocco, Davide Ruscio, Thomas Degueule, and Massimiliano Di Penta. Hereafter, I give details on how the chapters relate to the research questions (*cf.* Section 1.3) and the aforementioned publications.

Chapter 2 - OSGi Dependency Management Best Practices

Chapter 2 addresses **Q1** and the main results are consolidated in the following publication.

> Lina Ochoa, Thomas Degueule and Jurgen J. Vinju. "An Empirical Evaluation of OSGi Dependencies Best Practices in the Eclipse IDE". In: *15th International Conference on Mining Software Repositories* (2018). IEEE/ ACM, pp. 170-180, DOI: 10.1145/3196398.3196416

As a result of this first step towards understanding the complexity of project relations, researchers at the University of L'Aquila strove to create a collaborative-filtering recommender system that suggests usage patterns when depending on and using an OSS project. The result of this research was the development of FOCUS, a tool that mines OSS repositories to recommend li-

brary method invocations and usage patterns. These suggestions are taken from projects that resemble the one that is requesting the advice. The Centrum Wiskunde & Informatica (CWI) team contributed with static analysis tools that helped in the identification of the library method invocations and usage patterns, as well as on the final report of the study. This collaboration resulted in the following publication—which has not been included in the current thesis.

> Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule and Massimiliano Di Penta. "FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns". In: *41st International Conference on Software Engineering* (2019). IEEE/ACM, pp. 1050-1060, DOI: 10.1109/ICSE.2019.00109

Chapter 3 - Breaking Bad? Semantic Versioning and Breaking Changes

Q2 is answered in Chapter 3. The results of this study were accepted as a presentation abstract in the 19th Belgium-Netherlands Software Evolution Workshop (BENEVOL) in 2020 and further developed in the following article.

> Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen J. Vinju. "Breaking Bad? Semantic Versioning and Impact of Breaking Changes in Maven Central". In: *Empirical Software Engineering* 27-3 (2022), pp. 1573–7616 DOI: 10.1007/s10664-021-10052-y

Afterwards, the same article was accepted at the Journal-First Papers track at the International Conference on Software Engineering (ICSE) in 2023.

Chapter 4 - MARACAS: Designing and Implementing the Static Impact Analysis Approach

Chapter 4 introduces MARACAS, the backbone tool of this thesis. It was used to answer **Q2** and **Q3**. It is designed as a technical chapter that presents the functional and extra-functional require-

ments of the static impact analysis approach, the architecture of the approach, its implementation (*i.e.*, MARACAS), and the testing of such an implementation.

Chapter 5 - BREAKBOT: Static Reverse Dependency Compatibility Testing for Java Libraries

Chapter 5 answers **Q3**. The first results of this research were published in the following article selected as best paper of the New Ideas and Emerging Results (NIER) track at ICSE in 2022.

> Lina Ochoa, Thomas Degueule, and Jean-Rémy Falleri. "BREAKBOT: Analyzing the Impact of Breaking Changes to Assist Library Evolution". In: *44th International Conference on Software Engineering: New Ideas and Emerging Results* (2022). IEEE/ACM, DOI: 10.1145/3510455.3512783

The approach and its evaluation were later extended in the following article, which has been submitted to the IEEE Transactions on Software Engineering (TSE) journal at the end of 2022. This article is still subject to modification depending on the comments shared with the journal reviewers.

> Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen J. Vinju. "BREAKBOT: Static Reverse Dependency Compatibility Testing for Java Libraries". In: *IEEE Transactions on Software Engineering* (2023). IEEE, (Submitted)

Part II

THE NOUNAL VIEW

On the *nature* of the library-client co-evolution phenomenon: the role of dependencies, breaking changes, and their impact on client code

2

OSGI DEPENDENCY MANAGEMENT BEST PRACTICES

ABSTRACT *Open Service Gateway Initiative (OSGi) is a module system and service framework that aims to fill Java's lack of support for modular development. Using OSGi, developers divide software into multiple* bundles *that declare constrained dependencies towards other bundles. However, there are various ways of declaring and managing such dependencies, and it can be confusing for developers to choose one over another. Over the course of time, experts and practitioners have defined "best practices" related to dependency management in OSGi. The underlying assumptions are that these best practices (i) are indeed relevant and (ii) help to keep OSGi systems manageable and efficient. In this chapter, we investigate these assumptions by first conducting a systematic review of the best practices related to dependency management issued by the OSGi Alliance and OSGi-endorsed organizations. Using a large corpus of OSGi bundles (1,124 core plug-ins of the Eclipse IDE), we then analyze the use and impact of 6 selected best practices. Our results show that the selected best practices are not widely followed in practice. Besides, we observe that following them strictly reduces classpath size of individual bundles by up to 23% and results in up to ±13% impact on performance at bundle resolution time. In summary, this chapter contributes an initial empirical validation of industry-standard OSGi best practices. Our results should influence*

This chapter is originally published as Lina Ochoa, Thomas Degueule, and Jurgen J. Vinju. "An Empirical Evaluation of OSGi Dependencies Best Practices in the Eclipse IDE". In: *15th International Conference on Mining Software Repositories* (2018). IEEE/ ACM, pp. 170-180, DOI: 10.1145/3196398.3196416.

practitioners especially, by providing evidence of the impact of these best practices in real-world systems.

2.1 INTRODUCTION

The time-honored principle of separation of concerns entails splitting the development of complex systems into multiple components interacting through well-defined interfaces. This way, the development of a system can be broken down into multiple, smaller parts that can be implemented and tested independently. This also fosters reuse by allowing software components to be reused from one system to the other, or even to be substituted by one another provided that they satisfy the appropriate interface expected by a client. Three crucial aspects [126] of successful separation of concerns are module interfaces, module dependencies, and information hiding—a module's interface hides any number of different functionalities, possibly depending on other modules transitively.

Historically, the Java programming language did not offer any built-in support for the definition of versioned modules with explicit dependency management [151]. This led to the emergence of Open Service Gateway Initiative (OSGi), a module system and service framework for Java standardized by the OSGi Alliance organization [5]. Initially, one of the primary goals of OSGi was to fill the lack of proper support for modular development in the Java ecosystem (popularly known as the "JAR hell"). OSGi rapidly gained popularity and, as of today, numerous popular software of the Java ecosystem, including Integrated Development Environments (IDEs) (*e.g.*, Eclipse, IntelliJ), application servers (*e.g.*, JBoss, GlassFish), and application frameworks (*e.g.*, Spring) rely internally on the modularity capabilities provided by OSGi.

Just like any other technology, it may be hard for newcomers to grasp the complexity of OSGi. The OSGi specification describes several distinct mechanisms to declare dependencies, each with different resolution and wiring policies. Should dependencies be declared at the package level or the component level? Can the content of a package be split amongst several components or

should it be localized in a single one? These are questions that naturally arise when attempting to modularize Java applications with OSGi. There is little tool support to help writing the meta-data files that wire the components together, and so modularity design decisions are mostly made by the developers themselves. The quality of this meta-data influences the modularity aspects of OSGi systems. The reason is that OSGi's configurable semantics directly influences all the aforementioned key aspects of modularity: the definition of module interfaces, what a dependency means (wiring), and information hiding (*e.g.*, transitive dependencies). A conventional approach to try and avoid such issues is the application of so-called "best practices" advised by experts in the field. To the best of our knowledge, the assumptions underlying this advice have not been investigated before: are they indeed relevant and do they have a positive effect on OSGi-based systems? Our research questions are:

Q1 What OSGi best practices are advised?
Q2 Are OSGi best practices being followed?
Q3 Does each OSGi best practice have an observable effect on the relevant qualitative properties of an OSGi bundle?

To begin answering these questions, this chapter reports on the following contributions:

- A systematic review of best practices for dependency management in OSGi emerging from either the OSGi Alliance itself or OSGi-endorsed partners; we identify 11 best practices and detail the rationale behind them (**Q1**).
- An analysis of the bytecode and meta-data of a representative corpus of OSGi bundles (1,124 core plug-ins of the Eclipse IDE) to determine whether best practices are being followed (**Q2**), and what is their impact (**Q3**).

Our results show that:

- *Best practices are not widely followed in practice*. For instance, half of the bundles we analyze specify dependencies at the bundle level rather than at the package level—despite the

fact that best practices encourage to declare dependencies at the package level.
- *The lack of consideration for best practices does not significantly impact the performance of OSGi-based systems*. Strictly following the suggested best practices reduces classpath size of individual bundles by up to 23% and results in up to ±13% impact on performance at bundle resolution time.

The remainder of this chapter is structured as follows. In Section 2.2, we introduce background notions on OSGi itself. In Section 2.3, we detail the methodology of the systematic review from which we extract a set of best practices related to dependency management. In Section 2.4, we evaluate whether best practices are being followed on a representative corpus of OSGi bundles extracted from the Eclipse IDE. We discuss related work in Section 2.5 and conclude in Section 2.6.

2.2 BACKGROUND: THE OSGI FRAMEWORK

OSGi is a module system and service framework for the Java programming language standardized by the OSGi Alliance organization [5], which aims at filling the lack of support for modular development with explicit dependencies in the Java ecosystem (*aka.*, the "JAR hell"). Some of the ideas that emerged in OSGi were later incorporated in the Java standard itself, *e.g.*, as part of the module system released with Java 9. In OSGi, the primary unit of modularization is a *bundle*. A bundle is a cohesive set of Java packages and classes (and possibly other arbitrary resources) that together provide some meaningful functionality to other bundles. A bundle is typically deployed in the form of a JAR that embeds a *Manifest file* describing its content, its meta-data (*e.g.*, version, platform requirements, execution environment), and its dependencies towards other bundles. The OSGi framework itself is responsible for managing the life cycle of bundles (*e.g.*, installation, startup, pausing). As of today, several certi-

Listing 2.1: An idiomatic `MANIFEST.MF` file

```
1  Bundle-ManifestVersion: 2
2  Bundle-Name: Dummy
3  Bundle-SymbolicName: a.dummy
4  Bundle-Version: 0.2.1.build-21
5  Bundle-RequiredExecutionEnvironment: JavaSE-1.8
6  Export-Package: a.dummy.p1,
7    a.dummy.p2;version="0.2.0"
8  Import-Package: b.p1;version="[1.11,1.13]",
9    c.p1
10 Require-Bundle: d.bundle;bundle-version:="3.4.1",
11   e.bundle;resolution:=optional
```

fied implementations of the OSGi specification have been defined, including Eclipse Equinox[2] and Apache Felix[3] to name but a few.

OSGi is a mature framework that comprises many aspects ranging from module definition and service discovery to life cycle and security management. In this chapter, we focus specifically on its support for dependency management.

The Manifest File

Every bundle contains a meta-data file located at `META-INF/MANIFEST.MF`. This file contains a list of standardized key-value pairs (known as *headers*) that are interpreted by the framework to ensure all requirements of the bundle are met. Listing 2.1 depicts an idiomatic Manifest file for an imaginary bundle named `Dummy`.

In this simple example, the Manifest file declares the bundle `a.Dummy` in its version `0.2.1.build-21`. It requires the execution environment `JavaSE-1.8`. The main purpose of this header is to announce what should be available to the bundle in the standard `java.*` namespace, as the exact content may vary according to the version and the implementer of the Java virtual machine on which the framework runs. The Manifest file specifies that the

2 https://www.eclipse.org/equinox/
3 https://felix.apache.org/

bundle exports the `a.dummy.p1` package, and the `a.dummy.p2` package in version `0.2.0`. These packages form the public interface of the bundle—its API. Next, the Manifest file specifies that the bundle requires the package `b.p1` in version `1.11` to `1.13` (inclusive) and the package `c.p1`. Finally, the Manifest declares a dependency towards the bundle `d.bundle` in version `3.4.1` and an optional dependency towards the bundle `e.bundle`. We dive into greater details of the semantics of these headers and attributes in the next section.

It is important to note that the Manifest file is typically written by the bundle's developer themself, and has to co-evolve with its implementation. Therefore, discrepancies between what is declared in the Manifest and what is actually required by the bundle at the source or bytecode level may arise. Although some tools provide assistance to the developers (for instance using bytecode analysis techniques on bundles to automatically infer the appropriate dependencies), getting the Manifest right remains a tedious and error-prone task.

OSGi Dependency Management

The OSGi specification declares 28 Manifest headers that relate to versioning, i18n, dependencies, capabilities, *etc.*,. Amongst them, six are of particular interest regarding dependency management: **`Bundle-SymbolicName`** which "*together with a version must identify a unique bundle*", **`Bundle-Version`** which "*specifies the version of this bundle*", **`DynamicImport-Package`** which "*contains a comma-separated list of package names that should be dynamically imported when needed*", **`Export-Package`** which "*contains a declaration of exported packages*", **`Import-Package`** which "*declares the imported packages for this bundle*", and **`Require-Bundle`** which "*specifies that all exported packages from another bundle must be imported, effectively requiring the public interface of another bundle*" [5]. The OSGi specification prescribes two distinct mechanisms for declaring dependencies: at the package level, or at the bundle level. In the former case, it is the responsibility of the framework to figure out which bundle provides the required package—multiple bundles can export the same pack-

age in the same version. Conversely, the latter explicitly creates a strong dependency link between the two bundles.

The `Import-Package` header consists of a list of comma-separated packages the declaring bundle depends on. Each package in the list accepts an optional list of attributes that affects the way packages are resolved. The *resolution* attribute accepts the values `mandatory` (default) and `optional`, which indicate, respectively, that the package must be resolved for the bundle to load, or that the package is optional and will not affect the resolution of the requiring bundle. The *version* attribute restricts the resolution on a given version range, as shown in Listing 2.1.

When it requires another bundle through the `Require-Bundle` header, a bundle imports not only a single package but the whole public interface of another bundle, *i.e.*, the set of its exported packages. As the `Require-Bundle` header requires to declare the symbolic name of another bundle explicitly, this creates a strong dependency link between both. Thus, not only does this header operate on a coarse-grained unit of modularization, but it also tightly couples the components together.

For a bundle to be successfully resolved, all the packages it imports must be exported (`Export-Package`) by some other bundle known to the framework, with their versions matching. Similarly, all the bundles it requires must be known to the framework, with their versions matching. This wiring process is carried out automatically by the framework as the bundles are loaded.

2.3 OSGi Best Practices

The OSGi specification covers numerous topics in depth and it can be hard for developers to infer idiomatic uses and good practices. Should dependencies be declared at the package or the bundle level? Can the content of a package be split amongst several bundles or should it be localized in a single one? These are questions that naturally arise when attempting to modularize Java applications with OSGi. Although all usages are valid according to the specification, OSGi experts tend to recommend or discourage some of them. In this section, we intend to identify

a set of best practices in the use of OSGi. In particular, we look for best practices related to the specification of dependencies between bundles, thus answering our first research question:

Q1 What OSGi best practices are advised?

Systematic Review Methodology

To perform the identification of best practices related to OSGi dependency management, we follow the guidelines specified by [83], which include the definition of the research question, search process, study selection, data extraction, and search results. In this regards, **Q1** is selected as the *research question* of the systematic review.

SEARCH PROCESS Given the absence of peer-reviewed research tackling OSGi best practices (*cf.* Section 2.5), we select as primary data sources web resources of the OSGi Alliance and OSGi-endorsed products. The complete list of certified products[4] corresponds to *Knopflerfish*, *ProSyst Software*, *SuperJ Engine*, *Apache Felix*, *Eclipse Equinox*, *Samsung OSGi*, and *KT OSGi Service Platform (KOSP)*. With the aim to identify best practices, we define a *search string* that targets a set of standard *best practices* synonyms, and their corresponding antonyms:

```
((good OR bad OR best) AND (practices OR design)) OR
                        smell
```

Some of the official web pages of the selected organizations provide their own search functionality. However, we seek to minimize the heterogeneous conditions of the searching environment and only use Google Search to explore the set of web resources. We use JSoup, an HTML parser for Java, to execute the search queries and to scrape the results. We compute all possible keyword combinations from the original search string

4 https://www.osgi.org/osgi-compliance/osgi-certification/osgi-certified-products/

and execute one query per combination and organization domain. For instance, to search for the `best AND practices` keywords in English-written resources on the OSGi Alliance domain, we define the following Google Search query: `http://www.google.com/search?q=best+practices+site:www.osgi.org&domains:www.osgi.org&hl=en`. We retrieved the resources in January 2018.

STUDY SELECTION Figure 2.1 details the resource selection process we follow in this study. First, we only include web resources written in English in the review. As shown before in the Google Search query, this language restriction is included as a filtering option in all searches: `hl=en`. In the end, the search engine returns a total of 268 resources.[5] Second, selected documents should describe best practices related to the management of dependencies in OSGi. To this aim, we conduct a two-task selection where we first consider the occurrences of keywords in the candidate resources, and then we perform a manual selection of relevant documents. On the one hand, we count the occurrences of the searched keywords in each web resource (including HTML, XML, PDF, and PPT files). If one of the keywords is missing in the resource, we automatically discard it. Using this criterion, we reduce the set to 156 resources, and finally 87 after removing duplicates. On the other hand, we manually review the resulting set, looking for documents that address the research question. In particular, if a resource points to another document (through an HTML link) that is not part of the original set of candidates, it is also analyzed and, if it is relevant to the study, it is included as part of our data sources. This task is performed by two reviewers to minimize selection bias. In the end, we select 21 web resources to derive the list of best practices related to OSGi dependencies specification. Some of the OSGi-endorsed organizations do not provide relevant information for the study.

DATA EXTRACTION During the data extraction phase, we consider the organization that owns the resource (*e.g.*, OSGi Alliance), its title, year of publication, authors, and the targeted best prac-

5 https://github.com/msr18-osgi-artifacts/msr18-osgi-artifacts/

Table 2.1: Systematic review of OSGi dependencies specification best practices I

	Resource	Year	Author(s)	Best practices										
				B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11
OSGi Alliance	Automatically managing service dependencies in OSGi [123]	2005	Offermans, M.	○	○	○	○	○	●	○	○	○	○	○
	OSGi best practices! [62]	2007	Hargrave, B.J. et al.	●	●	○	●	●	○	●	○	○	○	○
	Very important bundles [137]	2009	Roelofsen, R.	●	○	○	○	○	●	○	○	○	●	○
	OSGi: the best tool in your embedded systems toolbox [61]	2009	Hackleman, B. et al.	●	○	○	○	○	○	●	○	○	○	○
	bndtools: mostly painless tools for OSGi [9]	2010	Bartlett, N. et al.	○	○	●	○	●	○	○	●	○	○	○
	Developing OSGi enterprise applications [8]	2010	Barci, R. et al.	○	○	○	○	○	○	●	○	○	○	○
	Experiences with OSGi in industrial applications [44]	2010	Dorninger, B.	○	○	○	○	○	○	●	○	○	○	○
	Migration from Java EE application server to server-side OSGi for process management and event handling [76]	2010	Kachel, G. et al.	○	○	●	○	○	○	○	○	○	●	○
	10 Things to know you are doing OSGi in the wrong way [113]	2011	Moliere, J.	●	●	○	○	○	○	●	●	○	○	○
	Structuring software systems with OSGi [48]	2011	Fildebrandt, U.	●	●	○	○	○	○	●	○	○	○	○

Table 2.2: Systematic review of OSGi dependencies specification best practices II

	Resource	Year	Author(s)	Best practices										
				B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11
OSGi Alliance	Best practices for (enterprise) OSGi applications [155]	2012	Ward, T.	●	●	●	●	○	○	○	●	●	○	○
	Building a modular server platform with OSGi [69]	2012	Jayakody, D.	●	●	●	○	○	○	●	●	○	○	○
	OSGi application best practices [74]	2012	Jiang, E.	●	●	○	○	○	○	●	●	●	○	○
	TRESOR: the modular cloud - Building a domain specific cloud platform with OSGi [60]	2013	Grzesik, A.	○	○	○	○	○	○	●	○	○	○	○
	Guidelines [3]	n.d.	OSGi All.	○	○	○	○	○	○	●	○	○	○	○
	OSGi developer certification - Professional [4]	n.d.	OSGi All.	○	○	○	○	○	●	○	○	○	○	○
Felix	Using Apache Felix: OSGi best practices [124]	2006	Offermans, M.	●	○	○	●	○	○	●	○	○	○	○
	OSGi frequently asked questions [46]	2013	Apache Felix	○	○	○	○	○	○	●	○	○	○	●
	Dependency manager - Background [47]	2015	Apache Felix	●	○	○	○	○	○	●	○	○	○	○
Equinox	Best practices for programming Eclipse and OSGi [63]	2006	Hargrave, B.J. et al.	●	○	○	○	○	○	○	○	○	○	○
	OSGi component programming [156]	2006	Watson, T. et al.	●	○	○	○	○	○	●	○	○	○	○

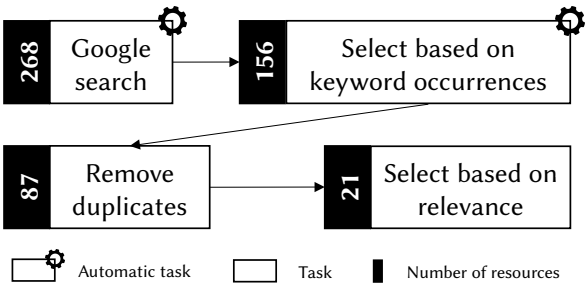


Figure 2.1: Resources selection of the systematic review

tices. To have a common set of best practices, one reviewer reads the selected resources and groups the obtained results in 11 best practices. Afterwards, two reviewers check which best practices are suggested per web resource. Table 2.1 and Table 2.2 present the results of the review. The best practices labels in the table correspond to the best practices presented in Dependencies Specification Best Practices.

Dependencies Specification Best Practices

In this section, we review the best practices identified and summarized in Table 2.1 and Table 2.2. We elaborate on the rationale behind each best practice using peer-reviewed research articles and the *OSGi Core Specification Release 6* [5].

PREFER PACKAGE-LEVEL DEPENDENCIES [B1] Dependencies should be declared using the `Import-Package` header instead of using the `Require-Bundle` header. The latter creates a tight coupling between the requiring bundle and the required bundle, which is an implicit dependency towards an implementation rather than an interface. Thus, it impacts the flexibility of dependency resolution, as the resolver has only one source to provide the dependency (*i.e.*, the required bundle itself). This also naturally complicates refactoring activities: moving a package from one bundle to the other requires to patch all bundles depending on it to point to the new bundle. In contrast, the `Import-Package`

header only relies on an interface and various bundles may offer the corresponding package. Finally, `Require-Bundle` automatically imports all the exported packages of the required bundle, which may introduce unnecessary dependencies. This can get worse in some cases, since package shadowing can be introduced unwittingly [5].

USE VERSIONS WHEN POSSIBLE [B2] Versions should be set when requiring bundles, or when importing or exporting packages. When a bundle requires another bundle or imports a package, a version range or a minimum required version can be defined. Versions must be consciously used to control the dependencies of a bundle, avoiding the acceptance of new versions that might break the component. Version ranges are preferred over minimum versions, because both upper and lower bounds, as well as all in between versions, are supposed to be tested and considered by bundle developers [35]. In addition, with version ranges the dependency resolver has fewer alternatives to resolve the given requirements, allegedly speeding up the process.

EXPORT ONLY NEEDED PACKAGES [B3] Only the packages that may be required by other bundles should be exported. Internal and implementation packages should be kept hidden. Because the set of exported packages forms the public API of a bundle, changes in these packages should be accounted for by the clients [28]. Consequently, the more packages are exported, the more effort is required to maintain and evolve the corresponding API.

MINIMIZE DEPENDENCIES [B4] Unnecessary dependencies should be avoided, given their known impact on failure-proneness [11] and performance of the resolution process. In the case of OSGi framework and the employment of the `Require-Bundle` header, a required bundle might depend on other bundles. If these transitive dependencies are not considered in the OSGi environment, then the requiring bundle may not be resolved [5]. Moreover, dependencies specification in `Require-Bundle`

and `Import-Package` headers may impact performance during the resolution process of the OSGi environment. A bundle is resolved if all its dependencies are available [5]. Presumably, the more dependencies are added to the Manifest file, the longer the framework will take to start and resolve the bundle assuming that all dependencies are included in the environment.

IMPORT ALL NEEDED PACKAGES [B5] All the external packages required by a bundle must be specified in the `Import-Package` header. If this is not the case, a `ClassNotFoundException` may be thrown when there is a reference to a class of an unimported package [5]. This also applies to dynamic dependencies, *e.g.*, classes that are dynamically loaded using the reflective API of Java. The only packages that are automatically available to any bundle are the ones defined in the namespace `java.*`, which are offered by the selected execution environment. However, this environment can offer other packages included in other namespaces. Thus, if these packages are not explicitly imported and the execution environment is modified, they will become unavailable and the bundle will not get resolved.

AVOID DYNAMICIMPORT-PACKAGE [B6] This header lists a set of packages that may be imported at runtime after the bundle has reached a resolved state. In this case, dependency resolution failures may appear in later stages in the life cycle of the system and are harder to diagnose. This effectively hurts the *fail fast* idiom adopted by the OSGi framework [146]. Also, the `DynamicImport-Package` creates an overhead due to the need to dynamically resolve packages every time a dynamic class is used [5].

SEPARATE IMPLEMENTATION, API, AND OSGI-SPECIFIC PACKAGES [B7] It is highly recommended to separate API packages from both implementation and OSGi-specific packages. Therefore, many implementation bundles can be provided for a given API, favoring system modularity. The OSGi service registry is offered to select an implementation once a bundle is requiring and using

the associated API packages. With this approach, API packages can be easily exported in isolation from implementation packages, allowing a change of implementation if needed. Moreover, implementation changes that result in breaking changes for clients bundles are avoided. The abovementioned APIs are known as *clean APIs*, *i.e.*, exported packages that do not use OSGi, internal, or implementation packages in a given bundle [5].

USE SEMANTIC VERSIONING [B8] Semantic versioning[6] is a version naming scheme that aims at reducing risks when upgrading dependencies. This goal is achieved by providing concrete rules and conventions to label breaking and non-breaking software changes [134]. Following these rules, a version number should be defined as `major.minor.micro`. In some cases, the version number is extended with one more alphanumerical slot known as `qualifier`. The `major` number is used when incompatible changes are introduced to the system, while the other three components represent backward-compatible changes related to functionality, bugs fixing, and system identification, respectively. The use of semantic versioning supposedly communicate more information and reduces the chance of potential failures.

AVOID SPLITTING PACKAGES [B9] A split package is a package whose content is spread in two or more required bundles [5]. The main pitfalls related to the use of split packages consist on the mandatory use of the **`Require-Bundle`** header, which is labeled as a bad practice, and the following set of drawbacks mentioned in the *OSGi Core Specification* [5]: (i) *completeness*, which means that there is no guarantee to obtain all classes of a split package; (ii) *ordering*, an issue that arises when a class is included in different bundles; (iii) *performance*, an overhead is introduced given the need to search for a class in all bundle providers; and (iv) *mutable exports*, if a requiring bundle *`visibility`* directive is set to `reexport`, its value may suddenly change depending on the *`visibility`* value of the required bundle.

6 http://semver.org/

DECLARE DEPENDENCIES THAT DO NOT ADD VALUE TO THE FINAL USER IN THE BUNDLE-CLASSPATH HEADER [B10] If a non-OSGi dependency is used to support the internal functionality of a bundle, it should be specified in the `Bundle-ClassPath` header. These dependencies are known as *containers* composed by a set of *entries*, which are then grouped under the *resources* namespace. They are resolved when no package or bundle offers the required functionality [5]. Given that a subset of these resources is meant to support private packages functionality, they should be kept as private packages and defined only in the classpath of the bundle.

IMPORT EXPORTED API PACKAGES [B11] All the packages that are exported and used by a given bundle should also be imported. This may seem counter-intuitive, as exported packages are locally contained in a bundle and can thus be used without being imported explicitly. Nevertheless, it is a best practice to import these packages explicitly, so that the OSGi framework can select an already-active version of the required package. Be aware that this best practice is only applicable to clean API packages [5].

Q1: **What OSGi best practices are advised?** We identify 11 best practices advised by OSGi experts, namely:

B1. Prefer package-level dependencies.

B2. Use version when possible.

B3. Export only needed packages.

B4. Minimize dependencies.

B5. Import all needed packages.

B6. Avoid the use of `DynamicImport-Package`.

B7. Separate implementation, API, and OSGi-specific packages.

B8. Use semantic versioning.

B9. Avoid splitting packages.

B10. Declare dependencies that do not add value to the final user in the `Bundle-ClassPath` header.

B11. Import exported API packages.

2.4 OSGi Corpus Analysis

The best practices we identify in Section 2.3 emerge from experts of the OSGi ecosystem. The goal of the following two research questions is to assess their relevance and impact critically:

Q2 Are OSGi best practices being followed?
Q3 Does each OSGi best practice have an observable effect on the relevant qualitative properties of an OSGi bundle?

Specifically, because beyond their qualitative aspect they are meant to improve performance, we study their impact on the classpath size and resolution time of individual bundles. We first discuss the initial setup and method of our evaluation, and then go through all the selected best practices, aiming at answering our research questions for each of them. After some concluding remarks, we discuss the threats to validity. A complete description of all the artifacts discussed in this section (corpora, transformations, results), along with their source code, is available on the companion webpage.[7]

Studied Corpus

We use an initial corpus consisting of 1,124 OSGi bundles (*cf.* Table 2.3) corresponding to the set of core plug-ins of the Eclipse IDE 4.6 (Neon.1). This corpus emerges from the specific needs of a partner in the collaborative project CROSSMINER in which the authors are involved. The Eclipse IDE consists of a base platform that can be extended and customized through plug-ins

7 https://github.com/msr18-osgi-artifacts/msr18-osgi-artifacts/

Table 2.3: Characteristics of the Eclipse 4.6 OSGi Corpus

Attribute	Value
Initial corpus size	1,124
Number of documentation bundles	17
Number of source bundles	446
Number of test bundles	97
Number of duplicate bundles	192
Studied corpus (C_0)	372
Total size of C_0 (MB)	163.76
Number of dependencies declared in C_0	2,751

that can be remotely installed from so-called *update sites*. Both the base platform and the set of plug-ins are designed around OSGi, which enables this dynamic architecture. The Eclipse IDE relies on its own OSGi-certified implementation of the specification: Eclipse Equinox. Because the Eclipse IDE is a mature and widely-used platform, its bundles are supposedly of high quality. As they all contribute to the same system, they are also highly interconnected: the combination of `Import-Package`, `Require-Bundle`, and `DynamicImport-Package` dependencies results in a total of 2,751 dependency links. As a preliminary step, we clean the corpus to eliminate duplicate bundles and bundles that deviate from the very nature of Eclipse plug-ins. This includes:

- *Bundles with multiple versions*. We only retain the most recent version for each bundle to avoid a statistical bias towards bundles which (accidentally) occur multiple times for different versions.
- *Documentation bundles* that neither contain any code nor any dependency towards other bundles are considered as outliers to be ignored. The best practices are specifically about actual code bundles so these documentation bundles would introduce arbitrary noise.
- *Source bundles* that only contain the source code of another binary bundle are ignored since they are a (technical) accident not pertaining to the best practices either.

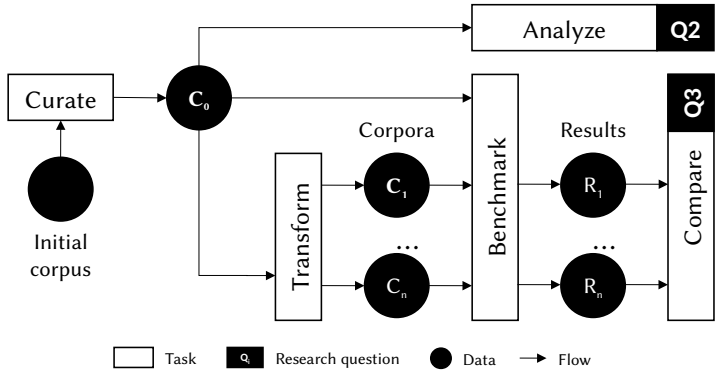


Figure 2.2: Corpus analysis process

- Similarly, *test bundles* which do not provide any functionality to the outside would influence our statistical observations without relating to the studied best practices.

We identify and remove these bundles from the corpus according to their names. The (strong) convention in this Eclipse corpus is that these, respectively, end with a `.doc`, `.source`, or `.tests` suffix. The remaining bundles constitute our control corpus C_0.

Method

The overall analysis process we follow is depicted in Figure 2.2 and detailed below.

SELECTED BEST PRACTICES For the current analysis, we focus on a subset of the best practices ([B1–B6]) elicited in Dependencies Specification Best Practices, which can be studied using a common research method. The other best practices are interesting as future work: [B7, B10, B11] require distinguishing between implementation and API packages, [B8] requires distinguishing between breaking and non-breaking software changes, and [B9] requires refactoring the source code organization of the bundles in addition to their meta-data.

Q2: ARE OSGI BEST PRACTICES BEING FOLLOWED? To answer this research question, we develop an analysis tool, written in Rascal [84], that computes a set of metrics on the control corpus C_0. Specifically, the tool analyses the meta-data (the Manifest files) and bytecode of each bundle to record in which way dependencies and versions are declared, which packages are actually used in the bytecode compared to what is declared in their meta-data, *etc.*,. Based on this information, we then count per best practice how many bundles (or bundle dependencies) satisfy it in the corpus. Using descriptive statistics we then analyze the support for the best practice in the corpus to answer **Q2**. For each best practice, based on the maturity of the Eclipse corpus the hypothesis is that they are being followed (H2.i).

Q3: DOES EACH OSGI BEST PRACTICE HAVE AN OBSERVABLE EFFECT ON THE RELEVANT QUALITATIVE PROPERTIES OF AN OSGI BUNDLE? To answer this research question, we hypothesize that each best practice would indeed have an observable impact on the size of dynamically computed classpaths (H3.1.i) and on the time it takes to resolve and load the bundles (H3.2.i). If either hypothesis is true, then there is indeed evidence of observable impact of the best practice of some kind, if not then deeper analysis based on hypothesizing other forms of impact would be motivated. We are also interested to find out if there exists a correlation between classpath size and related resolution time (H3.3). Since the latter requires an accurate time measurement setup, while the former can be computed from meta-data, it would come in handy for IDE tool builders (recommendations, smell detectors, and quick fix) if classpath size would be an accurate proxy for bundle resolution time.

Figure 2.2 depicts how we compare the original corpus C_0 to alternative corpora C_i in which each best practice B_i has been simulated. For each B_i, a specialized transformation $T(B_i)$ takes as input the control corpus C_0 and turns it into a new corpus $C_0 \xrightarrow{T(B_i)} C_i$ where bundles are automatically transformed to satisfy the best practice B_i. For all transformations $T(B_i)$, we ensure that for all bundles that can be resolved in the original corpus,

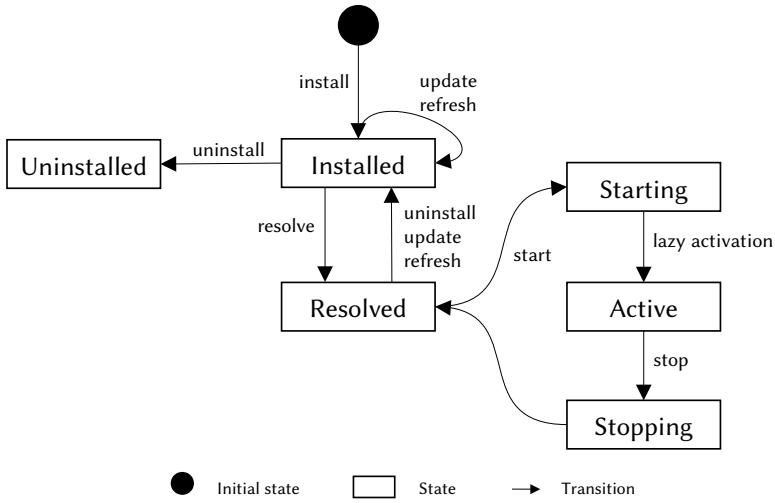


Figure 2.3: Bundle state diagram taken from the OSGi specification [5]

the corresponding bundle in the transformed corpus can also be resolved. For instance, the transformation $T(B_1)$ transforms every `Require-Bundle` header to a set of corresponding `Import-Package` headers, according to what is actually used in the bundle's bytecode. Note that bundles using extension points declared by other bundles must use the `Require-Bundle` header and therefore cannot be replaced with the corresponding `Import-Package` headers. Below, we discuss such detailed considerations with the result of each transformation. Then, we load every corpus C_i in a bare Equinox OSGi console and compute, for every bundle, (i) the size of its classpath, including the classes defined locally and the classes that are accessible through wiring links according to the semantics of OSGi, and (ii) measure the exact time it takes to resolve it. Figure 2.3 shows the bundle state diagram defined in the OSGi specification [5, p. 107]. Resolution time of a bundle is measured as the delta between the time it enters the `INSTALLED` state ("*The bundle has been successfully installed*") and the time it enters the `RESOLVED` state ("*All Java classes that the bundle needs are available*"). To report a change in terms of classpath size or performance, we also compute the relative change between observations in C_i and observations in C_0, where by an observation (v_{ij}), we

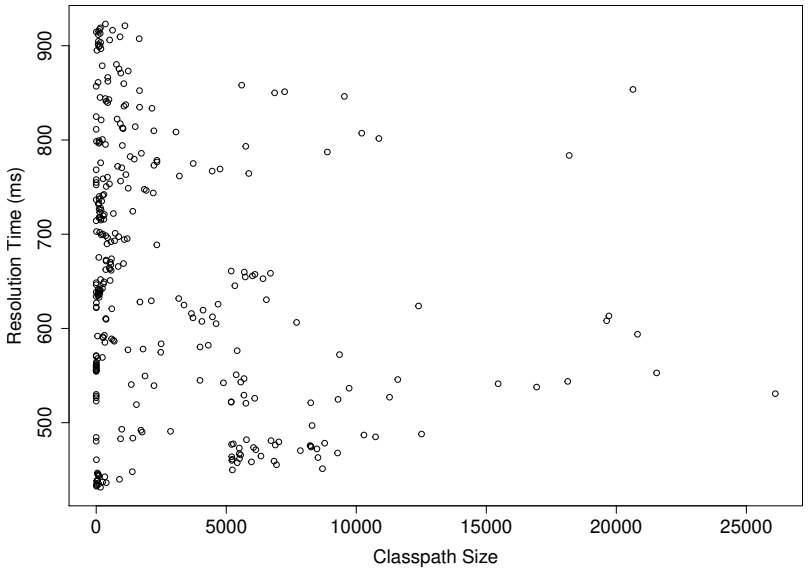


Figure 2.4: Classpath size is a poor indicator for resolution time (ms) in C_0 ($\rho_0 = -0.17$)

refer to the average measure of classpath size or performance for the j^{th} bundle in the i^{th} corpus. The relative change is computed as $d_{ij} = \frac{v_{0j} - v_{ij}}{v_{0j}} \times 100\%$, where d_{ij} is the relative change between the observation of the j^{th} bundle in C_0 (*i.e.*, v_{0j}) and the corresponding observation of the same bundle in C_i (*i.e.*, v_{ij}). The median ($\widetilde{x}$) value of the set of relative changes is used as a comparison measure.[8] All performance measurements are conducted on a MacOS Sierra version 10.12.6 with an Intel Core i5 processor 2GHz, and 16GB of memory running OSGi version 3.11.3 and Java Virtual Machine (JVM) version 1.8. Measurements are executed 10 times each after discarding the 2 initial warm-up observations [15]. We then compute the average of the 10 measurements and their standard deviation.

8 We use $\widetilde{x_c}$ and $\widetilde{x_p}$, respectively, for classpath size and performance comparisons.

Results

To evaluate H3.3 we use both scatter plots and correlations (per corpus) that show the relation between our two studied variables, classpath size and resolution time. Figure 2.4 shows the graph to identify the hypothesized correlation in C_0. Given that there is no linear relation between the variables, we compute the non-parametric Spearman's rank correlation coefficient $\rho_0 = -0.17$, resulting in a weak negative relation. Similar results are observed on all corpora C_i: $\rho_1 = -0.06$, $\rho_2 = -0.17$, $\rho_3 = -0.11$, $\rho_4 = -0.13$, $\rho_5 = -0.17$, and $\rho_6 = -0.17$. According to both visual and statistical analysis, we can reject hypothesis H3.3. Therefore, it remains interesting to study these variables independently. The benchmark results regarding classpath size and resolution time for every corpus C_i, compared to the control corpus C_0, are given in Figure 2.5 and Figure 2.6.

PREFER PACKAGE-LEVEL DEPENDENCIES [B1]

H2.1 To test this hypothesis, we count the number of bundles using the `Require-Bundle` and `Import-Package` headers. We cross-analyze these results by computing the number of extension plug-ins and the number of bundles declaring split packages, which may impact the use of the `Require-Bundle` header. The bundles declare 1,283 `Require-Bundle` dependencies and 1,459 `Import-Package` dependencies. 57.79% of the bundles use the `Require-Bundle` header, 50.00% use the `Import-Package` header, and 34.95% use both. These results suggest that this best practice tends not to be widely followed by Eclipse plug-ins developers. The declaration of extension points and extension bundles, as well as the use of split packages, contribute to these results. In fact, 38.98% of the bundles in C_0 are extension bundles that require a dependency on the bundle declaring the appropriate extension point to provide the expected functionality. Two of the bundles in C_0 use a `Require-Bundle` dependency to cope with the requirements of split packages.

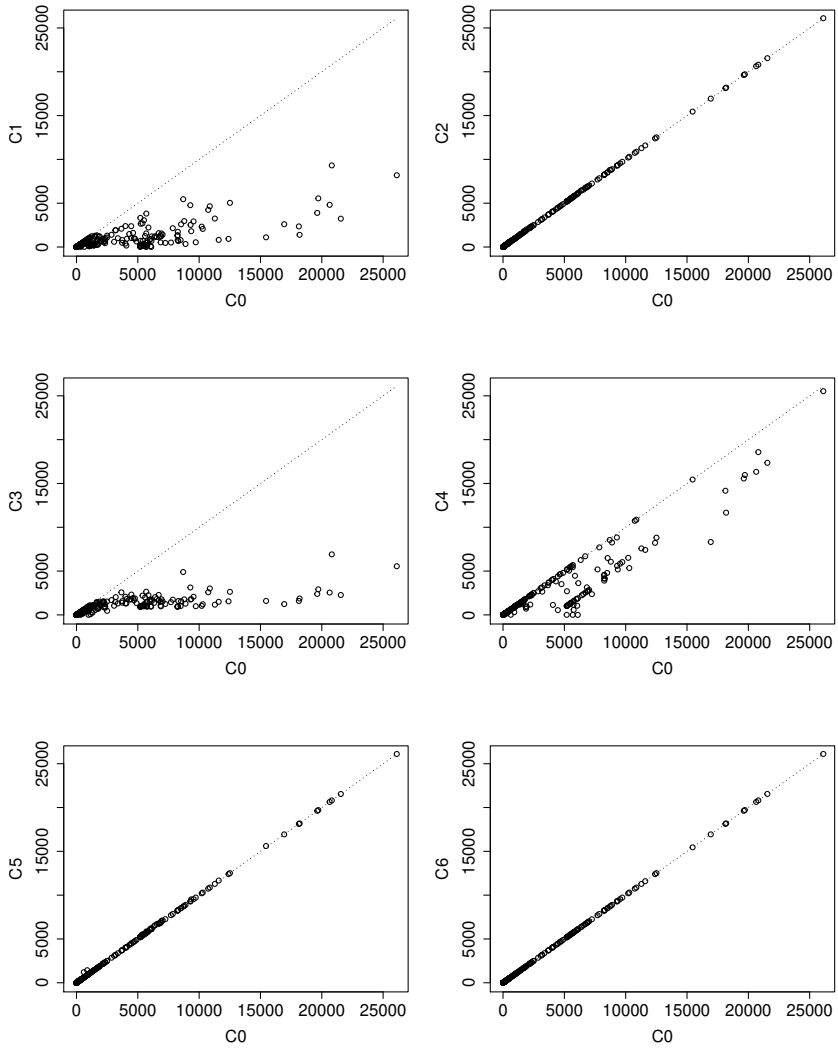


Figure 2.5: Comparing classpath size of corpora C_i (with best practices B_i applied) to the original corpus C_0

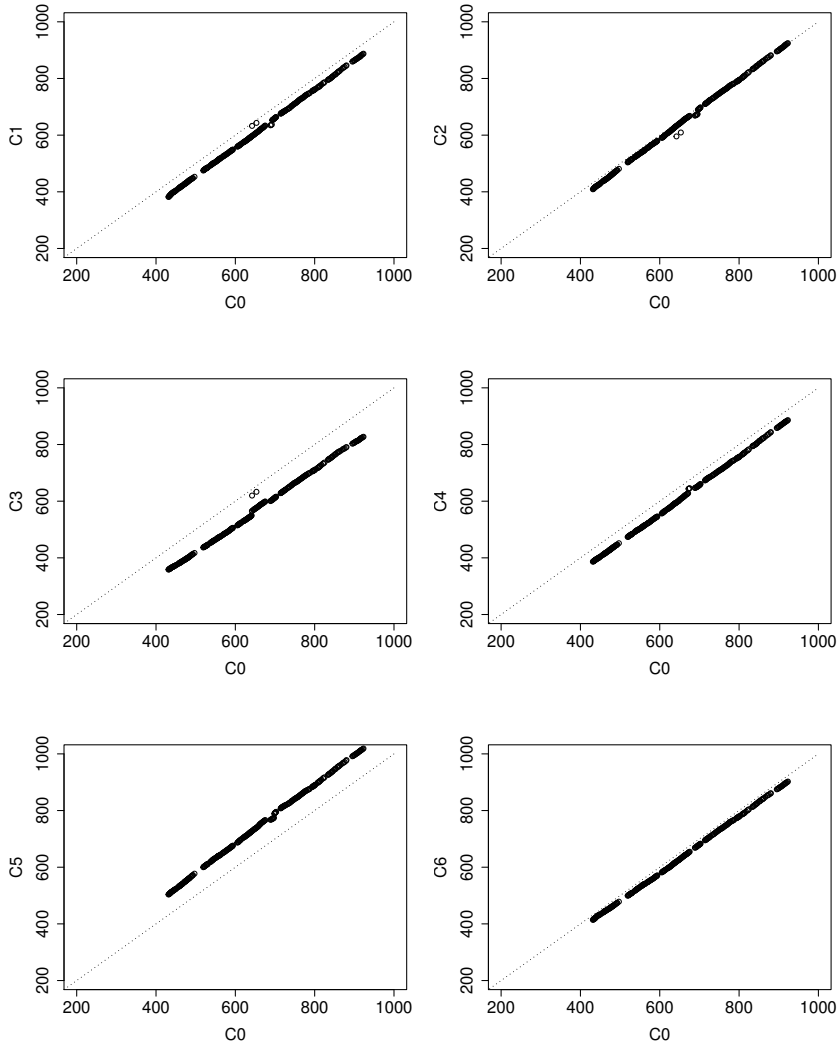


Figure 2.6: Comparing resolution time (ms) of corpora C_i (with best practices B_i applied) to the original corpus C_0

H3.1.1 AND H3.2.1 Transforming bundle-level dependencies to package-level dependencies reduces the classpath size of bundles by $\widetilde{x_c} = 15.40\%$. This is because, in the case of **Require-Bundle**, every exported package in a required bundle is visible to the requiring bundle, whereas the more fine-grained **Import-Package** only imports the packages that are effectively used in the bundle's code. We observe a gain of $\widetilde{x_p} = 7.11\%$ regarding performance (Figure 2.6).

USE VERSIONS WHEN POSSIBLE [B2]

H2.2 To tackle this question, we compute the number of versioned **Require-Bundle**, **Import-Package**, and **Export-Package** relations and the proportion of those that specify a version range. 84.80% of the **Require-Bundle** dependencies are versioned, of which 71.97% (*i.e.*, 783) use a version range. In the case of **Import-Package**, 59.22% of the dependencies are versioned, of which 45.14% use a version range. Finally, 24.88% of the 2,620 exported packages tuples are explicitly versioned. The remaining tuples get a value of `0.0.0` according to the OSGi specification. We observe a tendency to use versions when defining **Require-Bundle** relations, which is highly advised given the need to maintain a tight coupling with a specific bundle. Nonetheless, the frequency of version specifications decreases when using **Import-Package** and even more so with **Export-Package**.

H3.1.2 AND H3.2.2 The transformation $T(B_2)$ takes all unversioned **Import-Package** and **Require-Bundle** headers in C_0 and assign a strict version range of the form $[V, V]$ to them, where V is the highest version number of the bundle or package found in the corpus. In the resulting corpus C_2, we observe that this best practice has no impact on classpath size ($\widetilde{x_c} = 0\%$), and close to zero impact on resolution time ($\widetilde{x_p} = 1.56\%$) of individual bundles.

EXPORT ONLY NEEDED PACKAGES [B3]

H2.3 In this case, we investigate how many of the exported packages in the corpus are imported by other bundles, using either `Import-Package` or `Require-Bundle`, taking versions into account. If an exported package is never imported, this may indicate that this package is an internal or implementation package that should not be exposed to the outside. There may, however, be a fair amount of false positives: some of the exported packages may actually be part of a legitimate API but are just not used by other bundles yet. From the whole set of *exported package* tuples, 14.62% are explicitly imported by other bundles. This suggests that a large portion of the packages that are exported are never used by other bundles. Nevertheless, if we also consider packages imported through the `Require-Bundle` header, at least 80.34% of the total tuples are imported by other bundles. The question that arises is: is this situation intended, or is it a collateral effect of the use of `Require-Bundle`?

H3.1.3 AND H3.2.3 [B3] has an impact on classpath size in the transformed corpus C_3: exporting only the needed packages results in a $\widetilde{x_c} = 23.27\%$ gain sizewise. We also observe an improvement of $\widetilde{x_p} = 12.83\%$ in terms of resolution time for individual bundles.

MINIMIZE DEPENDENCIES [B4]

H2.4 To investigate whether bundles declare unnecessary dependencies, we cross-check the meta-data declared in the Manifest files with bytecode analysis. We deem any package that is required but never used in the bytecode as superfluous. In the corpus, 19.25% of the `Require-Bundle` dependencies are never used locally, *i.e.*, none of the packages of the required bundle are used in the requiring bundle's code. Regarding `Import-Package` dependencies, 13.78% of the explicitly-imported packages are not used in the bytecode. Digging deeper into the relations, we find that the `Require-Bundle` declarations are implicitly importing

15,399 packages that have been exported by the corresponding required bundles. From this set, only 16.50% are actually used in the requiring bundle bytecode. These results suggest that developers tend not to use all the dependencies they declare and that these could be minimized. The situation is much worse in the case of implicitly imported packages through the **Require-Bundle** header, which backs the arguments of [B1].

H3.1.4 AND H3.2.4 [B4] has a close to zero impact on classpath size in the transformed corpus C_4 ($\widetilde{x_c} = 0.14\%$). However, the improvement is higher with regards to resolution time ($\widetilde{x_p} = 7.24\%$).

IMPORT ALL NEEDED PACKAGES [B5]

H2.5 We compute the number of packages that are used in the code but are never explicitly imported in the Manifest file by analyzing the bundles meta-data and bytecode. Our analysis identifies 2,194 packages (269 unique) that are never explicitly imported. Overall, 45.70% of the bundles in C_0 use a package that they do not explicitly import (excluding `java.*` packages).

H3.1.5 AND H3.2.5 For every package that can be found somewhere in the corpus but is missing in the **Import-Package** list of a given bundle, the transformation $T(B_5)$ creates a new **Import-Package** statement pointing to it. The resulting corpus C_5 does not differ from C_0 in terms of classpath size, but appears to be slower in terms of resolution time ($\widetilde{x_p} = -13.35\%$). By creating new explicit dependencies to be resolved, this best practice adds to the dependency resolution process, which in turn may explain this difference.

AVOID DYNAMICIMPORT-PACKAGE [B6]

H2.6 In the corpus, only 7 bundles declare **DynamicImport-Package** dependencies, for a total of 9 dynamic relations declared in C_0. 4 of these dynamically imported packages are not exported by any

bundle. This may result in runtime exceptions after the resolution of the involved bundles. While there are some occurrences in the corpus of this not-advisable type of dependency, results suggest that developers tend to avoid using the `DynamicImport-Package` header and thus generally follow this best practice.

H3.1.6 AND H3.2.6 We do not observe any impact in terms of classpath size, and in terms of performance we observe a gain of $\widetilde{x_p} = 3.47\%$. As our benchmark stops at resolution time and [B6] only has an impact after resolution time, this is unsurprising.

Analysis of the Results

Figure 2.7 summarizes the overall results regarding relative change of our analysis for classpath size and resolution time.

Q2 Overall, we observe that most of the best practices we identify are not widely followed in the corpus. This is for instance the case with [B1], despite being the most-widely advocated best practice among the ones we select (*cf.* Table 2.1 and Table 2.2).

Q2: **Are OSGi best practices being followed?** Three ([B1], [B4], and [B5]) out of the six studied OSGi best practices related to dependency management ([B1-B6]) are not widely followed within the Eclipse ecosystem. No conclusive remark can be made regarding [B3] without further analysis of the packages usage within the bundles' code.

Q3 [B1] and [B3] appear to have a positive impact on classpath size (15.40% and 23.27%, respectively), whereas we observe a close to zero impact for [B2], [B4], [B5], and [B6]. Moreover, five of the selected best practices (*i.e.*, [B1], [B2], [B3], [B4], and [B6]) show an improvement on performance that oscillates between 1.55% and 12.83%. [B5] shows a negative impact of 13.35% relative change for the same variable. The absence of larger gains may be explained by the fact that the time required to build the

classpath is negligible compared to the other phases involved in bundle resolution (*e.g.*, solving dependencies constraints, as can be observed for [B5]).

Q3: **Does each OSGi best practice have an observable effect on the relevant qualitative properties of an OSGi bundle?** Only one third of the OSGi best practices we analyze have a positive impact (of up to ~23% change) on the classpath size of individual bundles. Either way, impact on resolution times does not exceed ±13% relative change for all practices.

Threats to Validity

In principle, the construct of measuring classpath size and resolution time for OSGi bundles can show the presence of a specific kind of impact of a best practice, but not the absence of any other kind of impact. Hence, for where we observed no impact, future analysis of possible other dependent quality factors (e.g. coupling metrics) is duly motivated. However, since the prime goal of OSGi is configuring which bundles to dynamical load into the classpath, any change to OSGi configuration must also be reflected in the classpath. Therefore, in theory, we would not expect any other unforeseen effects when a classpath does not change much. Although relevant, our research methods did not focus on the downstream effects of OSGi best practices on system architecture or object-oriented design quality in source code. However, minor changes to a classpath may have large impact on those aspects, in particular class visibility may impact software evolution aspects such as design erosion and code cloning. In IDEs specifically, performance is not always a key consideration and other aspects of dependency management remain to be studied as future work. With respect to internal validity of the research methods, we calculated classpath size using the OSGi classloader and wiring APIs. Internally, for every bundle, OSGi creates a Java classloader that holds every class local to the bundle,

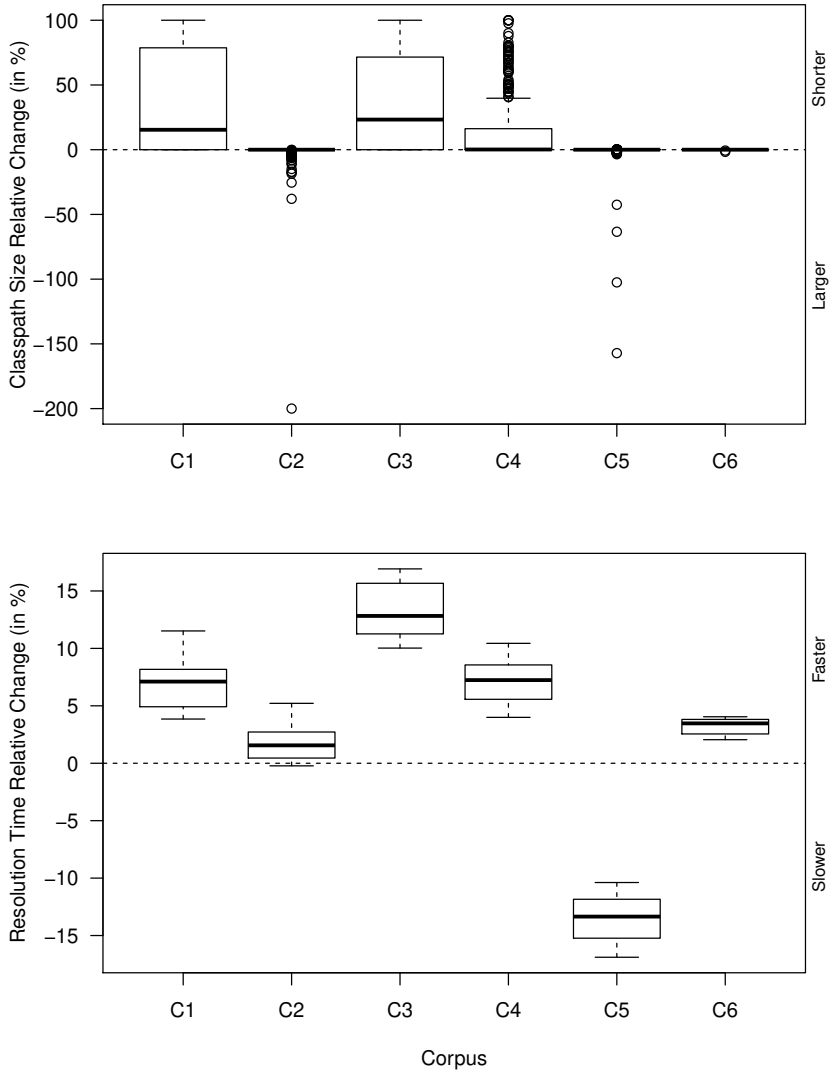


Figure 2.7: Relative change in classpath size and resolution time between the control (C_0) and transformed corpora (C_i)

plus all the classes of the bundles it depends on, regardless of the granularity of the dependencies, their visibilities, *etc.*,. The OSGi classloaders, on the other hand, *hide* classes from the required bundles when necessary, *e.g.*, when a bundle only requires a few packages from another one using the `Import-Package` header. We aimed to calculate classpath size as seen by the OSGi framework itself, but results may vary if we look at Java classloaders instead. Besides, our analysis and transformation tools may be incorrect in some way. We tried to mitigate this pitfall by having our code written and reviewed by several developers, as well as by writing a set of sanity tests that would catch the most obvious bugs. The corpus we use for the analysis may also greatly influence the results we obtain for **Q2**—our conclusions only hold for the Eclipse IDE. Nonetheless, we tried to mitigate this effect as much as possible, for instance by taking into account the specificities of the extensions and extension points mechanism within Eclipse which influences our conclusions for [B1]. Similarly, for **Q3**, a different implementation of the OSGi specification may influence the benchmark results.

2.5 RELATED WORK

Seider et al. [142] explore modularization of OSGi-based components using an interactive visualization tool. They extract information from meta-data files and organize it at different abstraction levels (*e.g.*, package and service). Forster et al. [49] perform static analyses of software dependencies in the OSGi and Qt frameworks. They identify runtime connections using source code analysis. Management of false positives is still a challenge in their research. Both approaches study dependencies in the OSGi framework but are focused on the current state of the framework rather than potential dependencies specification pitfalls or smells. We aim at filling in this gap by empirically studying dependency definition in relation to dependency management best practices in the OSGi ecosystem.

With regards to smell detection in configuration management frameworks, Sharma, Fragkoulis, and Spinellis [143] present a

catalog of configuration management code smells for 4K Puppet repositories on GitHub. Smell distributions and co-occurrences are also analyzed. Jha, Lee, and Lee [73] provide a static analysis tool that aims at detecting common errors made in Manifest files of 13K Android apps. Common mistakes are classified as: misplaced elements and attributes, incorrect attribute values, and incorrect dependency. Karakoidas et al. [79] and Mitropoulos et al. [112] describe a subset of Java projects hosted in the Maven Central Repository (MCR). They use static analysis to compute metrics related to object-oriented design, program size, and package design [79]. FindBugs tool is also used to detect a set of bugs present in the selected projects. They discover that bad practices are the main mistakes made by developers, but do not detail the kinds of recurrent smells. Furthermore, Raemaekers, Van Deursen, and Visser [134] analyze a set of projects hosted in the MCR to check whether they adhere to the semantic versioning scheme. They find that developers tend to introduce breaking changes even if they are related to a minor change. In like-manner, Decan and Mens [34] study `semver` compliance in Cargo, npm, Packagist, and Rubygems. As main takeaways, they find out that `semver` compliance has increased over time for all ecosystems, however, differences do exist among them (*e.g.*, Rubygems tends to be more permissive). They also highlight that these ecosystems assume that patch changes in initial development releases (releases with their version numbers starting with 0) remain backwards compatible—contrary to what `semver` suggests. Rather than beating the aforementioned studies, we offer a complementary view that brings OSGi to the body of studied ecosystems. In addition, we first identify best practices in such an ecosystem and then proceed with our empirical evaluation. Some of the exposed approaches start from a common ground where certain code smells are already known by the community focusing mainly on their identification.

Regarding dependency modeling approaches, Shatnawi et al. [144] aim at identifying dependency call graphs of legacy Java EE applications to ease their migration to loosely coupled architectures. Kula et al. [85] also study JVM-based projects to support

migration to more recent versions of a given open source library. Nevertheless, manifold dependencies are not only specified in source code but also in configuration files. To face this challenge, both approaches parse dependencies from these sources and extract information in their own models: the Knowledge Discovery Meta-model (KDM) [129, 144] and the Software Universe Graph (SUG) [85]. Jezek et al. [72] statically extract dependencies and potential smells from the source code and bytecode of applications. Similarly, Abate et al. [1] dig into *Debian*, *OPAM*, and *Drupal* repositories to identify failing dependencies and to present actionable information to the final user. To this aim, dependencies information is gathered from components' meta-data files, which is then represented in the Common Upgradability Description Format (CUDF) model. In our study, we rely on Rascal to create a data type that stores and represents the dependency information extracted from the `Manifest` file of each of the studied bundles.

Considering software repositories and dependencies description, Decan, Mens, and Claes [35] conduct an empirical analysis of the evolution of *npm*, *RubyGems*, and *CRAN* repositories, later extended to reach a total of 7 software ecosystems (*Cargo*, *CPAN*, *NuGet*, and *Packagist*) [38]. Kikas et al. [82] also consider the first two mentioned repositories and the *Crates* ecosystem. In both cases, dependencies are identified and modeled in dependency graphs or networks to analyze the evolution of the repositories and the ecosystem resilience. Tufano et al. [153] study the evolution of 100 Java projects, finding that 96% of them contain broken snapshots, mostly due to unresolved dependencies. Finally, Williams et al. [158] study more than 500K open source projects taken from *Eclipse*, *SourceForge*, and *GitHub*. They cross-check users needs against projects properties, by means of computing and analyzing metrics provided at forge-specific and forge-agnostic levels. We focus mainly on studying *Eclipse* and whether its developers follow best practices when defining dependencies. Additionally, we consider the observable effects these practices have in terms of classpath size and resolution time.

2.6 CONCLUSION

In this chapter, we first conducted a systematic review of OSGi best practices to formally document a set of 11 known best practices related to dependency management. We then focused on 6 of them and, using a corpus of OSGi bundles from the Eclipse IDE, we studied whether these best practices are being followed by developers and what their impact is on the classpath size and bundle resolution times. On the one hand, the results show that many best practices tend not to be widely followed in practice. We also observed a positive impact of applying two of the best practices (artificially) to classpath sizes (*i.e.*, [B1] and [B3]), from which we can not conclude that the respective best practices are irrelevant. Based on this we conjecture most of the identified advice is indeed relevant. Deeper qualitative analysis is required to validate this. On the other hand, the performance results show that OSGi users can expect a performance improvement of up to $\pm 13\%$ when applying certain best practices (*e.g.*, [B3]). For future work, building on this initial study, we plan to scale up our analysis on other OSGi-certified implementations (*e.g.*, Apache Felix) and other corpora of bundles (*e.g.*, bundles extracted from JIRA or GitHub), and to cross-reference relevant quality attributes on the system architecture and object-oriented design levels with the current OSGi meta-data and bytecode analyses.

3

BREAKING BAD? SEMANTIC VERSIONING AND BREAKING CHANGES

ABSTRACT *Just like any software, libraries evolve to incorporate new features, bug fixes, security patches, and refactorings. However, when a library evolves, it may break the contract previously established with its clients by introducing Breaking Changes (BCs) in its Application Programming Interface (API). These changes might trigger compile-time, link-time, or run-time errors in client code. As a result, clients may hesitate to upgrade their dependencies, raising security concerns and making future upgrades even more difficult. Understanding how libraries evolve helps client developers to know which changes to expect and where to expect them, and library developers to understand how they might impact their clients. In one of the most extensive Java studies to date, Raemaekers, Van Deursen, and Visser investigate to what extent developers of Java libraries hosted on the Maven Central Repository (MCR) follow semantic versioning conventions to signal the introduction of BCs and how these changes impact client projects. Their results suggest that BCs are widespread without regard for semantic versioning, with a significant impact on clients. In this chapter, we conduct an external and differentiated replication study of their work. We identify and address some limitations of the original protocol and expand the analysis to a new corpus spanning seven more years of the MCR. We also present a novel static analysis tool for Java bytecode,* Maracas*, which provides us with: (i) a subset of syntactic BCs between two versions of a*

This chapter is originally published as Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen J. Vinju. "Breaking Bad? Semantic Versioning and Impact of Breaking Changes in Maven Central". In: *Empirical Software Engineering* 27-3 (2022), pp. 1573–7616 DOI: 10.1007/s10664-021-10052-y

library, and; (ii) the set of locations in client code impacted by individual BCs. Our key findings, derived from the analysis of 119,879 library upgrades and 293,817 clients, contrast with the original study and show that 83.4% of these upgrades do comply with semantic versioning. Furthermore, we observe that the tendency to comply with semantic versioning has significantly increased over time. Finally, we find that most BCs affect code that is not used by any client, and that only 7.9% of all clients are affected by BCs. These findings should help (i) library developers to understand and anticipate the impact of their changes; (ii) library users to estimate library upgrading effort and to pick libraries that are less likely to break, and; (iii) researchers to better understand the dynamics of library-client co-evolution in Java.

3.1 INTRODUCTION

Just like any software, libraries evolve to incorporate new features, bug fixes, security patches, and refactorings. It is critical for clients to stay up to date with the libraries they use to benefit from these improvements and to avoid technical lag [55, 171]. When a library evolves, however, it may break the contract previously established with its clients by introducing Breaking Changes (BCs) in its public Application Programming Interface (API), resulting in compilation-time, link-time, or run-time errors. These errors burden client developers given the sudden urgency to fix issues without intrinsic motivation. As a result, clients may hesitate to upgrade their dependencies, raising security concerns and making future upgrades even more difficult [87, 111].

BCs are language-specific: they vary with the syntax and semantics of a particular programming language. In Java, seemingly innocuous changes such as altering the visibility or abstractness modifier of a type declaration, or simply inserting a new method into an abstract class can, under certain conditions, break client code [57]. Most refactoring operations, although essential to maintain and evolve libraries, also induce BCs. Thus, it does not come as a surprise that BCs are widespread in Java libraries [165]. It is, however, essential to realize that not all BCs are intrinsically harmful. Nonetheless, they should not come unannounced and

take clients by surprise. It should be clear for clients what consequences upgrading their dependencies will have on their own software, so they can make an informed decision beforehand.

To this end, Java library developers can leverage various mechanisms to communicate with their clients on the stability of their APIs and the effort required to upgrade to a newer version. These mechanisms enable them to specify *when* and *where* BCs are to be expected. On the one hand, semantic versioning (`semver`) enables developers to use well-defined versioning conventions to classify new library releases as *major* releases (which may introduce BCs), *minor* releases (which may introduce new backward-compatible features but should not introduce any BC), *patch* releases (which should not affect the public API whatsoever), and *initial development* releases (which may break anything at any time) [130]. On the other hand, annotations directly placed on source code elements (*e.g.*, Google's `@Beta` and Apache's `@Internal`) and naming conventions (such as *internal* and *experimental* packages) can be used to indicate that certain parts of the public API are exempt from compatibility guarantees and subject to sudden changes.

Clients who upgrade towards a new major release of a library or early adopters who rely on beta-stage APIs are well aware of the consequences. It is thus crucial to distinguish between libraries that evolve gracefully by introducing BCs only when and where appropriate, and those that "break bad" by introducing BCs in minor and patch releases or in allegedly stable APIs.

In one of the most extensive Java studies to date, Raemaekers, Van Deursen, and Visser dissect backwards compatibility issues in the Maven Central Repository (MCR)[2] with respect to semantic versioning [135]. The study uses the tool `clirr` to infer the list of BCs between two versions of a Java library and measures their impact on client code using the Java compiler itself. The empirical evaluation is carried on a complete snapshot of MCR, up to the year 2011. To name but a few of their findings, the study concludes that: (i) BCs are widespread without regard for versioning conventions; (ii) the adherence to semantic versioning principles has increased over time, and; (iii) BCs have a significant

2 https://search.maven.org/

impact on clients. The relevance and quality of this study for understanding the API-client co-evolution problem motivate us to replicate and expand its protocol and corpus.

In this chapter, we conduct an external and differentiated replication study [101] of the study by Raemaekers, Van Deursen, and Visser [135], which from now on we will refer to as the *original study*. After reviewing the original protocol, we introduce major changes to alleviate some of its limitations and address key threats to its validity. The main differences between our study and the original study are as follows:

- We refine the original protocol by introducing new filters and sanity checks to avoid analysing Maven artefacts that are not used as libraries and versions that are not meant to be used by clients—only 12% of all artefacts in our replication corpus are indeed used as libraries;
- We implement a new tool built atop `japicmp`,[3] MARACAS, more accurate than `clirr`, which we use to analyse Java bytecode and compute the set of BCs between two versions of a library, as well as to compute how client projects are impacted by individual changes;
- We re-analyse the original corpus to assess the impact of our new protocol and tool, and expand the analysis to a new corpus spanning seven more years of the MCR (from 144K Maven artefacts to 2.4M).

We focus on a subset of three of the research questions investigated in the original study which are central to the API-client co-evolution problem, eluding other less relevant questions related to deprecation tags and characteristics of libraries that break more, among others. Our research questions are as follows:

Q1 How are semantic versioning principles applied in the Maven Central Repository in terms of BCs?
Q2 To what extent has the adherence to semantic versioning principles increased over time?
Q3 What is the impact of BCs on clients?

3 https://siom79.github.io/japicmp/

Our results show that, overall, library and client projects on MCR are *not* "breaking bad". First, 83.4% of all library upgrades comply with `semver` principles, introducing BCs only when they are expected. However, 20.1% of non-major releases are breaking, being a potential threat to their clients. Second, the tendency to comply with `semver` practices has significantly increased over time. In particular, the ratio of non-major releases introducing BCs has gradually decreased from 67.7% in 2005 to 16.0% in 2018. Third, only 7.9% of clients are actually impacted by BCs introduced in library releases. In most cases, clients do not use the breaking declarations (*i.e.*, the library declarations affected by BCs)—but when they do, they are very likely to break. These results should help library developers to understand and anticipate the impact of their changes; library users to estimate library upgrading effort and to pick libraries that are less likely to break, and; researchers to better understand the dynamics of client-library co-evolution in Java and prioritize research in the future.

The remainder of this chapter is organized as follows. We first introduce background notions on Maven, `semver`, and backwards compatibility in Java in Section 3.2. We then briefly present the original study in Section 3.3. In Section 3.4, we detail the key differences in the protocol and datasets for our replication study. We discuss our new results in Section 3.5 and then present related work in Section 3.6. We then discuss the key findings and implications of our study in Section 3.7 and finally conclude the chapter and discuss future work in Section 3.8.

3.2 BACKGROUND

In this section, we first introduce some background notions on Apache Maven, APIs, and backwards compatibility in Java. We also discuss the mechanisms available to developers to communicate the stability of their libraries through versioning conventions and source code annotations.

Apache Maven (simply referred to as Maven hereafter) is a build automation tool particularly popular in the Java ecosystem. Maven follows a plugin-oriented architecture that enables developers to specify the dependencies of a particular piece of software and how to build it. When used to build Java projects, it enables developers to convert Java source code to Java bytecode (`.class` files) typically bundled as JARs, which potentially depend on other JARs. These artefacts can be deployed to and retrieved from remote Maven repositories. The most popular Maven repository is the MCR which, as of May 2021, hosts 6,723,367 artefacts.

The cornerstone file defining a Maven project is the Project Object Model (POM) file. Typically, the POM file is an XML file that contains metadata about the current project, its dependencies, and additional configurations required to build it. Listing 3.1 illustrates the typical structure and tags defined within a POM file, using the Spring TestContext Framework as an example. The **`modelVersion`** tag specifies the POM version of the file; the **`groupId`** tag identifies the organization or group that develops the project (`org.springframework`); the **`artifactId`** tag identifies the project itself (`spring-test`); the **`version`** tag specifies the current version of the project (`4.2.5.RELEASE`), and; the **`packaging`** tag specifies how the project is packaged (`jar`). Together, the group, artefact, and version (also known as *project coordinates* and denoted **`groupId`**:**`artifactId`**:**`version`**) uniquely identify a Maven artefact.

Dependencies of a project are declared within the **`dependencies`** tag. Each **`dependency`** points to a unique Maven artefact (using its project coordinates), possibly supplemented with additional metadata. In particular, the **`scope`** tag specifies when the dependency is needed and thus in which classpath(s) it is included (*e.g.*, compile-time, test-time, or run-time dependencies). One can automatically determine which libraries a Maven artefact depends on by parsing its POM file. In Listing 3.1, the Spring TestContext Framework declares a compile-time dependency towards the JavaServlet library version `3.0.1`. Dependencies may employ

Listing 3.1: Excerpt of the POM file of the Spring TestContext Framework project version `4.2.5.RELEASE`

```
<project>
   <modelVersion>4.0.0</modelVersion>
   <groupId>org.springframework</groupId>
   <artifactId>spring-test</artifactId>
   <version>4.2.5.RELEASE</version>
   <packaging>jar</packaging>
   <dependencies>
      <dependency>
         <groupId>javax.servlet</groupId>
         <artifactId>javax.servlet-api</artifactId>
         <version>3.0.1</version>
         <scope>compile</scope>
      </dependency>
   </dependencies>
</project>
```

version constraints (*e.g.*, `[1.0, 2.0)`), letting the dependency resolver find a suitable version within this range.

API Evolution & Backwards Compatibility

An Application Programming Interface (API) is an interface that exposes the set of services from a library that can be invoked by client projects. In Java and other object-oriented languages, this interface consists of programming constructs such as *packages*, *types*, *methods*, and *fields*. To delimit this interface, library developers use visibility modifiers and other dedicated constructs provided by the host language [43].

As an environment changes, software used in such an environment faces the need to change accordingly. This is what Lehman [94] coined as *software evolution*, later formalized as the eight Lehman's laws that synthesize observations about software evolution [53, 99]. Consequently, APIs—being software themselves—undergo continual and progressive change over time. The motivation behind this evolution is to provide more value to users by

patching security issues, adding new features, simplifying the current API, fixing bugs, and improving maintainability [40, 88].

API evolution comes with the introduction of changes that can be classified according to how they affect client projects [43] and specifically whether they ensure *backwards compatibility*. In Java, backwards compatibility is defined at the source, binary, and behavioural levels [40]. *Source compatibility* is checked by the compiler when recompiling a client project with the new version of an API. *Binary compatibility* is checked by the Java Virtual Machine (JVM) during the linking process, as described in Chapter 13 of the Java Language Specification (JLS) [39, 57]. Lastly, *behavioural compatibility* can only be verified at run time to check whether the program exhibits a behaviour that is different from its previous version, without triggering compilation or linkage errors [40].

There are two types of API changes, namely breaking and non-breaking changes. On the one hand, *Breaking Changes (BCs)* are not backwards compatible: client projects using an API entity affected by a BC might break when migrating to a more recent version of the API [43]. On the other hand, *non-breaking changes* are backwards compatible, meaning that they do not trigger any source, binary, or behavioural incompatibility. If an API only introduces backwards compatible changes, it is said to be *stable*. It is important to note that some BCs break several kinds of compatibility (*e.g.*, removing a public method is both source and binary incompatible), but none is a superset of the other [71]. In this chapter, to align with the original study, we only consider *binary compatibility* and the associated set of BCs.

To illustrate how backwards incompatible changes might impact client projects, we refer to the Spring TestContext Framework example. The JavaServlet library releases version `3.1.0` in April 2013. This happens almost two years after its latest major release (*i.e.*, `3.0.1`) in July 2011. This new version introduces backwards incompatible changes that might break client code. Some of those changes include adding new abstract methods to classes and interfaces. One example of such changes is illustrated in Listing 3.2 and Listing 3.3. These changes can potentially impact

Listing 3.2: `HttpServletRequest` in JavaServlet version `3.0.1`

```
public interface
    HttpServletRequest
  extends ServletRequest {
    public String getAuthType();
    public String getMethod();

    [...]
}
```

Listing 3.3: `HttpServletRequest` in JavaServlet version `3.1.0`

```
public interface
    HttpServletRequest
  extends ServletRequest {
    public String getAuthType();
    public String getMethod();
    public String changeSessionId
    ();
    [...]
}
```

Listing 3.4: Broken `MockHttpServletRequest` in Spring TestContext Framework version `4.2.5.RELEASE`

```
public class MockHttpServletRequest implements HttpServletRequest {
  Override public String getAuthType() return
    this.authType;Override public String getMethod() {
    return this.method;
  }
  // MockHttpServletRequest must implement method
    HttpServletRequest.changeSessionId()
}
```

the Spring TestContext Framework in its `4.2.5.RELEASE` version. In some cases, stating that a BC affects client code is straightforward. For instance, removing a type, method, or field that is used by a client will obviously break this client. However, client code may also break when inserting new declarations in the library, for instance when inserting a new abstract method in an interface. This change will break client code if it extends this interface, as illustrated in Listing 3.4. In this case, the `changeSessionId()` method is added to the `HttpServletRequest` class within the JavaServlet library. Given that the `MockHttpServletRequest` class in the Spring TestContext Framework implements such an interface, it will be forced to implement the new abstract method resulting in broken code. The literature often overlooks the BCs

induced by uses of a library in an Inversion of Control (IoC) style (*i.e.*, where the client extends types exposed in the library, following the Hollywood principle "*don't call us, we'll call you!*") [20, 165]. In contrast, we include all of those cases in our analyses. An exhaustive list of the 31 BCs we consider in this chapter is available on the companion webpage.[4]

API Stability Conventions

It is critical for clients to be able to pinpoint which versions and which parts of an API introduce changes that might break their code. *Semantic versioning*, also known as `semver`, is a popular convention to announce the introduction of BCs, and its use is encouraged in many software ecosystems (*e.g.*, npm, RubyGems, Cargo, Maven Central) [34]. This versioning scheme is used to label library versions according to compatibility guarantees. Each version number is specified in the form `<major>.<minor>.<patch>`, where `major`, `minor`, and `patch` are non-negative integers. A change in the `major` version signals the possible introduction of backwards-incompatible changes. Changes in the `minor` or `patch` versions signal the introduction of new features or bug fixes in a backwards-compatible fashion [130]. Initial development releases, which use zero as the `major` version, should also be considered unstable:

> "*[m]ajor version zero (`0.Y.Z`) is for initial development. Anything MAY change at any time. The public API SHOULD NOT be considered stable.*" [130]

Finally, version numbers may be suffixed with hyphen-separated qualifiers specifying pre-releases or build metadata (*e.g.*, `2.1.1-beta2`). At the code level, library developers may use annotations such as Google's `@Beta` and Apache's `@Internal` to signal unstable declarations. For instance, Guava developers state that

4 https://crossminer.github.io/maracas/detections/

> "*APIs marked with the* `@Beta` *annotation at the class or method level are subject to change. They can be modified in any way, or even removed, at any time*,"[5] and Apache POI developers state that "*Program elements annotated* `@Internal` *are intended for [...] internal use only. Such elements are not public by design and likely to be removed, have their signature change, or have their access level decreased [...] without notice*."[6]

Naming conventions on packages (*e.g.*, *internal* and *experimental* packages) are sometimes used for the same purpose [21]. For instance, the following comment is attached to the class `Finalizer` contained in the package `com.google.common.base.internal` of Guava:

> "*While this class is public, we consider it to be *internal* and not part of our published* API*. It is public so we can access it reflectively across class loaders in secure environments*".[7]

This comment highlights the lack of mechanisms for developers to fine-tune the boundaries of their APIs in languages such as Java. Some elements are made public because of technical constraints and not because of the desire to expose these elements in the API; developers must therefore rely on band-aid solutions such as naming conventions. When used in relation to `semver`, these code-level mechanisms enable library developers to delimit a portion of their API that escapes the strict rules regarding backwards compatibility. That is, BCs can be introduced in declarations labelled with these mechanisms without regard for `semver`.

5 https://guava.dev/#important-warnings/
6 https://poi.apache.org/apidocs/dev/org/apache/poi/util/Internal/
7 https://guava.dev/releases/9.0/api/docs/com/google/common/base/internal/Finalizer/

In this section, we briefly introduce the original study object of this replication. We present its goal, main findings, and the protocol used to answer its research questions.

The original study by Raemaekers, Van Deursen, and Visser, entitled "*Semantic versioning and impact of breaking changes in the Maven repository*" and published in *The Journal of Systems and Software* in 2017, investigates whether API developers use versioning practices to signal backwards incompatibility, and how unstable interfaces impact client projects in terms of compilation errors [135]. Although the original study is organized around seven research questions, we decide to focus our effort on three of them that are specifically aimed at understanding the API-client co-evolution problem. In particular, they address the relationship between BCs and versioning conventions, and the impact of BCs on client code. The main findings of the original study are summarized in the following statements. Each of these answers one of the research questions we selected: statement H_i corresponds to question **Qi**. In this chapter, we reuse these results as new hypotheses, which we aim to test under different conditions for replication purposes.

Q1 How are semantic versioning principles applied in the Maven Central Repository in terms of BCs?

H_1 *BCs are widespread without regard for semantic versioning principles.*

Q2 To what extent has the adherence to semantic versioning principles increased over time?

H_2 *The adherence to semantic versioning principles has increased over time.*

Q3 What is the impact of BCs on clients?

H_3 *BCs have a significant impact in terms of compilation errors in client systems.*

On the one hand, the study relies on `clirr` [89] to study backwards compatibility. This tool is used to compute the list of changes between two versions of a Java library. However, the development of `clirr` has stopped in 2005, and Jezek and Dietrich [70] later showed that it is the least sound of a list of 9 tools for BCs detection in Java. On the other hand, to identify the impact of BCs on client code, the original study uses a novel approach that isolates individual changes on the newer version of the API, and injects them one by one in the older version. Then, clients are recompiled against each variant of the old version. The number of compilation errors raised by the Maven compiler is used as a proxy to measure the impact of BCs. As the main corpus, the study uses a snapshot of MCR dated July 2011, consisting of 148,253 JARs and named the Maven Dependency Dataset. In the next section, we dive deeper into the design of our replication study to highlight how it differs from the original study in terms of protocol and corpora.

3.4 Design of the replication study

In this section, we present the protocol of our replication study, summarized in Figure 3.1. The source material of our study is extracted from two different corpora: the Maven Dependency Dataset (MDD) [133], which is used in the original study, and the Maven Dependency Graph (MDG) [14]. These two corpora are snapshots of MCR containing metadata information about artefacts, versions, and dependencies between artefacts. The MDD includes all artefacts from the MCR up to 2011, while the MDG spans seven more years up to 2018. However, due to subtle differences in the methodology used to build these snapshots, the MDD is not strictly a subset of the MDG. In this study, we run the very same analysis protocol on both corpora. On the one hand, re-analysing the MDD enables us to assess the impact of our updated protocol on the results obtained in the original study. On the other hand, analysing the MDG enables us to broaden the scope of analysed artefacts and strengthen our conclusions.

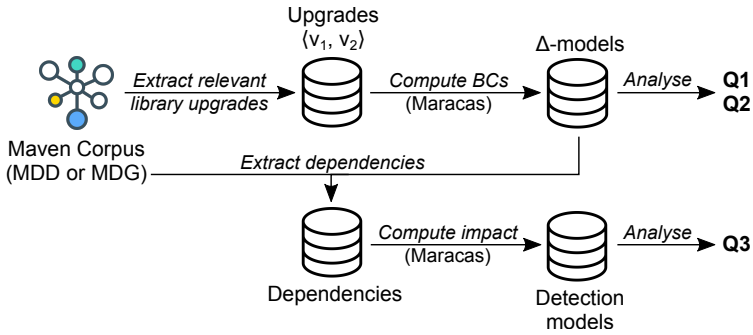


Figure 3.1: Overview of the analysis protocol

First, to answer **Q1** and **Q2**, we extract relevant upgrades for all libraries in the corpora, *i.e.*, pairs of adjacent releases (*e.g.*, JavaServlet versions 3.0.1 and 3.1.0) that conform to the selection criteria presented in Derived Datasets section. The outputs of this task are the *upgrades datasets* $\mathcal{D}_{u}^{o}$ (for the MDD) and $\mathcal{D}_{u}^{r}$ (for the MDG). Then, we use our tool Maracas to compute delta models (Δ-*models*) that store the list of BCs introduced in a particular upgrade between two versions of a library. We analyse the resulting Δ-models in Section 3.5 to answer our first two research questions.

Second, to uncover the impact of BCs on client code and answer **Q3**, we build the dependencies datasets $\mathcal{D}_{d}^{o}$ and $\mathcal{D}_{d}^{r}$, which consist of all clients in the corresponding corpus that might be impacted by the changes identified in a Δ-model (*i.e.*, all artefacts declaring a dependency towards a library upgrade extracted previously). We again use Maracas to identify locations in these clients that are impacted by BCs. The output is stored in a set of *detection models*, where BCs are linked to affected locations in client code. We analyse the resulting models in Section 3.5 to answer our last research question.

The remainder of this section is structured as follows. The Data Extraction section describes the data extraction process of the protocol. Then, the Maracas section gives an overview of our static analysis tool, Maracas. Finally, in the Analysis Approach section, we highlight the key differences between our protocol and

the original study's protocol in terms of data selection, filtering, and treatment.

Data Extraction

In this section, we introduce the two corpora used in this study, together with the datasets derived from them to answer our research questions.

CORPORA Our study relies on two corpora: the Maven Dependency Dataset (MDD) and the Maven Dependency Graph (MDG). The former is used to verify whether the main findings of the original study hold when following a different protocol, while keeping the same base data. The latter is included to assess whether the conclusions of the original study remain valid on a larger population, and whether the phenomenon under study (BCs and `semver` in MCR) has evolved between 2011 and 2018.

THE MAVEN DEPENDENCY DATASET. The MDD is a publicly available snapshot of the MCR dated July 30, 2011 [131]. The corpus contains 148,253 JARs plus additional metadata stored in three different database formats: MySQL, Berkeley DB, and Neo4j [133]. For our purpose, we rely on the metadata stored in the MySQL database. More specifically, we consider the `files` table which stores information about the **`groupId`**, the **`artifactId`**, and the **`version`** of each JAR in the corpus. There is a minor difference in the number of JARs reported in the original study [135] and the dataset paper [133]. We consider the information presented in the latter after manually validating the data exposed in the MySQL database.

THE MAVEN DEPENDENCY GRAPH. The MDG is a graph-based snapshot of all artefacts on MCR as of September 6, 2018 [13, 14]. It is available as a Neo4j graph database where nodes are Maven artefacts and edges are either dependency relations between two artefacts (denoted `:DEPENDS`) or upgrade relations between two artefacts of the same library (denoted `:NEXT`). The MDG contains

2.4M libraries, 9.7M dependency relations, and 2.1M upgrade relations. We rely on the MDG to extract libraries metadata (*e.g.*, versions, clients), and to identify dependency and upgrade relations from which we derive the datasets required for our analyses.

DERIVED DATASETS In what follows, we present the datasets that are derived from each of the two corpora: the *upgrade datasets* $\mathcal{D}^{o}_{u}$ and $\mathcal{D}^{r}_{u}$, and the *dependencies dataset* $\mathcal{D}^{o}_{d}$ and $\mathcal{D}^{r}_{d}$.

UPGRADES DATASETS To answer **Q1** and **Q2**, we derive datasets from our corpora consisting of a set of library upgrades $\langle v1 \rightarrow v2 \rangle$. For our analysis to be accurate and relevant, these library upgrades must fulfill a set of criteria.

As a first filter, we only consider library upgrades $\langle v1 \rightarrow v2 \rangle$ such that v_1 and v_2 are two versions of the same library (uniquely identified by its `groupId` and `artifactId`) which comply with the `semver` scheme. More precisely, these versions must be of the form `X.Y[.Z]`, where `X, Y, Z` $\in \mathbb{N}$, `X` is the major version, `Y` the minor version, and `Z` the (optional) patch version. Versions suffixed with an additional hyphen-separated qualifier often used to tag release candidates, beta versions, or particular build metadata (*e.g.*, `-b01`, `-rc1`, `-beta`, `-issue101`) are discarded, as they are not meant to be used by the general public. In the MDG, for instance, we find 328,448 suffixed versions for 8,251 unique suffixes. The top five most frequent suffixes which we have discarded correspond to release candidates and milestones, namely: `-rc1` (20,007 occurrences, 6.1%), `-rc2` (12,373 occurrences, 3.8%), `-M1` (10,614 occurrences, 3.2%), `-rc3` (7,829 occurrences, 2.4%), and `-M2` (7,780 occurrences, 2.4%). Looking closely at the data, we also notice that a number of versions, even though they technically comply with `semver`, use dates as versions numbers (*e.g.*, `2.5.20110712`). We decide to discard them, as they do not convey the meaning originally intended by `semver`.

Second, v_1 and v_2 must either be adjacent versions (v_2 was released immediately after v_1) or separated with non-`semver`-compliant versions only (all intermediate versions connecting v_1 and v_2 through upgrade relations do not match our criteria). For

instance, considering the three versions $\langle$`3.0.1` $\rightarrow$ `3.1-b01` $\rightarrow$ `3.1.0`$\rangle$, only $\langle$`3.0.1` $\rightarrow$ `3.1.0`$\rangle$ would be included.

Third, we only consider upgrades where v_1 has at least one external client in the dependency graph (either MDD or MDG). An *external client* c of a library version v is a Maven artefact such that c depends on v and belongs to a different `groupId`. This way, we confirm that the artefacts we analyse are indeed used as libraries in practice, and that there are real clients potentially affected by the changes between v_1 and v_2. In the MDG, 56% of all artefacts do not have any client (1,356,413 out of 2,407,395), and only 12% of all artefacts (293,152) have at least one external client. In the MDD, 61% of all artefacts do not have any client (89,772 out of 148,253), and only 17% of all artefacts (24,522) have at least one external client.

Fourth, as we are only interested in the Java language, we discard all JARs that contain code written in any other JVM-based programming language (*e.g.*, Scala, Clojure, Kotlin, Groovy), also hosted on MCR. Our heuristic reads the `source` attribute of `.class` files, set by most bytecode compilers, to retrieve the source file that was used to produce the bytecode and infer the base language. When languages other than Java are detected in a JAR, it is discarded.

Fifth, to ensure that Maracas can process the JARs accurately, we only select library upgrades for which v_1 and v_2 are packaged as JAR files and are compiled with a Java version up to 8 included, as the list and semantics of BCs differ in later versions with the introduction of new language constructs. This differs from the original study, given that Java 8 was released in 2014 and the original snapshot dates from 2011. Thus, we expect to report new types of BCs that were not considered for previous Java versions (*e.g.*, insertion of a new `default` method). The choice of Java 8 is motivated by its popularity: looking at the data, we notice that it is still by far the most popular Java version on MCR.

Lastly, when looking for clients of libraries, we discard all dependency relations that are not in the `compile` scope or `test` scope since they are either not reliably resolvable (*e.g.*, `provided` and `system` dependencies are not hosted on MCR) or are not

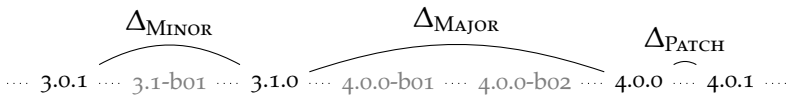


Figure 3.2: Extracting relevant upgrades from the JavaServlet project (`javax.servlet:javax.servlet-api`) between versions `3.0.1` and `4.0.1`

included in the compile-time and link-time classpaths of the client and thus cannot impact binary compatibility (*e.g.*, `runtime` dependencies). Only dependencies in the `compile` and `test` scopes are considered.

As an illustration of the selection process, Figure 3.2[8] depicts how interesting upgrades are picked up between versions `3.0.1` and `4.0.1` of the JavaServlet library, and how Δ-models are classified as major, minor, or patch.

From the original corpus (MDD), we obtain the upgrades dataset $\mathcal{D}_u^o$ consisting of 11,384 upgrade pairs, along with the associated Δ-models computed using Maracas. This dataset differs from the one presented in the original study which contains 126,070 pairs [135]. This difference is explained by the additional filters employed in our protocol: most upgrades are discarded because they do not have any external client; 848 because they contain bytecode generated from other JVM-based languages (2 in Clojure, 71 in Groovy, 76 in Scala, 699 a mix of these or other languages); 641 because of an invalid Java version; 306 because the JARs could not be retrieved from MCR; 2 because Maracas raised an exception when processing the JARs; 31 because they use dates as versions; 309 because the metadata states that v_1 of the library was released after v_2, and; 27 that have an erroneous release date strictly greater than 2011.

From the replication corpus (MDG), we obtain the upgrades dataset $\mathcal{D}_u^r$ consisting of 119,879 upgrade pairs. Most upgrades are discarded because they do not have any external client; 39,986

8 Dotted lines denote upgrade relationships between Maven artefacts. Only major, minor, and patch upgrades are analysed: release candidates, alpha and beta versions, and other qualified versions are discarded. Here, Δ-models are computed for the upgrades $\langle 3.0.1 \rightarrow 3.1.0 \rangle$, $\langle 3.1.0 \rightarrow 4.0.0 \rangle$, and $\langle 4.0.0 \rightarrow 4.0.1 \rangle$.

Table 3.1: Descriptive statistics of the datasets $\mathcal{D}_u^o$ and $\mathcal{D}_u^r$

Dimension	Min.	Q1	Med.	Mean	Q3	Max.
			$\mathcal{D}_u^o$			
External clients	1	1	3	24.31	8	6,153
# of releases	1	1	1	1.23	1	55
Age (in months)	1	1	2	5.55	5	146
Releases/month	0.01	0.25	0.5	0.53	1	18
# of decls.	1	71	280	2,076	1,276	218,274
# of API decls.	1	49	200.5	1,515.6	891.2	159,478
			$\mathcal{D}_u^r$			
External clients	1	1	2	25.31	7	36,186
# of releases	1	7	20	39.4	49	587
Age (in months)	1	10	24	34.38	49	158
Releases/month	0.01	0.4	0.85	1.96	1.79	320
# of decls.	1	79	329	2,519	1,346	586,172
# of API decls.	0	52	220	1,850	974	561,465

because they are written in other JVM-based languages (20 in Clojure, 1,137 in Groovy, 1,359 in Kotlin, 14,402 in Scala, 23,068 a mix of these or other languages); 852 because of an invalid Java version; 10,588 because the JARs could not be retrieved from MCR; 271 because Maracas raised an exception when processing the JARs; 115 because they use dates as versions, and; 2,929 because the metadata states that v_1 of the library was released after v_2.

Table 3.1, Figure 3.3, and Figure 3.4 summarize some descriptive statistics of both datasets. As most distributions are strongly skewed and difficult to visualize (number of clients, size, etc.), Table 3.1 lists their minimum, maximum, median, mean, and quartile values. As an illustration, the top five most popular libraries in $\mathcal{D}_u^r$ are `commons-io 2.4` (36,186 clients), `slf4j-api 1.7.21` (33,582 clients), `commons-codec 1.10` (32,990 clients), `slf4j-api 1.7.12` (25,317 clients), and `slf4j-api 1.7.7` (22,939 clients). We

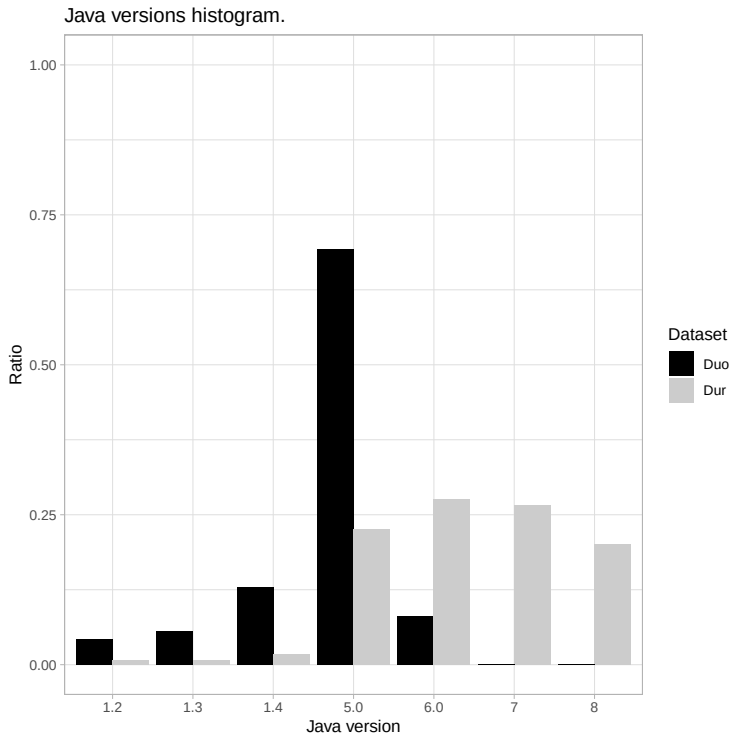


Figure 3.3: Histogram of projects Java versions in MDD and MDG

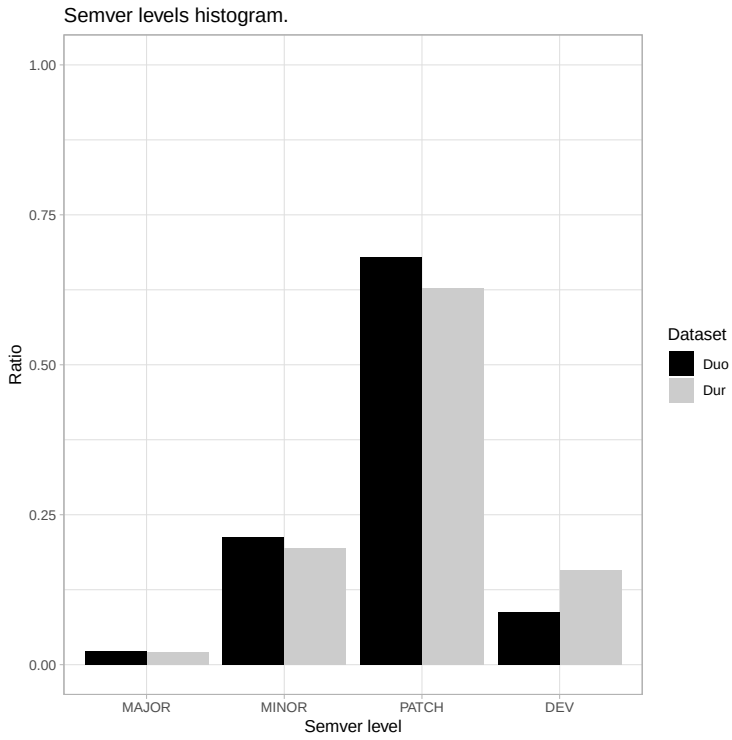


Figure 3.4: Java versions and `semver` levels histograms

refer the reader to the companion webpage and Zenodo repository[9] to access and interact with the datasets.

DEPENDENCIES DATASETS To answer **Q3**, for each upgrade in $\mathcal{D}^{o}_{u}$ and $\mathcal{D}^{r}_{u}$, we compute the list of all clients potentially impacted. That is, all clients that declare a compile-time or test-time dependency towards the library $v1$ in a $\langle v1 \rightarrow v2 \rangle$ upgrade pair, which are potentially affected by the Δ-model between $v1$ and $v2$. We observe that, often, different versions of the same client (*e.g.*, c_{v_1}, c_{v_2}, and c_{v_3}) all depend on the same library version $v1$. In such a case, it is unlikely that c_{v_1} would migrate to v_2 as it is superseded by c_{v_3}. Thus, we only include c_{v_3} in the resulting datasets to avoid counting the impact of the Δ-model between $v1$ and $v2$ on c multiple times. The resulting datasets for $\mathcal{D}^{o}_{u}$ and $\mathcal{D}^{r}_{u}$ are $\mathcal{D}^{o}_{d}$ and $\mathcal{D}^{r}_{d}$, which contain, respectively, 35,539 and 293,817 clients.

Maracas

MARACAS is a new static analysis tool written in Rascal [84] and Java, which allows us to (i) automatically compute a Δ-model between two binary versions of a library and (ii) detect locations in a client binary that are affected by the BCs listed in a Δ-model.

Δ-MODELS A Δ-model is a model that stores the list of BCs between two versions $\langle v_1 \rightarrow v_2 \rangle$ of a library. To compute the Δ-model between two versions of a library, MARACAS internally relies on `japicmp`. `japicmp` is a tool that compares two JAR files and generates a list of BCs between these two JARs. It is able to identify 31 binary incompatible BCs as specified in the JLS 8^{th} Edition [57]. Examples of BCs include removals (*e.g.*, `fieldRemoved`, `methodRemoved`), changes in modifiers (*e.g.*, `methodNowAbstract`, `classNowFinal`) and visibilities (*e.g.*, `fieldLessAccessible`), type

9 https://zenodo.org/record/5221840/

changes (*e.g.*, `methodReturnTypeChanged`), to name a few.[10] As we shall see later in the **Q3** section, some are more critical than others in terms of impact on clients. A Δ-model in `japicmp` follows a tree structure and consists of a list of modified types (classes, interfaces) that recursively contain all modified child elements (*e.g.*, methods, fields, modifiers). Modified elements themselves are labelled with a kind of BC (*e.g.*, `classRemoved`, `fieldNowFinal`). MARACAS transforms `japicmp`'s tree-structured models into a value in Rascal conforming to a Δ-model Algebraic Data Type (ADT), which we use for further analysis. The choice of `japicmp` is motivated by its high popularity and accuracy [70], and by its active community.

Atop `japicmp`, we implement in MARACAS a mechanism to classify declarations in a library as *stable* or *unstable*. Unstable declarations are code elements (*e.g.*, classes, methods, fields) that are explicitly marked with a specific annotation or that are contained in a package not meant to be used by clients. For instance, Google libraries use the annotation `@Beta` and Apache libraries use the annotations `@Internal` to denote code elements that are subject to sudden changes or that should not be used by clients. Similarly, Eclipse packages containing the word `internal` and Java Development Kit (JDK) packages starting with `sun.*` should not be considered part of the API [21]. Because declarations that are explicitly marked as unstable by developers escape the rules of `semver`, it is important to classify stable and unstable declarations. To come up with a list of annotations to consider, we automatically extract all annotations used in the 100 most popular libraries on MCR and manually review their documentation to state whether they are used to delimit unstable APIs. This way, we extract 185 unique annotations and a list of keywords that typically appear in their name (*api, alpha, beta, internal, protected, private, restricted, experimental, dev, access*). Then, we conduct a keyword-based search on the annotations used in the top 1,000 most popular libraries on MCR using as input

10 A complete list and description of these BCs is available on the companion webpage (`https://crossminer.github.io/maracas/emse21/`) and Zenodo repository (`https://zenodo.org/record/5221840/`).

the keywords extracted manually. This way, we extract 1,258 annotations of which 48 match a keyword. The five most common API annotations encountered in these 1,000 libraries are as follows: `@Beta` (1,451 occurrences), `@InterfaceAudience` (1,819 occurrences), `@InternalApi` (1,414 occurrences), `@Internal` (716 occurrences), and `@SdkInternalApi` (607 occurrences). In Maracas, we use the list of extracted keywords to extract package and annotation names used to delimit unstable APIs. Then, we classify each BC in the Δ-models according to whether they affect a stable or unstable declaration of the library.

DETECTION MODELS Prior work uses two main techniques to assess the impact of BCs on client projects: either by tracing library types that are imported in client code (through `import` statements in Java) [12, 165] or by measuring the ripple effect of changes on clients that have already been migrated manually by developers [136]. The former approach largely over-estimates the impact of BCs (a client may not use the broken declaration in the imported type) and the latter requires the availability of migrated clients. The original study employs a novel technique that consists in isolating and injecting each individual BC in the old library's source code. Then, every client is compiled against every ad-hoc version of the library where a single BC is inserted to measure its impact in terms of compilation errors [135]. To the best of our knowledge, however, it is rarely possible to inject individual changes in a library without having to refactor other parts of the API. Removing or renaming a method, for instance, triggers a ripple effect within the library itself, which results in multiple changes being inserted and impacting the clients. Therefore, we hypothesize that this technique overestimates the impact of BCs on clients. Moreover, while measuring the impact of *source incompatible* BCs by counting compilation errors is a valid approach, it is not appropriate to measure the impact of *binary incompatible* BCs which are instead checked by the JVM linker. This motivates the need of having a dedicated tool for identifying BCs impact using static analysis.

MARACAS leverages the Δ-models and Rascal M^3 models to link BCs to affected client declarations using static analysis of binary code. An M^3 model is an ADT that models relations between Java elements, extracted from a JAR file, in immutable binary relations (*e.g.*, containment relations among classes and method declarations, invocation relations among method declarations) [10]. Internally, M^3 relies on the ASM framework[11] to parse Java bytecode and populate the relations. By combining information about breaking declarations in Δ-models and uses of these declarations in client code using M^3 model, MARACAS is able to mark affected client declarations. This detection algorithm is based on the JLS specification [57], and its implementation in MARACAS for each kind of BC is detailed on the companion webpage. The output of this task is a set of *detections* that point to the affected client element, the modified API element, the way it is being used (*e.g.*, `methodInvocation`, `fieldAccess`, `implements`), and the type of BC. In the example of Listing 3.4, the affected client element is `MockHttpServletRequest` which uses the modified API element `changeSessionId()` through an `implements` relation due to a `methodAddedToInterface` change.

LIMITATIONS

OVERRIDDEN METHODS M^3 models generated from Java binaries do not have information related to overridden methods. This means that MARACAS cannot detect BCs impact on code that uses the API through method overriding. For instance, the *method now final* BC breaks clients that override the now-final method, but has no impact on clients that do not override this method. In such a case, due to the lack of information, we follow a pessimistic approach and report a detection in all cases.

EXCEPTIONS HANDLING Information related to thrown and caught exceptions is not part of the M^3 models. MARACAS has no information related to the types of exceptions handled in the `try-catch` statements of clients. Thus, if a method in a library

11 https://asm.ow2.io/

throws a new kind of checked exception, MARACAS is not able to state whether the client will be impacted. In this case, we follow a pessimistic approach and always report a detection.

INHERITANCE HIERARCHY Changing the type of a field, method, parameter, or any other member, or casting might turn out to be a generalization or specialization of the associated type. A type is generalized when it is changed to a supertype and a type is specialized when it is changed to a subtype. In MARACAS we only have access to the client and to the analysed API binaries. Other APIs used by the client are not part of the analysis. Therefore, when a type is changed in a library, we cannot build the whole inheritance hierarchy to state whether this type change corresponds to a generalization or specialization. Without this information, MARACAS might report false positive detections, following a pessimistic approach.

THE SUPER KEYWORD The **`super`** keyword in Java gets a special treatment when detecting errors caused by BCs at the client level. When the visibility of a constructor goes from public to protected, and the constructor is invoked through the use of the **`super`** keyword in the subtype constructor, no error should be reported. However, if the constructor is invoked without using the **`super`** keyword, an error should be reported. MARACAS is not able to differentiate between these two types of invocations, and thus follows a pessimistic approach to always report a detection.

THE STRICTFP AND NATIVE MODIFIERS `japicmp` does not report BCs related to the **`strictfp`** and **`native`** modifiers. Therefore, MARACAS is unable to detect client code affected by changes related to these modifiers, directly affecting the recall of the tool.

VALIDATION MARACAS is the cornerstone tool of our approach as it is used to both compute the Δ-models revealing BCs (using `japicmp` under the hood) and the detection models revealing their impact. To correctly interpret our results, it is essential to analyse the accuracy of MARACAS.

When MARACAS cannot accurately state whether a BC actually has an impact due to the limitations listed above, the approach we follow is to always over-approximate the detections at the cost of sometimes reporting false positives, while avoiding any false negative. This means that the results we obtain regarding BCs and their impact might be slightly overestimated, but they are not underestimated.

Binary compatibility is checked by the JVM linker. To evaluate MARACAS, we aim to compute its accuracy by comparing MARACAS detections with the error messages thrown by the JVM linker itself when encountering code impacted by BCs. The JLS states that binary compatibility should be checked during the loading and linking phases of the JVM, but the choice of implementing lazy or eager initialization of classes is up to the implementors. In practice, the reference implementation (OpenJDK HotSpot) implements lazy loading and waits for class initialization to load and link a class. It follows that, to record the errors thrown by the linker, it is necessary to *execute* a Java program making use of breaking declarations. In addition, the Java linker throws an exception and stops processing the class after the first error is encountered, so the executed Java programs should contain only a single use of a breaking declaration to record all errors.

To evaluate the accuracy of MARACAS, we thus reuse and extend the benchmark proposed by Jezek and Dietrich [70]. Their benchmark consists of a library $v1$, a library $v2$ that breaks $v1$ in all possible ways, and a client c that uses all declarations of $v1$ in various ways. In order to trigger linking errors, client c consists of a set of `Main` files. In the original version of the benchmark, however, some `Main` files use several declarations of the library $v1$: if the first use fails, the others are not evaluated by the linker. Thus, we split the client c into more cases so that every case exercises a single declaration of $v1$.

Our final benchmark for detections consists of 345 cases, where each case consists of a Java entry point (`Main` file and method) that exercises one particular BC and one particular way of using it. The benchmark script first compiles $v1$, $v2$, and c in their binary form (JAR), and then attempts to run every single `Main` file in c,

replacing $v1$ with $v2$ in its classpath. Whenever a linking error is encountered, it is written to disk. Then, we run MARACAS giving it $v1$, $v2$, and c as inputs to get the list of detections. If a detection matches an error reported by the linker, it is a true positive, if it does not match any linker error it is a false positive, and if there is no detection for a particular linker error it is a false negative. To support future research, we have made our benchmark publicly available on the companion webpage.

Out of the 345 cases, the JVM linker reports 132 errors and MARACAS reports 135 detections. Out of the 135 detections, 130 are true positives and 5 are false positives. There are 2 false negatives. In this benchmark, MARACAS achieves a precision of 96.3% and a recall of 98.5%. The five false positives are due to the limitations listed in the section above. The two false negatives are due to a limitation of `japicmp`, which does not compute BCs related to the **`strictfp`** and **`native`** modifiers.

In addition to this benchmark, we developed a test suite as part of MARACAS consisting of 402 test cases. Using our own test cases, we highlighted a bug in `japicmp` which we fixed through a pull request accepted by the project maintainers.[12]

Analysis Approach

In this section, we compile some of the most relevant aspects of the analysis performed in this study, and we contrast them against the original study (*cf.* Table 3.2). We refer the reader to Section 4 of the original work for further information.

BACKWARDS COMPATIBILITY The original study computes binary incompatible changes with `clirr`. However, `clirr` is not able to report BCs related to exceptions and generics, and misinterprets changes related to inheritance and other modifiers [70]. In this study, we use `japicmp` to compute both source and binary incompatible changes. The latter performs better than `clirr` according to Jezek and Dietrich [70]. Although `japicmp` is unable to

12 https://github.com/siom79/japicmp/pull/251/

Table 3.2: Main commonalities and differences between the original study and the replication study protocols

	Original study	Replication study
Corpus	MDD [133]	MDD+MDG [14]
Corpus date interval	2005–2011	2005–2018
Backwards compatibility	Binary	Binary
Static analysis tool	`clirr`	`japicmp`
Compared versions	Adjacent	Adjacent
Versioning scheme	`semver`	`semver`
Languages	JVM-based	Java
Clients per library	$\geqslant 0$	$\geqslant 1$
Client impact detection	Compilation errors	Static analysis
Code-level mechanisms	`@Deprecated`	Annotations, package naming

identify changes related to generics—which, due to type erasure in Java, does not impact binary compatibility analysis—it accurately reports all changes related to exceptions and inheritance. In addition, it is more accurate than `clirr` when reporting on changes associated with modifiers. We also contributed to the tool by fixing a bug related to the detection of modifier changes. Other committers have also made some contributions to improve `japicmp` accuracy in recent times. With these changes, one new case within the Jezek and Dietrich [70]'s benchmark passes: the decrease of a nested interface access modifier from **`public`** to **`protected`**.

LIBRARY AND VERSION SELECTION As is the case in the original study, we only compute deltas between adjacent versions of an API, which strictly follow the `X.Y[.Z]` version convention. However, in contrast, we only consider artefacts that have at least one external client on the MCR. This way, we ensure that the artefacts we analyse are indeed used as libraries by clients. This is a significant difference with the original study, as only 17% of all artefacts in the MDD and 12% of all artefacts in the

MDG have at least one external client. We also account for initial development releases (`0.Y[.Z]`), which are not considered in the original study.

PARALLEL BRANCHES AND MAINTENANCE RELEASES The original study does not account for maintenance releases that happen in practice. For instance, suppose version `2.4` is released after `3.0`, as a maintenance release for the `2.X` branch. Using release dates to infer the order of versions, BCs for the upgrade $\langle$`3.0` $\rightarrow$ `2.4`$\rangle$ would be computed, even though this is not the expected behaviour. Instead, we employ the MDG which properly represents these upgrade relations regardless of release date, and accounts for maintenance releases.

BREAKING CHANGES IMPACT The original study detects client code affected by BCs by means of injecting changes in the source code of an API. After the code injection, the client is compiled against the modified API and new compilation errors are recorded. There might be pre-existing compilation errors before changes are injected in the API. These errors are intentionally excluded from the analysis. However, this approach introduces a set of limitations that can affect the outcome of the study, some of them have already been identified by Raemaekers, Van Deursen, and Visser [135]. We describe them as follows: (i) injecting changes in isolation might introduce compilation errors that must be fixed. In some cases, multiple changes should be injected at the same time in order to avoid introducing compilation errors; (ii) pre-existing errors might hide new errors related to the injected BC; (iii) reporting on compilation errors gives us an idea of how source compatibility is affected. However, binary compatibility is not equivalent to source compatibility. Compilation errors account for source incompatible changes and cases of binary incompatibility that are shared between both sets; (iv) we cannot guarantee that all expected compilation errors are reported by the compiler. For instance, if at least one imported package in a class cannot be found, the compiler will not reach subsequent errors [135]; (v) when injecting changes in the API, it is difficult

to manage cases where one piece of code is related to multiple BCs. For instance, we inject a piece of code related to changes C_1 and C_2 in a given API. If we want to measure the impact of both changes, we will end up with the same number of compilation errors for both cases without discriminating their origin, and; (vi) the compiler cannot tell the cause or the BC that produces a given error.

Overall, given the widespread use of the language features involved in the above possible causes of inaccuracy, and their relation to the research questions, we believe that developing and using a more accurate tool will have a significant impact on the outcome. Thus, we use Maracas to detect affected code on the client side and report on its accuracy.

DEPRECATED AND UNSTABLE INTERFACES The original study uses `@Deprecated` annotations to identify unstable interfaces. Occurrences of this annotation are computed, except for nested cases. This means that the analysis will not detect declarations within a deprecated class, where explicit annotations have not been used [135]. These cases are considered in the present work. Moreover, there are also other mechanisms to signal unstable interfaces. We argue that other annotations, such as Google's `@Beta` and Apache's `@Internal` annotations, are also used to signal instability in an API. In addition, naming conventions on packages are also used for the same purpose. We then include the detection of these cases to perform a deeper analysis of the derived datasets.

3.5 RESULTS & ANALYSIS

In this section, we analyse the data extracted using the protocol described in Section 3.4. Each subsection describes the method, results, and analysis of a particular research question.

***Q1**: How are semantic versioning principles applied in the Maven repository in terms of BCs?*

Method

With **Q1**, we analyze when and where BCs happen and with which frequency. We attempt to distinguish expected and unexpected BCs according to the `semver` principles and the use of code-level mechanisms to signal unstable APIs. In a first step, we seek to highlight the impact of the updated protocol described in Section 3.4 on the results reported in the original study. To do so, we compute the Δ-models between every $\langle v_1 \rightarrow v_2 \rangle \in \mathcal{D}^{o}_{u}$, while distinguishing among major, minor, patch, and initial development releases. In a second step, to assess whether the results hold on the larger dataset $\mathcal{D}^{r}_{u}$ comprising seven more years of MCR, we run the same analysis for every $\langle v_1 \rightarrow v_2 \rangle \in \mathcal{D}^{r}_{u}$. The Δ-models distinguish between BCs that are introduced in stable and unstable parts of the APIs, according to code-level annotations (such as `@Beta` or `@Internal`, *cf.* Section 3.4), as well as naming conventions (such as *internal*). As BCs in unstable parts of an API are to be expected, only BCs introduced in the stable parts are included.

To know where to expect BCs or in which type of upgrades, we compare the percentage of breaking upgrades per `semver` level. Alongside `semver` categories (*i.e.*, major, minor, patch, and initial development), we also consider the group of non-major releases as a whole (*i.e.*, minor and patch releases combined). Then, to know how many BCs are usually introduced in each `semver` level, we consider the distribution of BCs over all groups.

We wrap up the analysis of **Q1** by studying the frequency of each type of BC. From these results, we identify the most common BCs in our datasets and compare these results against the ones presented in the original study.

Results

BREAKING UPGRADES Table 3.3 highlights the main results obtained for **Q1**. The first block lists the results reported in the

original study [135], the second block the results obtained for $\mathcal{D}_u^o$ (for replication purposes), and the third block the results obtained for $\mathcal{D}_u^r$.

First of all, we compare the results reported in the original study against those obtained for $\mathcal{D}_u^o$. While we report a similar number of breaking upgrades overall (*cf. Total* row: 32.2% in $\mathcal{D}_u^o$ vs. 30.0% in the original study), we observe that the difference in the ratio of breaking upgrades per `semver` level is stronger (*cf. Major*, *Minor*, *Patch*, and *Dev* rows). As expected, most major upgrades in $\mathcal{D}_u^o$ introduce BCs (72.7%), which contrasts with the results obtained in the original study (35.9%). While the original study reports that there are as many breaking major upgrades as breaking minor upgrades, we observe a sharper difference between these two levels in the same corpus: 72.7% of major upgrades (vs. 35.9% in the original study) and 50.1% of minor upgrades (vs. 35.7% in the original study) break in $\mathcal{D}_u^o$. With regards to patch upgrades, we observe a similar percentage of breaking cases (24.2% in $\mathcal{D}_u^o$ vs. 23.8% in the original study). 39.3% of the initial development upgrades, which are not considered in the original study, are breaking. Overall, we report that 30.5% of non-major releases[13] do not conform to `semver`, which matches the results obtained in the original study (29.0%). Upgrades that comply with the scheme principles (*i.e.*, major upgrades, initial development upgrades, and non-breaking minor and patch upgrades) represent 72.8% of all upgrades in $\mathcal{D}_u^o$.

For $\mathcal{D}_u^r$, which spans seven more years of the MCR and comprises ten times more upgrades, we observe that the tendency to comply with `semver` improves. The ratio of breaking upgrades is lower overall (22.0% in $\mathcal{D}_u^r$ vs. 32.2% in $\mathcal{D}_u^o$), and for each level: 61.8% for major upgrades, 37.9% for minor upgrades, 14.6% for patch upgrades, and 26.7% for initial development upgrades. This amounts to 83.4% of all upgrades conforming to `semver` principles. However, 20.1% of non-major upgrades are still breaking and thus do not comply with the versioning conventions.

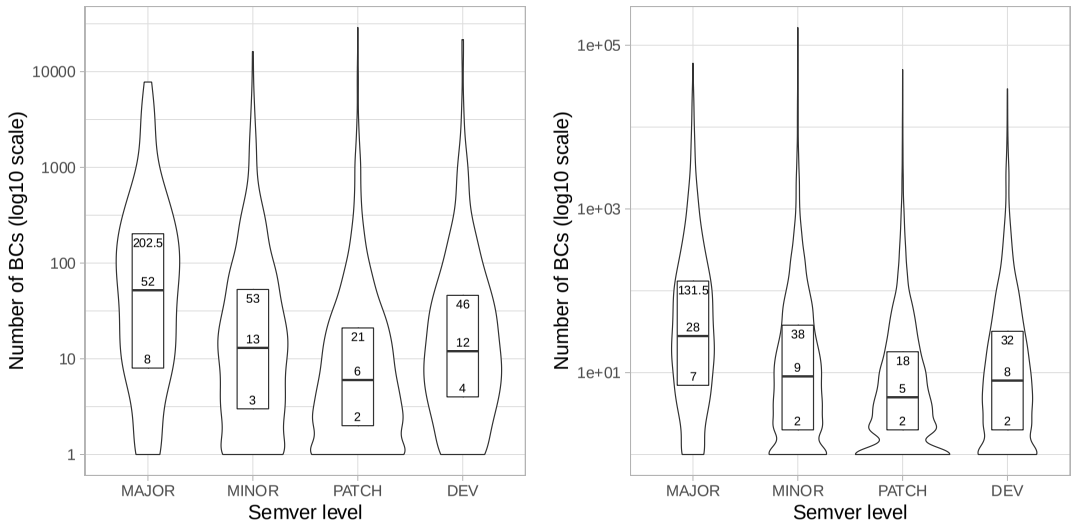
13 Non-major releases only consider patch and minor releases. Initial development releases are excluded from this category as these ones are allowed to introduce BCs at any moment.

Table 3.3: Total and breaking upgrades in the original study, $\mathcal{D}_u^o$, and $\mathcal{D}_u^r$ datasets

Level	Total		Breaking	
	Count	%	Count	%
Original study				
Major	11,892	14.8	4,268	35.9
Minor	29,957	37.2	10,690	35.7
Patch	38,740	48.1	9,239	23.8
Dev	n/a	n/a	n/a	n/a
Non-major	68,697	85.3	19,929	29.0
Total	80,589	100	24,197	30.0
$\mathcal{D}_u^o$				
Major	253	2.2	184	72.7
Minor	2,413	21.2	1,228	50.1
Patch	7,728	67.9	1,870	24.2
Dev	990	8.7	389	39.3
Non-major	10,141	89.1	3,098	30.5
Total	11,384	100	3,671	32.2
$\mathcal{D}_u^r$				
Major	2,431	2.0	1,503	61.8
Minor	23,309	19.4	8,837	37.9
Patch	75,282	62.8	11,031	14.6
Dev	18,857	15.7	5,036	26.7
Non-major	98,591	82.2	19,868	20.1
Total	119,879	100	26,407	22.0

The difference in results between $\mathcal{D}_u^o$ and $\mathcal{D}_u^r$ suggests that the adherence to semantic versioning may have increased over time. This intuition is investigated further in the next research question **Q2**.

NUMBER OF BC To study the frequency of BCs introduction, we first look at the number of BCs introduced in breaking releases,



(a) $\mathcal{D}_u^o$ (b) $\mathcal{D}_u^r$

Figure 3.5: Violin plots of the number of BCs in breaking upgrades per `semver` level

i.e., releases that contain at least one BC. Figure 3.5[14] shows the distribution in a logarithmic scale of BCs per `semver` level for breaking releases in $\mathcal{D}^{o}_{u}$ and $\mathcal{D}^{r}_{u}$. Looking at the median values, we notice that the number of BCs is higher in major upgrades (52 in $\mathcal{D}^{o}_{u}$ and 28 in $\mathcal{D}^{r}_{u}$). Minor and initial development upgrades tend to have a similar number of BCs in both datasets (13 and 12 in $\mathcal{D}^{o}_{u}$, and 9 and 8 in $\mathcal{D}^{r}_{u}$, respectively). Patch upgrades introduce the least number of BCs (6 in $\mathcal{D}^{o}_{u}$ and 5 in $\mathcal{D}^{r}_{u}$). This suggests that non-major development releases not only do break less often, they also tend to introduce fewer BCs when they do.

BC TYPES Figure 3.6 and Figure 3.7 present the ratio of BC types (*e.g.*, *method removed*, *field removed*) using a bar plot, for both $\mathcal{D}^{o}_{u}$ and $\mathcal{D}^{r}_{u}$. BCs are discriminated by `semver` levels and ordered from the most to the least frequent. We notice that the ratio of BC types is consistent across `semver` levels (except perhaps for the *method return type changed* and *field type changed* in the original dataset). In both datasets, the 10 most common BC types and their associated ratios remain mostly unchanged: *method removed*, *field removed*, *interface removed*, *constructor removed*, *superclass removed*, *class removed*, *interface added*, *method added to interface*, *method return type changed*, and *field type changed*. Similarly, in both datasets, the BC kinds ranked after *method return type changed* are very rare. Interestingly, the five most frequent BC kinds are all related to the removal of API entities. We cannot directly compare these results with the original study, given that not all reported BC types are identified in the same way between the underlying tools (*i.e.*, `clirr` and `japicmp`). However, our observations align with the results reported in the original study where method, class, and field removal headed the list. Naturally, the BCs *method new default* and *method abstract now default* do not occur in the $\mathcal{D}^{o}_{u}$ dataset, as they relate to the **`default`** operator which was only introduced in Java 8 (2014), while the most recent artefacts in $\mathcal{D}^{r}_{u}$ date back to 2011.

14 Inside each violin plot, we display the first quartile, the median, and the third quartile.

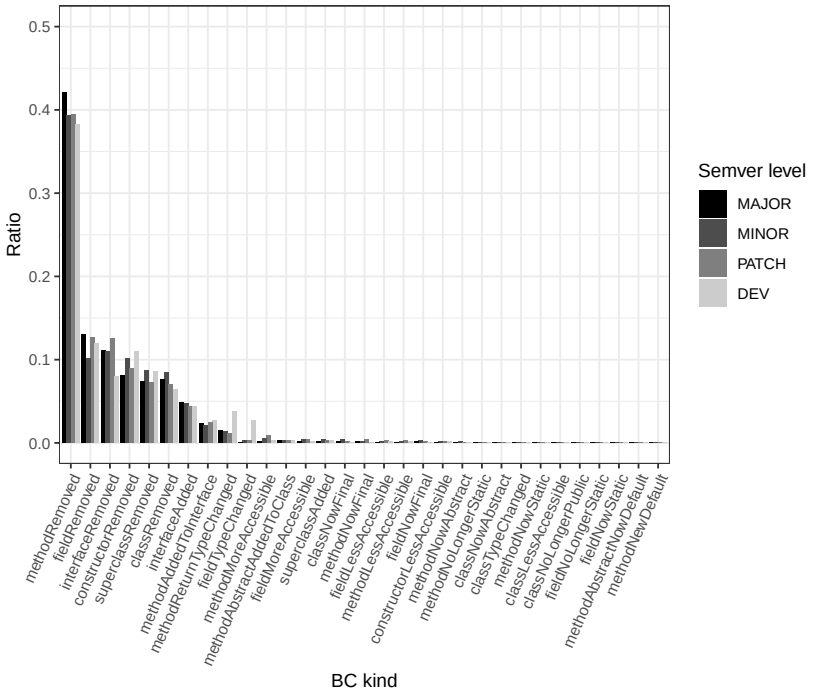


Figure 3.6: BC types frequency per `semver` level in $\mathcal{D}_u^o$

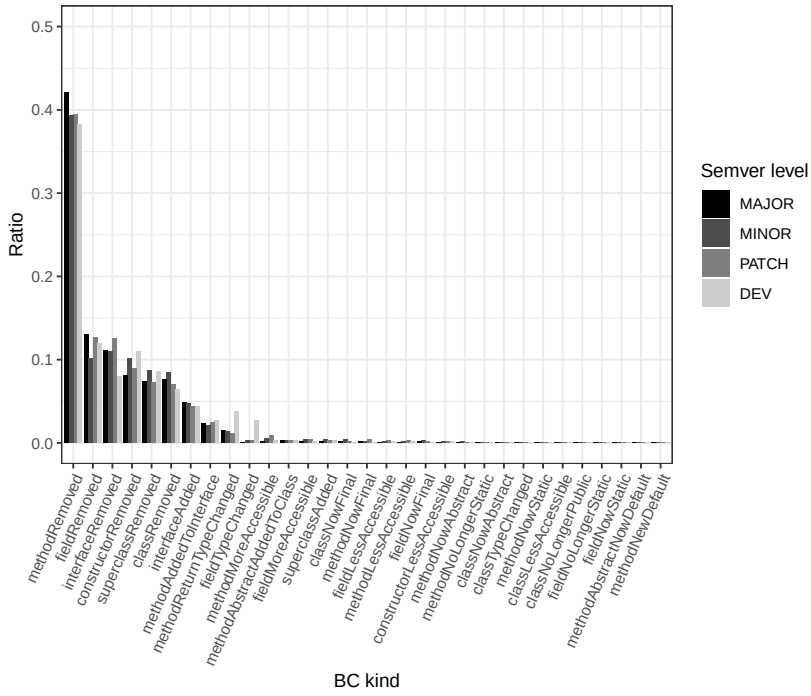


Figure 3.7: BC types frequency per `semver` level in $\mathcal{D}_{u}^{r}$

Analysis

Artefacts on MCR do not strictly follow `semver`, as an important ratio of non-major upgrades are breaking (20.1% in $\mathcal{D}^{r}_{u}$), confirming the main result from the original study. However, we observe a sharp difference in the ratio of breaking upgrades per `semver` level with our protocol (61.8% of major upgrades, 37.9% of minor upgrades, 14.6% of patch upgrades, and 26.7% of initial development upgrades break). This contrasts with the original study, which reports a similar ratio of breaking upgrades for major and minor cases, with patch upgrades only slightly more stable. In general, differences between the results reported in the original study and $\mathcal{D}^{o}_{u}$ are explained by the additional filters considered in our protocol, the increased accuracy of MARACAS and `japicmp` in detecting BCs, and the consideration of APIs annotated as unstable at the source code level (*cf.* Analysis Approach section). Differences between $\mathcal{D}^{o}_{u}$ and $\mathcal{D}^{r}_{u}$ are mainly due to the increased time span and the number of artefacts. This rationale applies to the forthcoming analyses. Our results suggest that `semver` principles are followed to some extent in practice, as 83.4% of the library upgrades we analyse do comply with the backwards compatibility requirements of the versioning scheme.

We also notice that major upgrades not only result in a higher number of breaking cases but also tend to introduce more BCs per breaking upgrade. Patch upgrades are the ones introducing the least number of BCs. This suggests that, even when a non-major release is breaking, the amount of work ending on the clients' shoulders is not as high as for a major release. Finally, the most common BCs are aligned with results presented in the original study: removal of API members is the most common type of BC occurring in libraries.

Q1: **How are semantic versioning principles applied in the MCR? H_1** asserts that "*BCs are widespread without regard for versioning principles*." From our analysis, we conclude that although `semver` principles are not always strictly applied (20.1% of non-major releases are breaking), they are largely followed: 83.4% of

all upgrades comply with `semver` regarding backwards compatibility guarantees, and the differences between `semver` levels are notable. Not only do minor and patch releases break less often than major releases, they also introduce fewer BCs. This leads us to reject $\mathbf{H_1}$.

***Q2**: To what extent has the adherence to semantic versioning principles increased over time?*

Method

To answer **Q2**, we first study how the ratio of breaking upgrades for the various `semver` levels has evolved over time, aggregated per year. The ratio of breaking upgrades corresponds to the number of upgrades containing at least one BC over the total number of upgrades per `semver` level. We still consider the four different `semver` levels plus the analysis of non-major upgrades as a whole. Reported results are based on the data extracted from the $\mathcal{D}^{r}_{u}$ dataset. The latter spans fourteen years of Maven artefacts from MCR (2005 to 2018 included). We then contrast these results against the ones reported in the original study. Studying the evolution of the adherence to semantic versioning principles is especially relevant as the `semver` specifications are fairly recent in the history of MCR: `semver` 1.0.0 was released in 2009 and `semver` 2.0.0 in 2013. It is thus likely that the principles of `semver` did not percolate yet in the dataset used in the original study.

Results

Figure 3.8[15] depicts how the ratio of breaking upgrades has evolved for major, minor, patch, initial development, and non-major levels. Overall, the ratio of breaking upgrades, regardless of the `semver` level, tends to decrease. As expected, the ratio of

15 Each data point aggregates the number of breaking upgrades of the given type for an entire year. A vertical line delimitates the periods of the original and updated datasets.

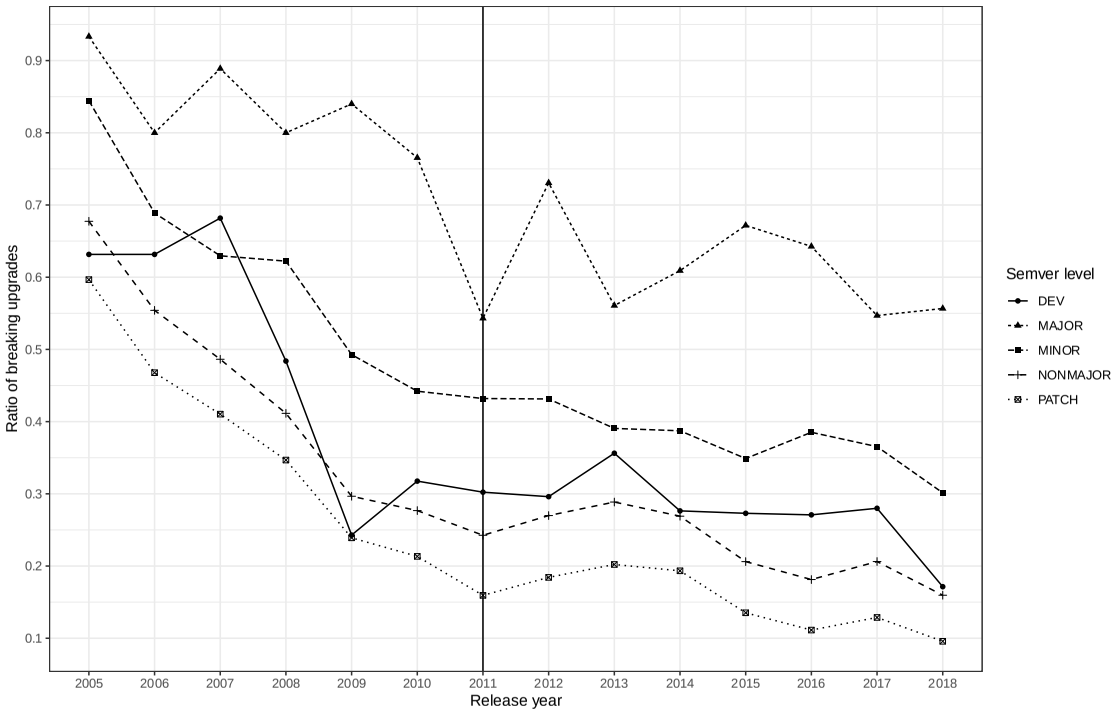


Figure 3.8: Evolution of the ratio of breaking upgrades per `semver` level in $\mathcal{D}_{u}^{r}$

breaking upgrades of major and initial development levels is more chaotic since BCs are allowed in these releases. Nevertheless, even major and initial development releases contain fewer BCs in 2018 than in 2005. One possible hypothesis is that clients' tolerance for BCs has decreased over time and that libraries are avoiding them more and more, even when allowed. Additionally, different ecosystems such as npm, RubyGems, Cargo, MCR [34], and GitHub are encouraging library developers to follow semver guidelines. In particular, GitHub explicitly states in its official documentation: "*We recommend naming tags that fit within semantic versioning.*"[16] Another relevant example is the Maven ecosystem, which offers the `maven-release-semver-policy` plugin to enforce the use of `semver` when releasing a project.

Over 14 years, the ratio of breaking minor upgrades has decreased almost by a factor of three (from 84.4% to 30.1%) and the ratio of breaking patch upgrades has decreased by a factor of six (from 59.7% to 9.6%). This is to be contrasted with the results of the original study, which finds that, from 2005 to 2011, the number of non-major breaking upgrades has decreased from 28.4% to 23.7%. Conversely, we find that non-major breaking upgrades have decreased from 67.7% in 2005 to 16.0% in 2018.

Analysis

As Maracas and `japicmp` are able to detect more types of BCs, the percentages we report are higher than the ones reported in the original study, which makes the decrease of the ratio of breaking non-major upgrades much steeper than originally reported (a 44% reduction instead of a 5% decrease). However, the decrease in the extended period is less evident: only a 9.2% decrease. Visually, 2011 appears as a turning point w.r.t. the decrease of breaking non-major upgrades, as the slope is less steep after this date. We found no plausible explanation for this phenomenon. Overall, it confirms once more the statement that even though not all artefacts on MCR follow `semver` guidelines,

16 https://docs.github.com/en/github/administering-a-repository/releasing-projects-on-github/managing-releases-in-a-repository/

there is an increasing tendency to comply with the versioning scheme principles.

Q2: **To what extent has the adherence to semantic versioning principles increased over time?** $\mathbf{H_2}$ states that "*The adherence to versioning principles has increased over time.*" Our results confirm the results of the original study. They also show that the improvement over time is much higher than initially reported for the 2005–2011 period. The tendency persists in the 2011–2018 period, although the slope is less steep. Thus, we cannot reject $\mathbf{H_2}$.

***Q3**: What is the impact of BCs on clients?*

Method

In **Q3**, we investigate to which extent BCs introduced in Java libraries impact their clients on MCR. More formally, for every Δ-model computed between versions $\langle v_1 \rightarrow v_2 \rangle$ of a given library (*cf.* **Q1**), we extract all the clients c declaring a compile-time or test-time dependency towards v_1 to uncover the impact $\Delta\langle v_1, v_2 \rangle$ would have on c if it was updated to $v2$. Concretely, we use the static analysis capabilities of Maracas to pinpoint which code locations in c are impacted by individual BCs of the corresponding Δ-model (*cf.* Maracas section).

The impact a BC has on client code varies according to if and how the client uses the declaration affected by the change (*cf.* Listing 3.4). Hence, determining the impact of BCs requires a deep understanding of how clients and libraries interact. For every client, we classify the impact of each individual BC in one of three categories: (i) the declaration affected by the change is not used in client code (*unused*); (ii) the declaration affected by the change is used in a non-breaking way (*non-breaking*), and; (iii) the declaration affected by the change is used in a breaking way (*breaking*).

As with the first two research questions, we report results for both the MDD and the MDG corpora. The datasets $\mathcal{D}_d^o$ and $\mathcal{D}_d^r$

contain, respectively, 35,539 and 293,817 clients which are potentially impacted by a Δ-model extracted in **Q1**. As it would be impractical to analyse these cases exhaustively, we resort to analyse a subset of them by performing a random sampling. The question we ask for each case is: does client c break when upgrading from version $v1$ to version $v2$ of a library it uses? To answer this question with a confidence level of 99% ($c = 0.99$), an error margin of 1% ($e = 0.01$), and an estimated proportion of the population $p = 0.5$ (the more conservative value yielding the largest sample size) of broken clients, we apply the standard Cochran's sample size formula to determine sample sizes for each kind of upgrade (*i.e.*, major, minor, patch, and initial development). Then, we draw upgrades at random, without replacement, from the set of all upgrades, all major upgrades, all minor upgrades, all patch upgrades, and all initial development upgrades, yielding the samples depicted in Table 3.4. For each tuple $\langle c, v_1, v_2 \rangle$ in the corresponding samples, we use MARACAS to compute their detection models and analyse the impact of $\Delta(v_1, v_2)$ on client c, distinguishing among *unused* declarations, *non-breaking* uses, and *breaking* uses.

To uncover which kinds of upgrade break clients the most, we compare the percentage of overall broken clients per `semver` level. Afterwards, we consider the number of broken locations per client for each level.

Results

BROKEN CLIENTS Table 3.4 depicts the size of each `semver` sample and the number and proportion of broken clients for both $\mathcal{D}^{o}_{d}$ and $\mathcal{D}^{r}_{d}$. We observe that 9.5% and 7.9% of all clients for $\mathcal{D}^{o}_{d}$ and $\mathcal{D}^{r}_{d}$, respectively, would break if they upgraded their dependency to the next release.

Taking into account the kind of upgrade yields interesting results: in both datasets, initial development upgrades lead to the highest percentage of broken clients (18.4% for $\mathcal{D}^{o}_{d}$ and 16.8% for $\mathcal{D}^{r}_{d}$), followed by major (12.7% for $\mathcal{D}^{o}_{d}$ and 11.7% for $\mathcal{D}^{r}_{d}$), minor (11.9% for $\mathcal{D}^{o}_{d}$ and 7.8% for $\mathcal{D}^{r}_{d}$), and finally, patch upgrades (6.0%

Table 3.4: Samples derived from the population of dependencies

	All	Major	Minor	Patch	Dev
		$\mathcal{D}^{o}_{d}$			
Population size	35,539	2,861	13,444	17,425	1,809
Sample size	11,310	2,440	7,426	8,498	1,631
Broken clients	1,076	309	883	514	300
% broken clients	9.5%	12.7%	11.9%	6.0%	18.4%
		$\mathcal{D}^{r}_{d}$			
Population size	293,817	29,847	111,830	123,286	28,854
Sample size	15,701	10,663	14,445	14,621	10,533
Broken clients	1,237	1,250	1,130	735	1,772
% broken clients	7.9%	11.7%	7.8%	5.0%	16.8%

for $\mathcal{D}^{o}_{d}$ and 5.0% for $\mathcal{D}^{r}_{d}$). This indicates that clients are more likely to break when upgrading to a version of a library that is potentially breaking according to `semver` conventions, with initial development releases being the most problematic. Conversely, clients that upgrade to minor and patch releases are less likely to be affected.

As we resort to random sampling to estimate the proportion of broken clients, we use statistical inference to assess our raw results. For the sake of simplicity, we only perform the statistical analysis for the $\mathcal{D}^{r}_{d}$ dataset. We have the following null hypothesis: "*the proportion of broken clients is the same across each* `semver` *level of library upgrades*". Note that in the remainder of this section, we use * to label the significance of the p-values using the following scale: * indicates a $p < 0.1$, ** a $p < 0.05$ and *** a $p < 0.01$. We run a χ^2 (chi-squared) test on the table containing the number of broken and non-broken clients for each level. This test yields a $p < 2.2 \times 10^{-16}$ ***, therefore, we reject the null hypothesis and accept the alternative hypothesis "*the proportion of broken clients is different across each* `semver` *level of library upgrades*".

To assess the differences across `semver` levels, we conduct post-hoc analyses for each pair of groups using Fisher's exact test on the contingency tables. We adjust the resulting p-values using

Table 3.5: p-values and odds ratios across all pairs of `semver` levels in $\mathcal{D}_d^r$ to assess the differences in terms of broken clients

Semver level	p-value	Odds ratio
Major vs. minor	7.45×10^{-25} ***	0.64
Major vs. patch	3.02×10^{-83} ***	0.40
Major vs. dev	6.34×10^{-26} ***	1.52
Minor vs. patch	1.80×10^{-22} ***	0.62
Minor vs. dev	2.13×10^{-104} ***	2.38
Patch vs. dev	1.37×10^{-206} ***	3.82

a Holm-Bonferroni correction. Finally, we assess the effect size using the odds ratio. We obtain the results shown in Table 3.5.

The p-values are all significant considering a 0.01 threshold. If we look at the direction of the odds ratios, the results are as expected w.r.t. the differences among levels. The proportion of broken clients is higher for initial development upgrades, then major upgrades, then minor upgrades, and finally patch upgrades. Looking at the values of the odds ratios, we note that the difference in the odds of being broken depending on the `semver` level is perhaps not as high as one would expect. For instance, for major versus minor the odds of being broken in a minor upgrade is 0.6 times the odds of being broken in a major upgrade. An interesting finding is that, in the Maven ecosystem, initial development upgrades break a greater proportion of clients than major upgrades.

NUMBER OF DETECTIONS Figure 3.9 presents the distribution in logarithmic scale of the number of broken declarations (*i.e.*, detections) per client. Figures are presented for each `semver` sample in both $\mathcal{D}_d^o$ and $\mathcal{D}_d^r$. In these distributions we only consider broken clients, that is, clients that have at least one declaration affected by a BC. In both datasets $\mathcal{D}_d^o$ and $\mathcal{D}_d^r$, we observe a similar trend: major upgrades yield the highest number of broken declarations (medians of 5 and 6, respectively), followed by minor upgrades (medians of 4 and 3.5, respectively) and patch upgrades (medians of 3 and 3, respectively). In $\mathcal{D}_d^o$, initial devel-

Table 3.6: p-values and Cliff's delta across all pairs of `semver` levels in $\mathcal{D}_{d}^{r}$

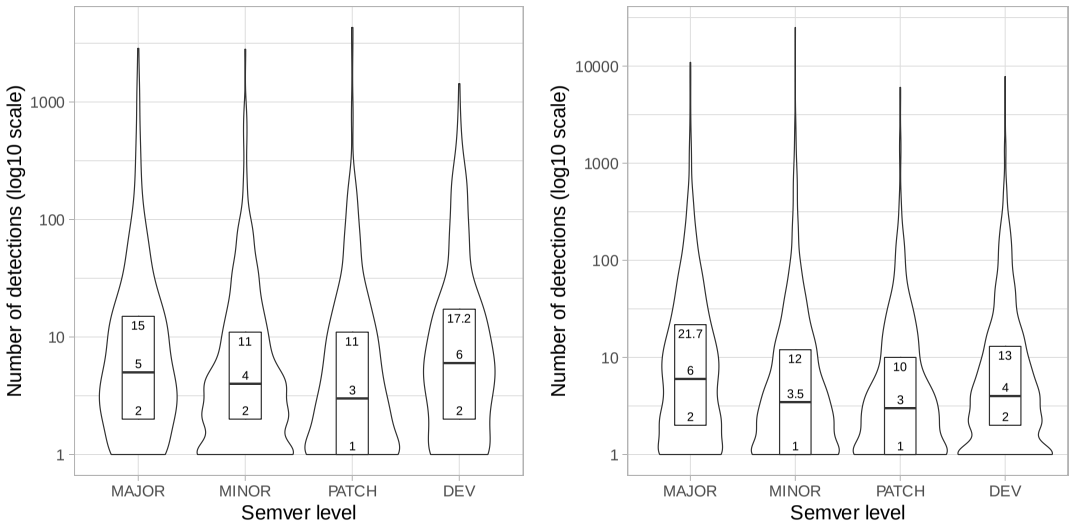
Semver level	p-value	Cliff's delta
Major vs minor	4.89×10^{-11} ***	0.16 (small)
Major vs patch	2.79×10^{-13} ***	0.20 (small)
Major vs dev	2.04×10^{-8} ***	0.12 (negligible)
Minor vs patch	0.165	0.04 (negligible)
Minor vs dev	0.165	−0.04 (negligible)
Patch vs dev	0.004 ***	−0.08 (negligible)

opment upgrades yield even more broken declarations than in major upgrades (median of 6), as opposed to what we observe in $\mathcal{D}_{d}^{r}$ (median of 4).

As we resort to random sampling to estimate the number of breaking declarations in broken clients, we use statistical inference to assess our raw results. For the sake of simplicity, we only perform the statistical analysis for the $\mathcal{D}_{d}^{r}$ dataset. We have the following null hypothesis: "*the number of broken declarations is the same across each* `semver` *level of library upgrades*". We run a Kruskal-Wallis rank-sum test on the number of broken declarations for each `semver` level. This test yields a $p = 3.82 \times 10^{-16}$ ***, therefore, we reject the null hypothesis and accept the alternative hypothesis "*the number of broken declarations of broken clients is different across each* `semver` *level of library upgrades*".

To make an in-depth assessment of the differences across `semver` levels, we conduct post-hoc analyses for each pair of groups, using a two-tailed Mann-Whitney test. We adjust the resulting p-value using a Holm-Bonferroni correction. In addition, we compute Cliff's delta to assess the effect size and report the interpretation of its value using Cohen's scale. We obtain the results shown in Table 3.6.

We note that two pairs are not significant (minor vs. patch and minor versus initial development), while the others are all significant at the 0.01 threshold. Looking at the direction of Cliff's deltas, the results are aligned with our expectations: the number of breaking declarations in major upgrades is greater



(a) $\mathcal{D}_{\mathrm{d}}^{\mathrm{o}}$ (b) $\mathcal{D}_{\mathrm{d}}^{\mathrm{r}}$

Figure 3.9: Number of detections per `semver` level

than in minor, patch, and initial development upgrades, and the number of breaking declarations in initial development upgrades is greater than in minor and patch upgrades. Looking at the values of Cliff's deltas, however, we note that the differences are very small across the groups. It indicates that, when a client is broken, the number of broken declarations it contains is similar whatever the `semver` level of the upgrade is.

BC TYPES Figure 3.10 shows the ratio of breaking and non-breaking uses of broken declarations for each BC type in $\mathcal{D}_{\mathrm{d}}^{\mathrm{r}}$. We note that most BCs result in breaking clients as soon as they use the broken declaration. Interestingly, we find several BCs that, in most cases, do not break clients even when the broken declaration is used in the client code. On the other hand, apart from the *interface removed* and *interface added* BCs, all other popular BCs (as computed in Section 3.5) are prone to break clients. However, it should be noted that, for most types of BCs, there is not enough data to support a definitive conclusion. This prevents us from proceeding to a reliable statistical analysis.

Analysis

Considering the results for the MDG corpus, we find that initial development and major releases tend to impact a higher number of clients (11.7% and 16.8%, respectively), as compared to minor and patch releases (7.8% and 5.0%, respectively). The same tendencies can be observed in the MDD corpus. Additionally, not only do clients break more often in major and initial development upgrades, but they also tend to break more. In general, clients are rarely impacted by breaking declarations in the libraries they use because they do not explicitly use the affected declaration. However, when a client uses a declaration that is affected by a BC, it is likely to break. These results are probably explained by the fact that library developers introduce BCs in parts of their API that are less likely to be used by their clients. This intuition has already been investigated in the literature [64], and our results are aligned with their observations.

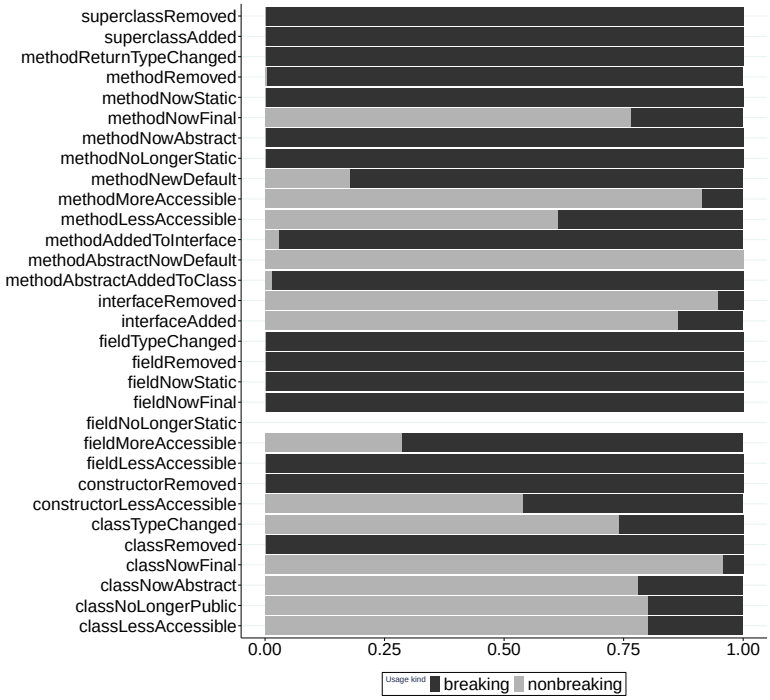


Figure 3.10: Ratio of breaking and non-breaking uses of API elements w.r.t. the BC type in $\mathcal{D}^{r}_{d}$

Q3: **What is the impact of BCs on clients?** $\mathbf{H_3}$ asserts that "*BCs have a significant impact in terms of compilation errors in client systems*." Conversely, we observe that in most cases breaking declarations are not used by client projects, which instead yields a low number of broken clients (7.9% for all releases). The number is even lower in the case of minor and patch upgrades. However, when a breaking declaration is used by a client, there is a high chance that it will be impacted. These results contrast with those of the original study and lead us to reject $\mathbf{H_3}$.

Threats to Validity

In this section, we discuss the main threats to the validity of our replication study, following the structure recommended by Wohlin et al. [159].

INTERNAL VALIDITY As explained in the Maracas section, the tool we implement and use to detect BCs and their impact on client code is not perfectly accurate, which impacts the metrics we compute. As a sanity check, we reuse and extend a benchmark for Java evolution and compatibility to evaluate the accuracy of our tool, using the reference implementation of the Java linker itself as ground truth. MARACAS obtains a precision of 96.3% and a recall of 98.5%, making the impact of this threat very low. Moreover, our tool is designed to be pessimistic and to over-approximate the impact of BCs in case of uncertainty. Therefore, the impact of this threat is to slightly over-estimate the number of broken clients. MARACAS does not reach 100% recall because of two false negatives: these are due to limitations of the underlying tool `japicmp` which is not able to detect BCs related to the **`strictfp`** and **`native`** modifiers. However, we do not expect that BCs related to these modifiers are common in practice.

We identify unstable API declarations of the libraries using a pre-defined list of naming conventions and annotations, which was extracted by semi-automatically analysing the top-1,000 most

popular libraries on MCR. However, this list is not exhaustive and cannot account for library-specific or organization-specific conventions. As a result, we have probably misidentified some API declarations as stable. Assuming that unstable declarations are more likely to break than stable ones, the impact of this threat is to over-estimate the number and impact of BCs.

Since it is not possible to reliably state whether a particular Maven artefact is a library or not, we consider that an artefact from MCR is a library if it has as least one *external* client. As a result, we potentially misidentify some artefacts as libraries. Assuming that libraries are more likely to be careful about BCs than other kinds of projects, the impact of this threat is to over-estimate the number and impact of BCs.

Our protocol excludes every library version suffixed with a qualifier (*e.g.*, `-beta1`, `-rc2`) as they are not final and are not meant to be used by the general public. This complies with the Maven principle stating that every qualified version is anterior to the corresponding non-qualified version. However, there are some libraries that always tag their versions with specific qualifiers. For instance, the developers of the popular Google Guava library tag every version released since 2017 with a `-jre` or `-android` suffix: there are no unqualified versions. Google Guava versions released after 2017 are thus excluded from our datasets—even though they are legitimate, while anterior versions are included. The impact of this threat is however low, as most of the qualifiers we found in our datasets correspond to pre-releases (*cf.* Derived Datasets section).

The corpora we use to extract our datasets (MDD and MDG) do not contain any information regarding version ranges and constraints. Concrete dependency versions have been resolved at the time the corpora were created, so a dependency with a version range (or no version at all) would be replaced with a concrete version picked by the Maven dependency resolver. As a result, even if some artefacts were using version ranges, we are not able to see and analyse them. We expect the impact of this threat to be low, as version ranges are not popular in the MCR [41].

EXTERNAL VALIDITY Our study targets Java libraries and clients inside the MCR ecosystem. Since the definition of BCs is specific to a particular programming language and since ecosystems have very different practices when it comes to BCs and versioning culture and habits [34], there is no particular reason to expect that our results generalize to the ecosystems of other programming languages or other Java ecosystems.

3.6 RELATED WORK

Prior research in the field of library evolution has focused on understanding *why* and *how* evolution happens [42]. Answering the *why* involves understanding the motives triggering the need to change a library and its API. In particular, researchers study the social factors motivating software change [12, 17, 18, 20, 166]. Conversely, to understand *how* library evolution occurs, researchers analyse the API evolution process and the evolving software itself. In this section, we discuss a set of studies that aim at understanding *how* software libraries evolve over time. We consider studies that analyse the evolution of software ecosystems as a whole; the nature of change in terms of backwards compatibility; and the impact that API evolution stirs up on client projects.

Ecosystems Evolution

Several studies aim at understanding the evolution of a software ecosystem on its own (*e.g.*, Eclipse, Apache). This is done to catch a glimpse of the evolution practices and expectations within the ecosystem community [17, 18]. As a direct consequence, researchers are able to create models and claims that support the development process within the studied ecosystem.

In the case of the Eclipse ecosystem, Businge, Serebrenik, and Brand [22] evaluate the applicability of Lehman's laws of software evolution to their corpus. In a later study, the same authors analyse to which extent client plug-ins rely on unstable and

internal Eclipse APIs [23]. They claim that more than 40% of Eclipse plug-ins depend on this type of libraries. Likewise, Wu et al. [164] study API changes and usages in both Eclipse and Apache ecosystems. The Apache ecosystem is also studied by Bavota et al. [12] and Raemaekers, Van Deursen, and Visser [132]. On the one hand, Bavota et al. [12] report on the evolution of dependencies among projects within the ecosystem. As the main conclusion, they discover that both the size of the ecosystem and the number of dependencies grow exponentially over time. On the other hand, Raemaekers, Van Deursen, and Visser [132] measure Apache APIs stability with a set of metrics based on method removal, modification, and addition.

The Squeak and Pharo ecosystems have also been a target of research. Robbes, Lungu, and Röthlisberger [136] study the ripple effects caused by method and class deprecation in these ecosystems. They state that 14% of the deprecated methods and 7% of the deprecated classes impact at least one client project. Hora et al. [66] complement Robbes, Lungu, and Röthlisberger [136] findings. They conclude that 61% of client projects are impacted by API changes, and more than half of those changes trigger a reaction in affected clients. In addition, Decan and Mens [34] perform an empirical study where they analyse the compliance to `semver` principles of projects hosted in four software packaging ecosystems (*i.e.*, Cargo, npm, Packagist, and RubyGems). They discover that certain ecosystems, such as RubyGems, do not adhere to `semver` principles when analysing dependency constraints.

The abovementioned studies give a good overview of the evolution of certain ecosystems. However, conclusions drawn by these studies do not hold outside the studied ecosystem [139]. Our study contributes to this body of knowledge by complementing the original results of Raemaekers, Van Deursen, and Visser [135] regarding the adherence to `semver` and the impact of BCs in the MCR ecosystem.

The growing interest in BCs is related to the need of analysing the stability of APIs and the impact these changes have on client projects. One of the main observations in the literature is that backwards incompatible changes are often introduced between two versions of an API. In fact, in a corpus of Java APIs, Dietrich, Jezek, and Brada [40] find that 75% of library upgrades introduce BCs between adjacent versions. This study was later enhanced by Jezek, Dietrich, and Brada [71] who argue that 80% of API releases are backwards incompatible. Mostafa, Rodriguez, and Wang [116] also report that 76.5% of the releases they analyse introduce behavioural incompatible changes. Nevertheless, there is still disagreement regarding these figures. For instance, Xavier et al. [165] claim that only 14.78% of the changes in their dataset are backwards incompatible, while Brito et al. [20] state that 39% of the introduced changes are classified as breaking. These differences are due to the diversity of libraries that are analysed and their characteristics, and the criteria used to select them (for instance, the most popular Java libraries hosted on GitHub). In this study, we detail a protocol that enables us to give a clear overview of the state of BCs in MCR over the past 13 years.

In addition, Raemaekers, Van Deursen, and Visser [134, 135] conduct a study that relates `semver` with backwards incompatibility. The authors discover that `semver` is not strictly followed in practice. That is, BCs are also introduced in minor and patch releases [71, 134]. They also claim that minor releases introduce more changes than major releases [135]. Similarly, some studies relate the nature of the API with the tendency to introduce BCs. Xavier et al. [165] find that APIs with a higher frequency of BCs introduction tend to be more popular, larger, and active. They also argue that the frequency of BCs increases over time. Raemaekers, Van Deursen, and Visser [135] partially confirm this claim: larger libraries tend to introduce more BCs. However, they also conclude that more mature APIs do not introduce more BCs, which seems counter-intuitive when contrasting the results against Xavier et al. [165] study. Decan and Mens [34] study adherence to `semver`

principles at the dependency constraints level. They find out that newer ecosystems tend to follow `semver` guidelines, and that `semver` practices have become more popular as time passes. Our study complements these results by studying the adherence to `semver` in MCR.

It is also important to be aware of the type of changes that are usually introduced during API evolution. Cossette and Walker [28] analyse a set of binary BCs based on the affected entity type (*i.e.*, class, method, field) and its visibility (*i.e.*, protected, public). In more recent work, Wu et al. [164] undergo a study that analyses 23 types of changes related to API types and methods [39, 57]. They find that missing classes and methods are important types of BCs affecting client projects. Ketkar, Tsantalis, and Dig [81] study type changes and required code adaptations. They find out that type changes are more common than renamings, and that they usually appear on public entities. Furthermore, a particular kind of change has drawn much attention from the community: API deprecation [20, 134, 136, 139]. While deprecating an API entity does not immediately break client code, it signals that BCs may be coming in the future—as the semantics of the annotation suggests. However, Raemaekers, Van Deursen, and Visser [134, 135] notice that API developers tag deprecated API entities without ever removing them from their API. In other cases, they do quite the opposite: API developers remove declarations from the API without deprecating them first. Brito et al. [20] point out that this is needed to reduce the required maintenance effort by API developers.

In spite of the contributions and findings of the abovementioned studies, there is still a long way to go. First, some studies do not define a clear scope of the applicability of their conclusions. In essence, it is not clear if findings account for source, binary, or behavioural incompatibility. Moreover, how to detect and classify behavioural incompatibilities is still an open problem. Second, the selection and study of the subset of BCs seems arbitrary, incomplete, and in some cases incorrect. For instance, some studies concluding on backwards compatibility claim that adding a method to a class is not a BC. Although this change

is indeed binary compatible, it will break source compatibility when the method is added to an abstract class that is extended by client code. Third, there is a lack of consensus in research findings across studies. This is the case when reporting on the percentage of incompatible API releases and the correlation between API properties and BCs frequency. This might be related to the underlying datasets: some studies analyse only popular projects [20, 116, 165], and others consider few libraries [88, 138].

Refactorings

Refactorings are changes aimed at improving the structure of a project without changing its observable behaviour [50]. One of the main inquiries concerning refactorings in API evolution is understanding to what extent API changes are actual refactorings. Dig and Johnson [42] discover that between 3% and 27% of changes on two common Java APIs are refactorings. Furthermore, they find that at least 81% of BCs in four Java APIs are due to refactorings. However, in more recent studies this number might be lower. For instance, Brito et al. [20] show that 47% confirmed BCs in their corpus are actual refactorings, and these are the most common types of BCs. Additionally, Kula et al. [88] state that refactorings break less than 37% of all clients of a given API. They find a tendency to find more BCs and refactorings in API internal entities [20, 88].

The main limitation of these studies is the level of abstraction at which the analysis is performed. That is, a refactoring might be composed of multiple BCs and, in some cases, by multiple refactorings. For instance, method renamed could be recorded in a delta as both a method removed and method added change. Analysing these changes requires an additional effort in dissecting compound cases.

More recent studies attempt to understand how client projects are impacted by API evolution. In some of them [12, 40, 136, 165] we find the same claim: there is no massive impact of API changes on client code. In fact, Bavota et al. [12] state that only around 5% of the projects in their corpus are impacted by API evolution; Xavier et al. [165] discover that only 2.54% client projects in the dataset are impacted by API BCs; and Robbes, Lungu, and Röthlisberger [136] show that 14% and 7% of class and method deprecations, respectively, impact client projects. From a different perspective, Kula et al. [88] state that BCs are more likely to appear in API entities that are not used by client code. Later, Raemaekers, Van Deursen, and Visser [135] relate the number of compilation errors with the introduced BCs. They do so by individually inserting BCs in the API source code.

Regarding API-client co-evolution, there is a growing interest in understanding *why*, *when*, and *how* client projects upgrade to a newer version of an API. On the quest of answering the *why* and *when*, Raemaekers, Van Deursen, and Visser [134] show that API upgrading tends to be performed when major API updates are released. Bavota et al. [12] indirectly confirm this claim by stating that client projects upgrade to a newer version of an API only when substantial changes are introduced. Regarding the *how*, Bavota et al. [12] highlight that even though few client projects might be affected by API evolution, certain dependencies that offer cross-cutting services can strongly impact them. To support these first insights, Robbes, Lungu, and Röthlisberger [136] find that resolving the first 25% ripple effects in the Squeak and Pharo ecosystem, requires more than 14 developers. In addition, several commits are registered to resolve the issue, which suggests the existence of non-trivial changes. Both Robbes, Lungu, and Röthlisberger [136] and Sawant, Robbes, and Bacchelli [139] claim that finding systematic changes in affected client code is rare. There are many cases where impacted code is simply dropped, or an ad-hoc solution is provided. These findings are contrary to what developers postulate in Brito et al. [20] study: they argue that

API migration results in minor and easy changes. In addition, Mostafa, Rodriguez, and Wang [116] claim that 67% of bugs introduced by behavioural changes can be fixed by simple changes (*e.g.*, replacing arguments, converting return values).

Despite the new contributions in the field, few papers study both how APIs evolve and how this evolution impacts client projects. Moreover, when they analyse API usage there is a misalignment between the API change and usage types. For instance, Wu et al. [164] label *inheritance for IoC* as an API usage type. However, this type of usage can be split into other categories, such as *method overriding*, *class extension*, and *interface implementation*. With this differentiation it is possible to relate API changes at different levels (*i.e.*, class, method, and field levels) with atomic API usages; and accurately point to affected client members. Finally, as in the case of studies related to backwards compatibility, we perceive contradicting results that are most likely due to differences in the studied datasets.

3.7 DISCUSSION

In this section, we discuss the main implications of our findings for library developers, library clients, and researchers.

IMPLICATIONS FOR LIBRARY DEVELOPERS The introduction of BCs is inherent to software evolution and cannot always be avoided. Although libraries are encouraged to preserve backwards compatibility, the need to introduce new features and improve the quality of the library sometimes results in incompatible changes. We claim that introducing BCs is tolerable as long as they are properly announced in advance to not take clients by surprise. Versioning conventions or code-level mechanisms (*e.g.*, annotations, naming conventions) are regularly used for this purpose. In addition, many software ecosystems such as Maven, GitHub, and npm encourage their users to adhere to the `semver` policies. They even offer core tooling to support developers on this quest for quality and compliance. The semantics of such

policies might differ among ecosystems, thus, library developers are exhorted to carefully understand and follow such rules [18].

Introducing too many BCs may, however, hurt the reputation of a library. It is very hard for developers to manually detect BCs in source code, so we believe that the use of tools such as `japicmp` and Maracas is important to help library maintainers make the right decisions. The Apache Commons developers, for instance, use `japicmp` on every new release of their libraries to check for backwards compatibility.[17] These tools are also able to automatically generate reports that help clients to anticipate the changes.[18] The first step towards a more disciplined evolution of libraries is to detect and communicate on BCs, two aspects addressed by these tools. As we have shown in our analysis of **Q3**, some BCs appear to be more critical than others for client developers. We thus encourage library developers to interpret the output of these tools wisely and to account for the severity of different changes.

The ability of Maracas to infer the impact of BCs on client code is also beneficial for library developers. Developers may run Maracas as part of a continuous integration pipeline to check that a particular commit, pull request, or release does not significantly impact their clients, and reconsider the changes if the impact is too high. This type of tooling can also yield valuable information on how unstable or unsafe parts of APIs are being used by client projects. With this information, library developers can either decide to promote internal interfaces to public ones, including new features in their list of public interfaces [67]. They can also analyse to what degree unsafe declarations are impacting client projects, and based on these results, they can even come up with new designs to transform an unsafe interface into a safe one [108].

IMPLICATIONS FOR CLIENT DEVELOPERS The main implication of our results for client developers is that the situation is not

17 https://garygregory.wordpress.com/2020/06/14/how-we-handle-binary-compatibility-at-apache-commons/

18 `https://commons.apache.org/proper/commons-lang/japicmp/`

as bad as reported in the literature. In MCR, most releases comply with `semver` requirements and avoid BCs in non-major releases. Besides, as we have shown in **Q2**, the situation has significantly improved over time.

Each ecosystem has its own policy regarding versioning conventions and the treatment of BCs [18]. Cargo, npm, Packagist, and Rubygems, for instance, do not apply semantic versioning in the same way [34]. Client developers should thus pay attention to ecosystem-specific guidelines and pick an ecosystem that advocates a strict policy to minimize the risk of being impacted by unwanted changes. Additionally, identifying unstable declarations in used APIs via `semver` or other code-level mechanisms is important to avoid broken client code after upgrading to more recent releases. Naming conventions [21] and use of annotations are some of the signaling mechanisms that client developers should look for.

Tools such as Maracas should also be beneficial to client developers. When faced with the possibility of upgrading a dependency, developers may employ Maracas to evaluate the impact of different versions, and choose the one that addresses their requirements without causing too much disruption.

IMPLICATIONS FOR RESEARCHERS As we have seen, the use of code-level mechanisms to delimit unstable APIs relies on conventions that might vary from one organization to the other or from one library to the other (`@Beta` for Google, `@Internal` for Apache, `sun.*` packages in the JDK, *etc.*,). Contrary to semantic versioning, there is no standardization of these mechanisms, which makes it hard for clients to understand which declarations should be considered stable or unstable. Besides, it is not clear how `semver` and code-level mechanisms interact: the `semver` specification only mentions the `@Deprecated` annotation. Should developers release a major revision when they introduce a BC in a beta-stage API? We believe that clarifying the role of code-level mechanisms and their relation with `semver` would be beneficial. Another interesting line of work would be to incorporate better mechanisms to delimit APIs directly in programming

languages: developers are currently forced to make some declarations public only for technical reasons (*cf.* Section 3.2) even though they are not part of the intended API.

Researchers should also strive to design and implement benchmarks to compare tools related to library evolution objectively. The benchmark of Jezek and Dietrich [70], which we reuse and extend to evaluate the accuracy of Maracas, is the first step in this direction and should be complemented with other benchmarks, for instance, related to behavioural compatibility.

Furthermore, when analysing software evolution, the design of a study protocol and the creation of the underlying datasets should be carefully performed. Sampling bias is a recurrent threat to validity that can hurt the interpretation of API evolution studies. For instance, selecting only the most popular libraries on a repository, or only the ones related to a particular ecosystem hurts the generalizability of the study findings. To cope with this issue, representative and diverse samples are required to come up with relevant conclusions [119].

Finally, because BCs are not always avoidable, researchers should continue to develop tools and methods that assist client developers in automatically migrating their code [28, 167, 169].

3.8 Conclusion

In this chapter, we conduct an external and differentiated replication study of the work presented by Raemaekers, Van Deursen, and Visser [135]. The motivation behind this study is to better understand which kind of BCs happen in libraries hosted on the MCR, and what is their impact. We rely on `semver` principles to draw conclusions that are aligned with versioning conventions that signal API instability. Our protocol addresses some limitations of the original study and expands the analysis to a new dataset spanning seven more years of the MCR. We implement and use Maracas to compute BCs between adjacent versions of libraries, and to detect locations in client code that are affected by such BCs.

The main results of the study are as follows:

Q1: **How are semantic versioning principles applied in the Maven Central Repository?** 83.4% of all upgrades on MCR do comply with `semver` principles. Still, 20.1% of non-major releases are breaking, threatening client projects.

Q2: **To what extent has the adherence to semantic versioning principles increased over time?** The tendency to comply with `semver` practices has significantly increased over time: the number of non-major breaking releases has decreased from 67.7% in 2005 to 16.0% in 2018.

Q3: **What is the impact of BCs on clients?** Only 7.9% of the clients we analyse are impacted by the BCs introduced in adjacent library releases. However, when breaking declarations are used by client projects, they are likely to break.

According to these results, we state that libraries and client projects on the Maven ecosystem are *not* "breaking bad". To be precise, developers of Maven projects tend to follow `semver` principles and are for the most part disciplined when introducing BCs. While the situation has improved over time, there is still room for improvement. Although the impact of BCs on client projects is low, more research is needed to support clients that are impacted and need to migrate their code. Differences with results reported in the original study are explained by major changes introduced in the protocol and the extended time span of the new corpus.

As future work, we first would like to perform qualitative analyses to complement our findings. In particular, we would like to explain the phenomenon we observed: what are the motivations behind inserting BCs in non-major releases, and why has the adherence to semantic versioning increased so significantly. These questions could be answered by interviewing library maintainers and clients. Second, we would like to study how the evolution of the Java language itself impacts the definition of BCs, and how this affects libraries and clients. As new constructs are made available in Java (*e.g.*, the `default` operator in Java 8 or the `record` data type in Java 15), new BCs appear. At the same time,

these new constructs provide new strategies to deal with certain BCs (*e.g.*, `default` methods allow to gracefully evolve an interface without forcing changes in existing implementations). Third, we believe that the understanding of how client projects react to BCs is another step towards finding a way to support library-client co-evolution. Thus, we aspire to study how clients react in the wild and which patterns can be identified from these reactions. Finally, we also would like to study behavioral incompatible changes in Java libraries.

Part III

THE VERBAL VIEW

On the *means* of the library-client co-evolution phenomenon: assisting library evolution

4

MARACAS: DESIGNING AND IMPLEMENTING THE STATIC IMPACT ANALYSIS APPROACH

ABSTRACT *Software projects seldom live in isolation: they offer services to other projects when acting as libraries, and consume services from other projects when acting as clients. These projects are undergoing constant change, which might result in the introduction of breaking changes. Several studies have focused on their detection, however, breaking changes are not enough to identify the actual broken code a library change can cause on clients. Thus, library developers might hesitate to introduce a change that decreases its technical debt for fear of breaking client code and losing advocates, or; on the contrary, an unnoticed breaking change can slip into the next library release of the library hampering both library and client projects. The lack of language-agnostic, accurate, and resource-efficient approaches that inform about Application Programming Interface (API) evolution and impact on client projects hinders software evolution as a whole. In this chapter, we introduce the requirements and design of the static impact analysis approach that aims at (i) providing a language-agnostic approach that can be implemented in any language to study API evolution and impact, and; (ii) efficiently and accurately identifying client code that is broken due to the introduction of a breaking change. The approach is implemented in* Maracas*, a static analysis tool for studying API evolution and impact on Java projects, which was previously introduced in Chapter 3. The tool has been tested by means of providing unit tests and an accuracy evaluation based on the synthetic benchmarks* `comp-changes` *and* API evolution data. *The tool reports a precision of 0.96 and 0.92 and a recall of 0.95 and 0.99 for these two benchmarks, respectively. Although some aspects about the tool have already been*

exposed in other chapters within this manuscript, this chapter expands on previous descriptions and centralizes the technicalities surrounding our approach and its implementation.

4.1 INTRODUCTION

Software developers favour reuse to focus on the development of features that add value to their projects. Designing software for reuse requires, among different aspects, complying with the long-standing principle of information hiding [126]. According to this principle, implementation details should be hidden for external use, and only the specification of offered functionality should be exposed to the public. This specification is known as an Application Programming Interface (API).

As time passes, the requirements and environment of software projects change. Thus, code undergoes an evolutionary process where both the implementation and offered API change. A changing API might pose a threat to client projects, especially if such changes are made in a backwards-incompatible manner. *Backwards compatibility* is a software property that can be achieved when no client project depending on the changing API is broken.

When backwards incompatibility manifests, changes that can break client code are said to be Breaking Changes (BCs). BCs are a proxy to foresee the *potential* impact—in the form of broken code—of the API evolution on clients. However, they do not inform about the *actual* broken code on clients of the library. The lack of tool support and information about BCs introduction and impact on client code, lead to decisions on whether to introduce BCs end up being made without enough evidence. As stated by Darcy, "[t]he basic challenge of compatibility is judging if the benefits of a change outweigh the possible negative consequences (if any) to existing software and systems impacted by the change." [32] For instance, the identification of a BC introduction in a release to maintain or add value to the library, might be reverted due to the fear of impacting too many clients. Moreover, a client might not upgrade to a newer backwards-incompatible release to avoid bearing the costs of fixing broken code. The lack of impact

information then hinders change, both on the library and client sides [91].

Several studies in the field of software evolution have focused on the detection and study of BCs [70]. From these studies, a vast majority have opted to investigate the phenomenon in Java ecosystems [91]. This has hampered the development of a language-agnostic theory about software evolution. Moreover, when studying impact analysis, current research focuses on conducting empirical studies to extract insights from the crowd [91]. To the best of our knowledge, a handful of studies have opted to develop static analysis approaches to detect such impact, however, they are either resource-expensive, often requiring manual labour or building the software project [136], or; too coarse compromising the accuracy of the developed tools [165].

This chapter aims at (i) identifying use cases and requirements coming from library and client developers when facing API evolution; (ii) proposing the static impact analysis approach to statically detect BCs and broken uses of modified API members in client code. The approach aims at being language-agnostic, resource-efficient, and accurate, and; (iii) presenting the implementation of our approach for Java projects, Maracas, which has been tested in terms of functionality and accuracy. Evaluating if the approach is indeed language-agnostic and efficient is yet to be investigated in future research.

The remainder of the chapter is structured as follows. In Section 4.2, we introduce core concepts related to backwards compatibility and a motivating example to illustrate such concepts. Section 4.3 briefly introduces the perspective of library and client developers, together with a list of functional and extra-functional requirements to consider when performing API evolution analysis. This section will motivate the introduction of our static impact analysis approach in Section 4.4. Then, in Section 4.5, we present Maracas, the implementation of the static impact analysis approach. In Section 4.6, we introduce the current state of the art regarding the detection of BCs (*aka.*, evolution analysis) and their impact on client code (*aka.*, impact analysis). We draw our conclusions in Section 4.7.

In this section, we present the core concepts related to backwards compatibility, which frame the requirements, design, and architecture of our static impact analysis approach. In particular, we define what an *application programming interface* is as a base to later understand *backwards compatibility*. We then dive into how backwards compatibility manifests in *Java* and its different flavours in this programming language.

Application Programming Interface

An Application Programming Interface (API) is a software interface that specifies and exposes functionality offered by a library, framework, or Web service to client projects [91, 126]. The API is a contract between two parties that specifies the communication between them and the type of inputs and outputs expected in each specific interaction. It is also a way to comply with the information-hiding principle—that is, hiding implementation details [126]. In the context of this thesis, we only study APIs exposed by libraries or frameworks—we do not consider APIs offered by Web services.

The API of a project can be shaped by means of leveraging specific constructs of the host programming language. For instance, Java offers a set of access modifiers (*i.e.*, **`public`**, **`protected`**, and **`private`**) that determine which parts of the code can be accessed by client classes. In particular, **`public`** members can be accessed by any client class; **`protected`** members can be accessed only by subclasses of the class owning the member or by classes declared in the same package as the owning class, and; **`private`** members can only be accessed in the body of the top-level class owning such members [57]. The language also offers annotations such as `@Deprecated`, which signals the parts of the code that will be removed in the future, and hence, should not be used by client projects anymore [57]. As a final example, the JSR 376 introduced the Java Platform Module System (JPMS), which brings in the **`module`** language construct. Modules aggregate packages and

define the set of offered and consumed services, where a service must be understood as a type—usually an interface—that provides a set of methods [100].

EXAMPLE JUnit 4[1] is a Java testing framework originally written by Erich Gamma and Kent Beck. The framework is supported by the main Java IDEs (*e.g.*, IntelliJ, Eclipse, Visual Studio Code), and it is popular among Java developers (*e.g.*, Concordion[2], Eclipse Tycho[3], Grails[4], Groovy[5]).

Software Deployment & Evolution

Software undergoes continuous and progressive change over time to cope with changes in its requirements and environment [97]. Changes are typically introduced to patch security issues, add new features, refactor the code, fix bugs, and introduce other enhancements [40, 87]. These changes lead to *software evolution*, which is the *phenomenon* resulting from executing a set of processes directed to modify a software project [80]. To manage this evolution library developers pack the project code into releases. A library *release* is a versioned distribution of a snapshot of a software project at a given moment. Client developers can set dependencies on a specific release of a library, ensuring the use of a set of stable features.

EXAMPLE JUnit 4 has several releases, each one with a set of clients that depend on it. Concordion is one such client. The latter is an open-source library that automates specification by example—that is, the documentation used to specify a feature is used as input to generate code that tests it. Figure 4.1[6] shows a co-evolution scenario where some changes are introduced to JUnit 4

1 https://junit.org/junit4/
2 https://concordion.org/
3 https://projects.eclipse.org/projects/technology.tycho/
4 https://grails.org/
5 https://groovy-lang.org/
6 Additional versioning information (*i.e.*, qualifier identifier coming after the third version number) is taken out from the image for readability purposes.

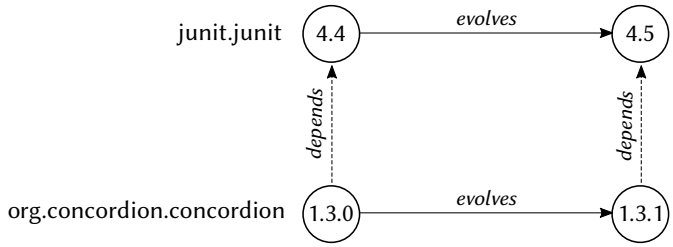


Figure 4.1: JUnit 4 and Concordion co-evolution

release 4.4 resulting in a new release labeled with version number 4.5. Concordion release 1.3.0 depending on JUnit 4.4 is modified to leverage the changes introduced in JUnit 4.5, resulting in a new release 1.3.1.

Backwards Compatibility

When software evolution manifests, any change introduced in a library can potentially break client code [32]. A change is then said to be a Breaking Change (BC) if it potentially breaks client code depending on the library at hand [32]; otherwise, it is said to be a non-breaking change.

If we consider an API to be a language that has a well-defined syntax and semantics, we can also classify BCs as either syntactic or semantic. On the one hand, *syntactic* BCs impact the form of the API. For example, when a method is renamed or its parameter list is altered. On the other hand, *semantic* BCs impact the meaning of API members. Other researchers also refer to them as *behavioural BCs* [26, 30, 116]. However, we stick to the term *semantic compatibility* given that any syntactic BC also impacts the behaviour of a program. Examples of semantic BCs include getting different outputs for two versions of the same method after providing the same inputs, or when side effects are added or removed from a method.

The definition of a BC is language-specific; it depends on how the language and related tools (*e.g.*, compiler, linker) are implemented, and, consequently, when and where to expect errors

in its programs. To illustrate, consider the changes in access modifiers in Java. Decreasing the visibility of an API member (*e.g.*, going from **`public`** to **`protected`** or **`protected`** to **`private`**) is considered a BC. Nevertheless, in Python, this BC cannot exist by design: access modifiers are not part of the constructs of the language.

Backwards compatibility is an attribute of a library release that guarantees that no BC has been introduced in the targeted release with respect to the previous one. In other words, safe substitution is guaranteed in the contracts established between the library and its clients [70]. As stated by Jezek, Dietrich, and Brada and based on the Liskov Substitution Principle (LSP):

> "[S]ubstitution is safe if the preconditions of invoked services are not strengthened, and the postconditions are not weakened." [71]

When studying backwards compatibility, this means that the preconditions of the API contracts should not have stricter validation rules, otherwise, an error might be raised when getting more general inputs, and; the postconditions of the contracts cannot be less restrictive, otherwise an error can be raised when returning a more general output.

Consider the sequence L of relevant releases of a given library, and the library releases l_i and l_{i+1} for $i \in \mathbb{N}$ and $l_i, l_{i+1} \in L$. Consider also the universe of clients C depending on a library release in L. l_{i+1} is said to be an *adjacent release* of l_i if it comes immediately after the latter in L. A set of changes Δ are introduced in l_{i+1} (*i.e.*, $l_i \xrightarrow{\Delta} l_{i+1}$). A subset of these changes Δ' (*i.e.*, $\Delta' \subset \Delta$) are BCs that potentially impact a client project c for $c \in C$. Thus, l_{i+1} is said to be backwards compatible if $\Delta' = \emptyset$, and backwards incompatible, otherwise. These definitions are to be used in the current chapter henceforth.

EXAMPLE Bringing back our motivating example, a set of changes Δ were introduced between JUnit 4.4 and 4.5. A subset of those changes Δ' resulted in a set of modifications that can potentially impact clients like Concordion. Listing 4.1 shows

Listing 4.1: Example of a method removal BC introduced in JUnit 4.5. The BC impacts Concordion 1.3.0 code

```
1  // JUnit diff between 4.4 and 4.5
2  public class RunNotifier {
3      ...
4  -    public void testAborted(Description description, Throwable
       cause) {...}
5  }
6
7  // Concordion 1.3.0 depending on JUnit 4.4
8  public class ConcordionRunner extends JUnit4ClassRunner {
9      private Description fixtureDescription;
10     ...
11     protected void runMethods(final RunNotifier notifier) {
12         Description description = fixtureDescription;
13         ...
14         try {...}
15         catch (InvocationTargetException e) {
16             notifier.testAborted(description, e.getCause());
17             ...
18         } catch (Exception e) {
19             notifier.testAborted(description, e);
20             ...
21         }
22         ...
23     }
24 }
```

an example of a syntactic BC introduced in JUnit 4.5. From line 1 to 5, we present the textual diff between the two releases of JUnit 4, while the rest presents Concordion code using the modified API member. In this example, JUnit 4 developers decided to remove the method `testAborted(Description, Throwable)` from the `RunNotifier` class (lines 3 and 4). This change impacts the code in Concordion 1.3.0. In particular, the `runMethods(RunNotifier)` method in the `ConcordionRunner` class is impacted twice (lines 16 and 19) when the client project invokes the `testAborted(Description, Throwable)` method via the `notifier` object of type `RunNotifier`.

In Java, syntactic backwards compatibility manifests in two flavours: source or binary compatibility [32]. Hereafter, we describe these two compatibility kinds and illustrate them with examples taken from the literature [70].

SOURCE COMPATIBILITY Source compatibility is preserved in l_{i+1}, if every client c depending on l_i *compiles* without error against l_{i+1}. An example of source-only incompatible changes includes, among others, modifications to type parameters [71]. These modifications are only considered by the compiler; this information is removed from the bytecode as a result of performing type erasure.

Listing 4.2 shows an example of a source-only BC based on an example presented by Jezek, Dietrich, and Brada. From line 1 to 5, we present the textual diff between the two releases l_i and l_{i+1}, while the rest presents client code using the modified API member. Consider the class `Library` defined in l_i (line 2) and a method `foo` that returns a list of integers (*i.e.*, `List<Integer>`) defined in such a class (line 3). The return type of the method and its implementation is later modified in l_{i+1}, so it now returns a list of strings (*i.e.*, `List<String>`) (line 4). A client project that originally depends on l_i defines the class `Client` (line 8). Within the `main` method of the class, a `Library` object is created (line 10) and then the `foo` method is invoked (line 11). The result of the invocation is saved in the `listOfIntegers` variable of type `List<Integer>` (line 11). However, when the client upgrades its dependency to l_{i+1}, a compilation error will be raised when the compiler checks the type parameters of the variable. The same change is not considered a binary-incompatible change due to type erasure. Type erasure was introduced in 2004, in Java Development Kit (JDK) 1.5, as a means to preserve backwards compatibility with previous versions of the JDK. It removes type parameters from the code when generating Java bytecode. This means that changes to classes that are used as type parameters

Listing 4.2: Example of a source-only BC adapted from Jezek, Dietrich, and Brada [71]

```
1  // Library diff between release i and i + 1
2  public class Library {
3  -    public List<Integer> foo() {...}
4  +    public List<String> foo() {...}
5  }
6
7  // Client using the library release i
8  public class Client {
9      public static void main(String[] args) {
10         Library library = new Library();
11         List<Integer> listOfIntegers = library.foo();
12     }
13 }
```

might impact source compatibility, but go completely unnoticed when checking binary compatibility.

BINARY COMPATIBILITY Binary compatibility is preserved if every client c depending on l_i *links* without error against l_{i+1} [57]. *Linking* is the process of taking the previously loaded binary form of a type and combining it with the JVM run-time state for future execution [57]. The Java programming language enforces binary compatibility as one of its main policies. It also encourages library developers to inform client developers about the impact new releases of the library can have on client binaries that cannot be recompiled [57]. This is of particular importance in ecosystems such as Open Service Gateway Initiative (OSGi) and Maven, where bundle wiring and transitive dependencies resolution can result in the use of a release different from the one used during compilation [70]. Furthermore, the Java Language Specification (JLS) dedicates a whole chapter (*cf. Chapter 13 - Binary Compatibility*) to describe binary compatibility in the Java language. The chapter enumerates a subset of known binary BCs that can impact client code [57].

Listing 4.3: Example of a binary-only BC adapted from Jezek, Dietrich, and Brada [71]

```
1  // Library diff between release i and i + 1
2  public class Library {
3  -    public Collection foo() {...}
4  +    public List foo() {...}
5  }
6
7  // Client using the library release i
8  public class Client {
9      public static void main(String[] args) {
10         Library library = new Library();
11         Collection collection = library.foo();
12     }
13 }
```

Listing 4.3 presents an example of a binary-only BC based on an example presented by Jezek, Dietrich, and Brada. From line 1 to 5, we present the textual diff between the two releases l_i and l_{i+1}, while the rest presents client code using the modified API member. Consider the class `Library` defined in l_i (line 2) and a method `foo` that returns a collection of values (*i.e.*, `Collection`) defined in such a class (line 3). The return type of the method and its implementation is later modified in l_{i+1}, so it now returns a list of values (*i.e.*, `List`) (line 4). A client project that originally depends on l_i defines the class `Client` (line 8). Within the `main` method of the class (line 9), a `Library` object is created (line 10) and then the `foo` method is invoked (line 11). The result of the invocation is saved in the `collection` variable of type `Collection` (line 11). However, when the client upgrades its dependency to l_{i+1} without recompiling its source code against the new dependency, a linkage error will be raised when the JVM linker checks the descriptor of the method. A method descriptor in bytecode has the format `methodName()returnType`, which in the case of the `foo` method in l_i will look like `foo()Ljava/util/Collection;`, while in l_{i+1} will look like `foo()Ljava/util/List;`. By simple string comparison these two descriptors are not interchangeable to the

eyes of the linker when performing the method resolution [100]. The same change will not be considered a source incompatible change given that the compiler does consider information about inheritance.

4.3 API CHANGE & IMPACT REQUIREMENTS

In this section, we introduce the main actors involved in the API evolution phenomenon and their perspectives when the phenomenon manifests. We wrap up the section by introducing the *use cases* and *general requirements* to provide an approach and tools that support the identified actors when facing API evolution.

Actors & Perspective

The API evolution phenomenon directly affects two kinds of actors, namely library developers and client developers. *Library developers* (*e.g.*, JUnit 4 developers) are the actors deciding whether to introduce changes in the library API. Once changes are introduced in the API, client developers are faced with a dilemma: *to upgrade or not to upgrade? Client developers* (*e.g.*, Concordion developers) are in charge of deciding how detrimental or convenient the upgrade is for their project [41]. Moreover, they need to assess what the consequences are in terms of resources investment when supporting any decision.

In general, when facing API evolution, library developers can follow one of two approaches to introduce required changes, namely:

TECHNICAL DEBT-AVERSE APPROACH Library developers prioritize a robust and clean design for their project. Ideal changes can come along in a backwards-compatible or backwards-incompatible manner. When the ideal change can be provided in a backwards-compatible fashion, no client project is impacted and no technical debt—associated with the evolution of the library—is generated. However, when the ideal change is introduced in a backwards-

incompatible fashion, there is a risk of breaking client code. Client projects can, then, (i) opt for upgrading to the new library release and the upgrade costs must be spread between the library and client projects; (ii) do not upgrade, increasing their technical lag as well as their technical debt [150], or; (iii) drop the use of the changing library as they assess too costly the resources required to upgrade to the new release. Costs of moving to other libraries are then to be considered.

BC-AVERSE APPROACH Library developers prioritize offering backwards-compatible changes that reduce the risk of impacting client code or losing clients. If the ideal change can be implemented without introducing BCs, no risk manifests. However, if the ideal change cannot be implemented in a backwards-compatible manner, library developers provide a workaround solution that can negatively impact their design and increase their technical debt. For instance, library developers can introduce an inefficient solution or duplicate code. Client projects can upgrade with the certainty that there will be no broken code, but even in this case, the technical debt borne by the library propagates to all its clients.

Following one of the two approaches requires having enough information about the evolution of the library and how this one impacts client code [16]. In the case of library developers, this information is relevant to know whether they are introducing BCs that are impacting client code. Based on this information, library developers can decide to keep the change or revert it. If they keep it and if the change is impacting client code, they can plan coping strategies to deal with the change [16]. In the case of client developers, they need to know if there are BCs in the new release they want to upgrade to, and if so, to what extent these changes impact their code. Based on this information, client developers can decide to not upgrade and bear the technical lag and debt inherent to these changes , upgrade to the new release

and modify their own code accordingly [150], or drop the use of the library and possibly migrate to other one.

Use Cases & General Requirements

As a general remark, library and client developers must be provided with enough evidence to face API evolution while prioritizing their values [18]. Without this information they need to make decisions based on assumptions and conventions (*e.g.*, semantic versioning) that provide incomplete, and, sometimes even, incorrect information. For instance, in Chapter 3, we have shown that 16.6% of projects in Maven Central Repository (MCR) do not increase the major number or use the initial development version to signal the introduction of BCs in their releases version numbers. They neither include these changes in the unstable part of their APIs, usually identified by the use of annotations or naming conventions. This behaviour might be related to a lack of knowledge about the nature of the introduced changes on the library side, and lead to misleading assumptions on the client side.

Furthermore, information about how BCs impact client code is also required. From the library perspective, developers might hesitate to include certain BCs in their new releases to decrease the risks of impacting client code [18, 35]. However, in MCR, there is evidence that only 7.9% of clients—within a sample of 15,701 projects in MCR—using a backwards-incompatible library release are impacted by BCs (*cf.* Chapter 3). Additionally, client developers need to foresee to what extent upgrading to a newer library release impacts their code, without waiting until performing the actual upgrade. Based on this information they can opt to bear the upgrade costs, incur in technical debt via technical lag, or drop the use of a library that to their eyes is problematic. Stringer et al. [150] have shown that for a set of 14 studied ecosystems, there is a percentage of dependencies that lag. This percentage varies depending on the values and policies of each ecosystem, being Maven the most affected one with 63% lagging dependen-

cies. This reasoning brings us to the two uses cases described below.

UC1 **Library use case:** Identify BCs introduction and whether they impact client code.
UC2 **Client use case:** Identify broken code caused by upgrading to a newer release of a library.

The aforementioned use cases can be further decomposed into a list of both functional and extra-functional requirements that must be addressed by approaches supporting API evolution.

R1 The approach must identify syntactic BCs introduction between two versions of a library.
R2 The approach must identify broken uses caused by BCs in a client project.
R3 The approach must be performed efficiently, so it can outperform language tools (*e.g.*, compiler, linker) and scale when analysing multiple clients.
R4 The approach must be accurate so it avoids erroneous information that hinders the evolution of both library and client projects. However, when facing undecidable scenarios, the approach must follow an over-approximation approach, prioritizing completeness over soundness.

4.4 STATIC IMPACT ANALYSIS: THE APPROACH

In this section, we introduce the *static impact analysis* approach, which performs the evolution analysis of an API and the succeeding impact analysis on a client project while considering the requirements defined in Section 4.3. In short, the goal of the approach is to efficiently and accurately (requirements **R3** and **R4**) identify the set of syntactic BCs introduced between two releases of a library (requirement **R1**), and the set of API uses on client code that are impacted by such changes (requirement **R2**).

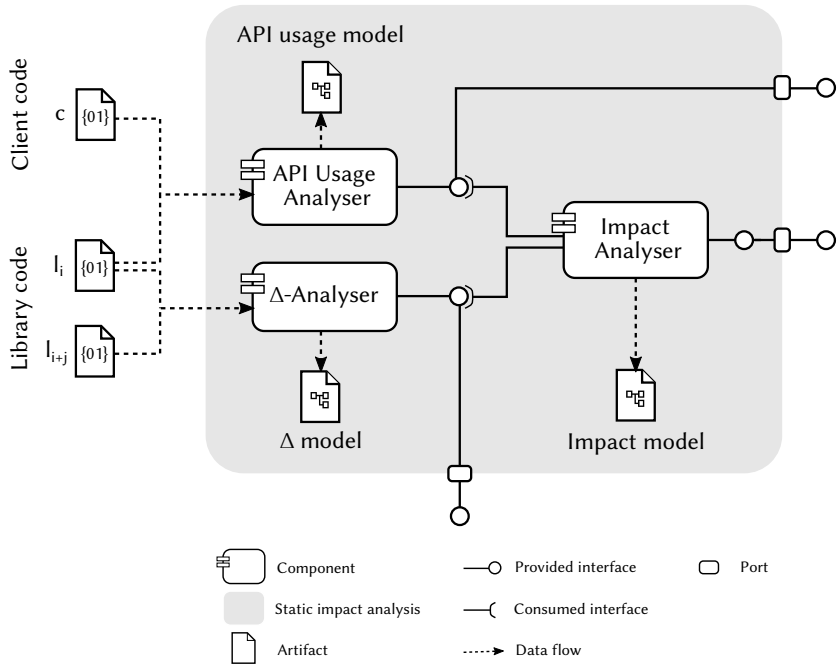


Figure 4.2: UML component diagram of the static impact analysis approach

Approach Overview

Figure 4.2 introduces our three-component approach to perform static impact analysis. The three components involved in the solution consists of the *API usage analyser*, the Δ*-analyser*, and the *impact analyser*. The approach takes as input the source or binary code of (i) an old release of a library l_i, (ii) a newer release of the same library l_{i+j}, and (iii) a client project c that depends on l_i, for $j \in \mathbb{N}$. Notice that the type of code provided for l_i and l_{i+j} must be of the same kind to perform an accurate comparison between the two. However, this is not a requirement for c. As a result, it generates (i) a Δ*-model* containing the set of changes introduced between l_i and l_{i+j}; (ii) an *API usage model* with a mapping between client members and used API members, and; (iii) an *impact model* with the set of impacted client members and the breaking API members. The Δ-model is required to address

requirement **R1**, while the API usage and impact models are needed to address requirement **R2**.

API Usage Analyser

The *API usage analyser* takes as input the byte or source code of client c and the library release l_i. The main output of the component is a model that captures the uses of a specific API in the client code. In particular, the API usage analyser reports *what* API members are being used in the client code, *how* they are being used, and *where* they are being used. For instance, in Listing 4.1, the API usage model should inform that the JUnit 4 method `testAborted(Description, Throwable)` from the `RunNotifier` class (what) has been invoked (how) twice in the Concordion method `runMethods(RunNotifier)` located in the class `RunNotifier` (where).

To compute the *API usage model*, the API usage analyser computes an annotated Abstract Syntax Tree (AST) of the client, with the structure provided by the parser and annotations on names and types as provided by the name and type analysers. The library release must be included as part of the classpath to resolve the API members. As proposed by Pawlak et al. [128], we differentiate between two types of nodes in the AST, namely *structural* and *code* nodes. On the one hand, *structural* nodes provide information about project members (*e.g.*, type, function, method, field). Usually, these nodes are top-level nodes defining the structure of the AST and containing low-level nodes with the code and logic implemented in each one of them (*aka.*, code nodes). On the other hand, *code* nodes give information about how project members are being used (*e.g.*, import statement, function call, method invocation).

The AST is, then, traversed to build the API usage model, which is a set of tuples $\mathcal{U}$, where $u = \langle CN, UAN, SAN, AU \rangle$, where $\forall u \in \mathcal{U}$. CN represents the client node using an API member, UAN represents the used API member, SAN represents the source API member, and AU represents the API type use. These four elements are further described below.

CLIENT NODE (*cf.* CN) This element is a reference to the client node that is using an API member. For example, in Java, we can refer to a code node representing a *method invocation* or a structural node representing a *class* with information about its superclass and interfaces.

USED API MEMBER (*cf.* UAN) This element is a reference to the structural node representing the API member that is being used by the client node. For example, in Java, we can refer to a structural node representing a *method* or a *class*.

SOURCE API MEMBER (*cf.* SAN) This element is a reference to the structural node representing the API member that is transitively being used by the client node via the UAN node. As it is the case of UAN, in Java, we can refer to a structural node representing a *method* or a *class*.

API USE (*cf.* AU) Abstraction that represents how the API member is being used. For example, in the case of Java, we can refer to a *method invocation* use, a superclass *extension*, or an interface *implementation*.

EXAMPLE In our JUnit 4 example, if we refer to the client code in Listing 4.1, we can extract two API uses. The first usage u has the following elements: (i) a *client node* pointing to the node in line 16—that is `notifier.testAborted(description, e .getCause());`; (ii) a *used API member* pointing to the declaration node of the `testAborted(Description, Throwable)` method in the class `RunNotifier`; (iii) a *source API member* pointing to the same *used API member* node—given that there is no transitive use of a parent method, and; (iv) an *API use* referring to a *method invocation* as defined in the JLS [57].

Δ *Analyser*

During the *evolution analysis* stage, the approach takes as input the old release l_i and the newer release l_{i+j} of a library, and generates the Δ-model with the set of changes introduced be-

tween the two releases. A Δ-model records changes performed over language constructs usually used to define the contracts of the API—that is, changes performed to constructs (*e.g.*, modifiers, names, parameters) that accompany the declaration of an API member (*e.g.*, type, field, method, or function), while excluding changes to the body of the member declaration.

Concretely, a Δ-model is an annotated tree $\Delta = \langle r, (LC, \prec) \rangle$, where:

(a) r is the root of the tree and the *unique maximal* element in the partially-ordered set $(LC, \prec)$ (it is not less than any other element in LC). The role of r is usually symbolic, meaning that it is used to gather all the compilation units found within the project, and;

(b) $(LC, \prec)$ is the strict partially-ordered finite set of modified *language constructs* within the software project. Each $lc \in LC$ is annotated with extra information referring to the type of change (*i.e.*, *removed*, *added*, or *modified*), and the previous and new value if a modification has been performed. Moreover, $(LC, \prec)$ preserves the *containment* relation among the language constructs. The partial order relation $\prec$ is a relation that preserves the scope containment of the language constructs—that is, minimal elements are those declarations that do not contain further declarations. $\prec$ is *irreflexive* (no element is related to itself, *i.e.*, $a \prec a$ for $a \in LC$), *asymmetric* (if $a \prec b$ then $b \not\prec a$ for $a, b \in LC$), and *transitive* (if $a \prec b$ and $b \prec c$, then $a \prec c$ for $a, b, c \in LC$).

EXAMPLE In the JUnit 4 example presented in Listing 4.1, the Δ-model contains a node pointing to the `testAborted(Description, Throwable)` method declaration, which was removed between versions 4.4 and 4.5. This node must be defined as a child of a class declaration node representing the owning class `RunNotifier`, which in turn is a child node of the compilation unit with the same name, and the latter is a child of the symbolic root node r. In particular, the node represents the BC type *method removed*. It is, therefore, annotated

with the label *removed* and no additional information about the new and old values is required.

Impact Analysis

The last stage of the approach, the *impact analysis*, takes as input the Δ-model and the API usage model. The impact model contains a set of *broken uses*, which represent the client code that would be broken due to the use of a BC introduced between two releases of a library. Thus, the *impact model* I is a set of tuples $\mathrm{bu} = \langle \mathrm{BC}, \mathrm{u} \rangle$ $\mathrm{u} \in \mathrm{U}$, where BC represents the BC type consisting of the type of API member and the performed modification (*e.g.*, *method removed*, *class now final*), and u an API use tuple included in the API usage model U.

To generate the impact model, the approach traverses the Δ-model to extract the set of modified API members with their corresponding BCs. API member nodes in the Δ-model are mapped to UAN and SAN nodes in the API usage model. Based on this mapping, the API usage model is filtered out, keeping the candidate tuples U' for $\mathrm{U}' \subseteq \mathrm{U}$ whose used and source API members refer to modified API members as reported by the Δ-model. Then, the client code must be inspected to verify if the API use of the modified API member is actually breaking client code. To determine it, one must consider (i) the type of BC introduced in the API member as defined by the Δ-model, and (ii) the API use for which such BC actually breaks client code. If the candidate $\mathrm{u} \in \mathrm{U}'$ breaks the client code, a tuple bu is created with the corresponding BC and API use tuple u, and then added to the impact model I.

EXAMPLE In the motivating example introduced in Listing 4.1, the impact model has a broken use bu pointing to the BC type *method removed*, as described at the end of the Δ *Analyser* section, and the usage u described at the end of the *API Usage Analyser* section.

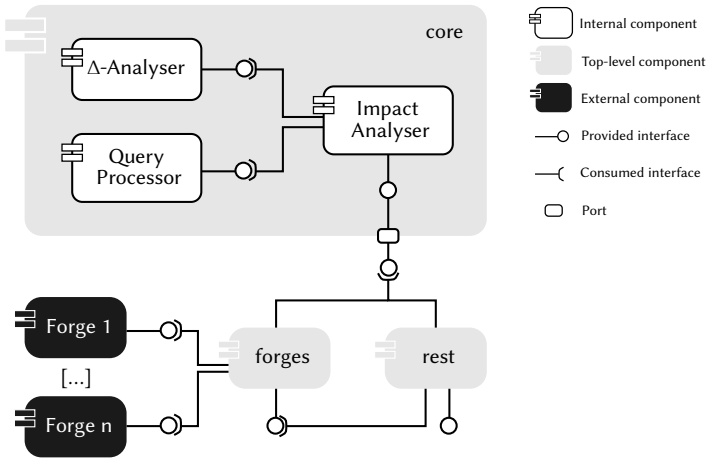


Figure 4.3: Component-view of the MARACAS architecture

4.5 MARACAS: THE IMPLEMENTATION

MARACAS is a static analysis tool that implements our three-stage static impact analysis approach for Java projects. On top of that, it offers both a REpresentational State Transfer (REST) API to support external services that leverage MARACAS capabilities, and; connects with software forges to fetch projects requested for specific API change and impact analysis. In its current version, MARACAS takes two JAR files with two different releases of a library and the source code of a client as input. The output of the tool is twofold: (i) a Δ-model reporting all BCs introduced between two releases of a library, and (ii) an impact model containing information about broken uses.

Architecture

Figure 4.3 shows a component view of the MARACAS architecture. As can be seen, MARACAS has three top-level components, namely the `core`, `rest`, and `forges` components.

The `core` component is the backbone of the tool: it is in charge of implementing the *static impact analysis* approach. The component orchestrates three sub-components, namely the *query*

processor, the Δ-*analyser*, and the *impact analyser* components. The *query processor* is used to (i) specify the type of expected analysis (*i.e.*, API change analysis or the whole static impact analysis), and; (ii) specify paths to the required code files (*i.e.*, JAR and Java source files) and add additional filtering options (*e.g.*, exclude unstable parts of the API from the analysis).

The Δ-*analyser* implements the API Δ-analysis stage of our approach. For that, it relies on `japicmp`,[7] a static analysis tool that detects a subset of BCs in Java projects. The tool takes two JAR files as input, meaning that it can only report binary BCs and the source BCs that are shared between the two sets of BCs. The output of the tool is then translated into a Maracas-specific Δ-model.

The API usage and impact analyses are implemented in the *impact analyser* component. Maracas relies on Spoon [128] to generate an AST from the client source code and via the visitor design pattern, it traverses the structure and reports the broken uses of the modified API members. To do so, there is one visitor per BC type, each one of them visiting AST nodes that under certain use conditions can be impacted by the BC. For each node, there is a broken use detection algorithm. The language construct gives information about the API usage and the used API member. Although there are several visitors for each BC type, only one traversal of the AST is required. All visitors are combined in one class, and each one of them is requested to visit each node from the AST. Together with the Δ-model and the expected behaviour of the compiler, we provide an algorithm that can statically infer broken uses. Maracas follows an over-approximation approach—that is, in cases where we cannot decide locally whether a change impacts client code, we still add a broken use to the model.

Lastly, Maracas provides both the `rest` and `forges` components. The `rest` component provides the Maracas REST API. This API enables a different way to use the tool, supporting services such as bots to leverage Maracas capabilities. The `forges` component connects Maracas to software forges like GitHub. This is done to exploit Maracas capabilities at a bigger scale.

7 https://siom79.github.io/japicmp/

For instance, with the `forges` component, we can specify a set of client repositories that need to be used in the impact analysis of a specific library. Then, the component will take care of fetching such repositories, cloning them, and performing the static impact analysis with Maracas `core`.

Testing

In this section, we showcase the methods used to test Maracas capabilities. In particular, we describe the use and design of *unit tests* based on a synthetic benchmark and Maven projects, and we refer to the accuracy evaluation of the tool presented in Chapter 5.

UNIT TESTING To test the backbone capabilities of Maracas, we implement a collection of unit tests that assess the generated models. Concretely, we discuss two types of unit tests, namely benchmark-based, and regression tests based on Maven projects.

The benchmark-based unit tests use the `comp-changes` synthetic benchmark as baseline. This benchmark consists of three main Java projects, namely the `old`, `new`, and `client` projects. The former two projects represent the old and new releases of a synthetic library. Although the `new` project preserves the same structure defined for the `old` project, it is responsible of introducing expected BCs that potentially impact the `client` project. Each package within the three projects addresses a specific BC type, and every class within a package addresses a specific scenario where a BC is used in a particular way. The projects cover 41 types of BCs as defined by the `japicmp` project. To date, Maracas has 377 benchmark-based unit tests. The tests have been generated by manually inspecting the errors reported by the compiler when upgrading the `client` project to use the `new` project.

The regression tests consider a set of releases of Maven libraries (*e.g.*, Google Guava, Spoon, Log4J). Particularly, we consider two different releases of each Maven project, and use the first release as the client upgrading to the newer version of the library. For instance, consider Google Guava releases 18.0 and 19.0. Both

releases are used to compute the Δ-model. In addition, release 18.0 is used as the potentially impacted client of the upgrade. We do so, assuming that the library is internally making use of the modified code, for instance when using API members in their test suites. These tests download the requested releases and perform some sanity checks on the generated Δ-models and impact models. To date, Maracas has a total of 19 regression unit tests.

ACCURACY EVALUATION To test the accuracy of Maracas, we performed an accuracy evaluation included as part of the study assessment presented in Chapter 5. In such evaluation, we focus on answering the question *how accurate is* Maracas *in detecting the impact of BCs on clients?* To do so, we consider two synthetic benchmarks, the previously mentioned `comp-changes` and the *API evolution data* [70] benchmarks. Both benchmarks provide two library releases and a client that depends on the first release of the library. The second library release introduces a set of BCs that can impact the client project under certain circumstances. While using Maven as build system, we automatically upgrade the dependency of the client project to the newest release of the library. This is done by injecting the new version of the library in the dependencies defined in the client Project Object Model (POM) file. We then use the warning and error messages reported by the Maven compiler as our ground truth to verify if Maracas correctly identifies the corresponding broken uses in client code. We test the accuracy of the tool in terms of precision and recall. After performing the evaluation, we find out that Maracas reports a precision of 0.96 and a recall of 0.95 on the `comp-changes` benchmark, and a precision of 0.92 and a recall of 0.90 on the *API evolution data* benchmark. To dive into the details of the evaluation and the limitations of the tool, we refer the reader to the Chapter 5.

In this section, we present current solutions performing the evolution and impact analysis of library releases in Java projects. We also include the advantages and disadvantages of such studies in light of our work.

Evolution Analysis: Detection of Breaking Changes

The Java compiler and linker systems are the main sources to identify the source and binary BCs, respectively. However, these tools cannot report on the nature of the BCs (*e.g.*, changing the access modifier of a method, making a class final) [162]. They also introduce some overhead in terms of compilation and linking time, requiring involved software projects to be compilable and executable—which turns on to be impractical when performing empirical compatibility analysis of several evolving projects.

Researchers in the software evolution field have proposed empirical and statistical approaches, among others, to overcome the aforementioned limitations when performing evolution analysis. For instance, Robbes, Lungu, and Röthlisberger [136] detect ripple effects caused by the deprecation of API methods and classes in Squeak and Pharo, both of them dialects of Smalltalk. They first identify the API members that have the *deprecated* annotation. Afterwards, they mine commits of client code and pinpoint the cases where the use of the deprecated API member has been modified. These members are labeled as ripple effect candidates and only the ones that have been modified in at least three commits are kept in the list. In general, this study does not focus on BCs detection but rather on the detection of API deprecation and the client reactions to such changes. Moreover, the study does not aim at deeply studying the impact on client code.

A different set of static analysis tools have been developed to identify the introduction of BCs [70, 162]. Examples of such tools include `clirr` [89], `jacc`, `japicc`, `japiChecker`, `japicmp` [59], `japitool`, `jour`, `revapi`, `sigtest`, and ACUA [162]. Usually, these tools take two JAR files or source folders as input and generate a

model with the list of BCs introduced between two releases of a project. Jezek and Dietrich find out that only a small portion of these tools are suitable to detect BCs: they are neither sound nor complete—that is, not all reported BCs are truly BCs, and not all existing BCs are reported by the tools. In particular, tools dealing with bytecode will not be able to detect source incompatible changes due to loss of information after compilation. Furthermore, impact on client code is out of the scope of such tools. However, the aforementioned tools do overcome the limitations imposed by the use of the compiler or the linker: they eliminate the compilation and linking resources overhead, and support BC types identification.

Impact Analysis: Detection of Breaking Changes Impact

Robbes, Lungu, and Röthlisberger [136] are some of the first researchers that pointed out the need for tools that can identify how BCs introduced in a library can impact client projects and trigger a wave of changes in a software ecosystem. As mentioned previously, they perform the impact analysis of deprecated API methods and classes. The authors introduced the Ripple Effect Browser tool, which visualizes the *diff* of the commit on the client side that modifies the use of the deprecated API member. In this way, they can identify the number of reacting clients, the magnitude of the change triggered by the deprecation, and the copying BC strategy. Although the previous approach gives relevant insights into the type of changes that succeed an API change, the scalability and performance of the solution are compromised by the required manual labour and the granularity level of the analysis (*i.e.*, commit level).

Wu et al. [162] introduce ACUA, a static analysis tool that goes a step further and performs API evolution and impact analysis. The tool takes as input the JAR files of two releases of a library and a client of the first release. Then, a model is generated from these files by means of using the ASM Java bytecode analysis framework. On the impact analysis part, the tool detects *where* and *how* API members are being used on client code. Based on

this information and the set of identified BCs, ACUA traces back broken client code to specific BCs. Nevertheless, maintenance of ACUA has been discontinued and details about the implementation of the approach are not provided. In particular, the way impact analysis is performed can have a non-negligible effect on the accuracy of the results.

Xavier et al. [165] also study API evolution and impact but instead, they use a heuristic to estimate it: (i) in the case of a *type*, they check if the client refers to the fully qualified name of the class in any of its `import` statements, and; (ii) in the case of a *field* or a *method*, they consider the owning type to perform the previous check. This approach might not be resource-intensive but the impact on the accuracy of the output is compromised. Not all imported types in a class are actually used by the client, and not all uses of certain breaking API members derive into broken client code. Maracas, on the contrary, does consider the specific usages of the API, and an algorithm mimics the behaviour of the compiler to provide more accurate insights.

Furthermore, Scalabrino et al. [141] study backwards incompatibility in the Android ecosystem. Their approach investigates Conditional API Usages (CAUs) in Android applications to detect BCs and infer patches. A CAU is a code block that checks the current Android version of the running app, and depending on the value provides a specific behaviour compatible with such a release. The CAUs are used to infer patching rules setting a confidence level. This confidence level is strongly related to the number of applications exhibiting a specific behaviour for a given Android version. The rules are not only used to directly patch Android clients, but also to detect impacted code on other applications that do not necessarily use CAUs. However, this approach is data-intensive, requiring several clients to provide meaningful patching rules. Moreover, the approach is not complete in the sense that BCs are detected only for those cases appearing on the Android application codebase.

4.7 CONCLUSIONS

In this chapter, we introduce a *static impact analysis* approach to perform API usage, evolution, and impact analysis in a language-agnostic, resource-efficient, and accurate way. In particular, the goal of the approach is to identify the set of syntactic BCs introduced between two releases of a library, and the set of broken uses on client code caused by such changes.

Our approach consists of three components, namely (i) *the API usage analyser*, which takes as input a library release and a client project and outputs an API usage model that contains information about *what* API members are being used within the client, *where* they are being used, and *how* (*e.g.*, interface implementation, method invocation); (ii) the Δ-*analyser*, which takes as input two library releases and generates the Δ-model with all introduced BCs, and; (iii) the *impact analyser*, which takes the API usage model and the Δ-model as inputs and builds an impact model that links client broken code to BCs. The static impact analysis approach is implemented into MARACAS, which performs the corresponding analysis on Java code. Both unit tests and an accuracy evaluation have been implemented to test the tool. Two benchmarks have been considered for the latter evaluation, namely the `comp-changes` and the *API evolution data* benchmarks. MARACAS reports a precision of 0.96 and a recall of 0.95 on the `comp-changes` benchmark, and a precision of 0.92 and a recall of 0.90 on the *API evolution data* benchmark.

The previous results suggest that the *static impact analysis* approach, together with our implementation, MARACAS, are accurate solutions to help both library and client developers to perform API evolution and impact analysis. An additional evaluation must be performed in the future to verify that the approach is also language-agnostic and resource-efficient, beating alternatives that include Java compilers and build systems. Furthermore, the limitations of the tooling (*e.g.*, handling generics, estimating the inheritance hierarchy) should also be addressed to increase the accuracy of the solution.

As future research directions, we plan to use the static impact analysis approach and the tools implementing it for different programming languages (*e.g.*, MARACAS) as an infrastructure to empower library and client developers when evolving their projects. In particular, we believe that the output models can be used, for instance, to (i) suggest improvements to the design of software projects; (ii) decide beforehand whether to introduce a BC based on the impact it has on client code, or decide to upgrade to a newer version of an API based on the broken uses introduced in a client project; (iii) measure the effort of upgrading to new releases of a library, and; (iv) identify reaction patterns when upgrading to a new version of an API [140] and even infer rules to patch clients that undergo the same process [141].

5

BREAKBOT: STATIC REVERSE DEPENDENCY COMPATIBILITY TESTING FOR JAVA LIBRARIES

ABSTRACT *"If we make this change to our code, how will it impact our clients?" It is difficult for library maintainers to answer this simple—yet essential!—question when evolving their libraries. Library maintainers are constantly balancing between two opposing positions: make changes at the risk of breaking some of their clients or avoid changes and maintain compatibility at the cost of immobility and growing technical debt. We argue that the lack of objective usage data and tool support leaves maintainers with their own subjective perception of their community to make these decisions. We introduce* BreakBot*, a bot that analyses the pull requests of Java libraries on GitHub to identify the breaking changes they introduce and their impact on client projects. Through static analysis of libraries and clients, it extracts and summarizes objective data that enrich the code review process by providing maintainers with the appropriate information to decide whether—and how—changes should be accepted, directly in the pull requests. Our accuracy evaluation shows that* BreakBot *is able to detect breaking changes and their impact using static analysis on two benchmarks with precision and recall scores $\geqslant 90\%$. By analysing thousands of pull requests on popular Java libraries hosted on GitHub, our usefulness evaluation shows that many are introducing breaking changes and impacting client projects, and that* BreakBot *provides insightful information to maintainers in the code review process.*

This chapter is originally submitted as Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen J. Vinju. "BreakBot: Static Reverse Dependency Compatibility Testing for Java Libraries". In: IEEE Transactions on Software Engineering (2023). IEEE, (Submitted).

Leveraging the time-honored principles of modularity and reuse, modern software systems development typically entails the use of external *software libraries*. Rather than implementing new systems from scratch, developers incorporate libraries that provide functionalities of interest into their projects. Such libraries expose their features through Application Programming Interfaces (APIs), which govern the interactions between client projects and libraries.

Libraries constantly evolve to incorporate new features, bug fixes, security patches, refactorings, and extra-functional improvements [12, 20]. It is critical for clients to stay up-to-date with the libraries they use to benefit from these improvements and to avoid technical lag and its associated technical debt [27, 55, 150]. When a library evolves, however, it may break the contract previously established with its clients by introducing Breaking Change (BC) in its API, resulting in syntactic and semantic errors [172]. In Java, for instance, seemingly innocuous changes such as altering the visibility or abstractness modifier of a type declaration or inserting a new abstract method can, under certain conditions, break client code. Errors triggered by these changes burden client developers, given the sudden urgency to fix issues out of their control without intrinsic motivation. As a result, clients sometimes hesitate to upgrade their dependencies, raising security concerns and making future upgrades even more difficult. Thus, it does not come as a surprise that the problem of helping clients respond to library evolution has garnered considerable interest in recent years [93, 110, 120, 163, 168].

Surprisingly, however, the problem of helping library maintainers anticipate the impact of their changes and plan accordingly—the other side of the coin—has received little attention. Libraries are in general accountable to their clients for providing stability [18]. We claim that library maintainers currently lack the necessary information and tool support to live up to this responsibility. In particular, these maintainers have limited means to foresee the consequences of their actions on their clients [66, 136] and would benefit from knowing precisely how their APIs are used in client projects.

The consequences of breaking an API declaration used by many clients, by a popular library, or by commercial clients might be substantial: clients might lose trust and either stay outdated or migrate to another API. Consequently, some very cautious library maintainers refuse to change any existing declaration, thinking it might be too impactful, leading to an accumulation of technical debt and an aging design that will scare people away [18]. Some more adventurous maintainers will push any change, disregarding clients using the affected declaration, at the risk of breaking too many of them and having to revert the change ultimately [145]. We believe that the sweet spot is at the crossroad of innovation[2] and clients' stability. Unfortunately, there are only limited ways for library maintainers to acquire evidence-based knowledge on the usage of their API, and they often must rely on their own judgment and the subjective knowledge of their community to figure out how to evolve it [173].

To assist maintainers in the evolution of their libraries while avoiding breaking their clients unexpectedly, we introduce a general framework for reasoning about the impact of library evolution: static Reverse Dependency Compatibility Testing (RDCT), as well as a first implementation: BreakBot. Static RDCT aims at empowering library maintainers to make evidence-backed decisions regarding the evolution of their libraries by efficiently and accurately informing them about BCs introduction and their impact on relevant clients. Concretely, library maintainers are able to ask *what-if* questions and explore what consequences evolving their library would have on their clients: "*What if we push this refactoring?*" "*What if we alter this method's signature?*" *etc.*,.

BreakBot is a GitHub bot that analyses the content of Pull Requests (PRs) in Java repositories and employs static analysis to identify BCs (on the library side) and their impact (on the client side). The resulting reports are fed back into the PRs, enabling maintainers to review changes and their impact to decide whether they should be accepted. BreakBot relies on Maracas

2 We adopt the description of "innovation" provided by Bogart et al. [18]—that is "[i]nnovation through fast and potentially disruptive changes".

(*cf.* Chapter 4), a static analysis tool that detects syntactic BCs that force clients to modify their code at upgrade time.

We evaluate BreakBot—and, indirectly, the static RDCT approach—on two fronts: usefulness and accuracy. As main results, we find that BreakBot (and indirectly the static RDCT approach) produces meaningful reports for a varied set of library evolution scenarios, and that many PRs do indeed introduce BCs, sometimes impacting clients, even in the most popular Java libraries. Besides, our accuracy evaluation concludes that Maracas, obtains excellent precision and recall scores. We expect our framework and its implementation in BreakBot to fight immobility and stagnation by pushing library maintainers to introduce BCs when it is safe and to avoid pushing changes when they are considered too impactful. Making the co-evolution of library and clients more harmonious should benefit library maintainers and client developers alike.

This chapter is an extension of our earlier short paper published in the New Ideas and Emerging Results (NIER) track of the International Conference on Software Engineering (ICSE'22), which outlined our vision [121]. This chapter extends the latter with a general framework for static RDCT, a detailed presentation of BreakBot, and an evaluation of its usefulness and accuracy. The remainder of the chapter is structured as follows. Section 5.1 introduces background notions on software evolution and software forges. Section 5.2 dives into a real-world evolution scenario that motivates our approach. Static RDCT and its implementation, BreakBot, are described in Section 5.3 and Section 5.4, respectively. Section 5.5 evaluates the useful and accuracy of BreakBot. Finally, we discuss our main findings in Section 5.6, related work in Section 5.7, and conclude the study in Section 5.8.

5.1 Background

This section introduces core concepts regarding software evolution, software forges, and version control systems, which are used throughout the manuscript.

Software Evolution

Software evolution is the phenomenon that results from modifying or maintaining software [80]. Often, this evolution affects more than a single software project and propagates to other projects as they are linked through dependencies and cohabit within software ecosystems. A software project can act both as a *library* and a *client*. When playing the library role, a change introduced in the project can impact its API, *i.e.*, an interface that exposes a set of features and services to client projects. Changes introduced in the API can either be backward-incompatible or compatible, meaning that they can or cannot potentially break client code, respectively. Backward-incompatible changes are also known as BCs and can be classified as *syntactic* or *semantic*. Syntactic BCs impact the structure of the API (*e.g.*, changing the return type of a method, the name of a type, or a parameter list), while semantic BCs impact its behavior (*e.g.*, for a given input, a different output is observed) [172]. BCs are language-specific—that is, they depend on the specification and implementation of the host programming language. For instance, BCs related to the modification of access modifiers exist in Java but not in Python, as the latter does not support such constructs.

Software Forge & Version Control System

A *software forge* is a collaborative system where software projects are hosted, developed, and shared [147]. The code and other artifacts of the project are managed via a Version Control System (VCS) which facilitates collaborative development and keeps track of the different versions of the project assets, to name but a few capabilities. GitHub is an example of a software forge relying on Git as VCS. All assets of a project are part of a *repository*, which consists of a tree of commits, each with links to its predecessors. A *commit* represents a chunk of changes (*diff*) to the files of the repository. It is usually labeled with a message and has a pointer to a *snapshot* of the repository at the moment the commit was created. Commits can be spread around differ-

ent *branches* representing various development endeavors (*e.g.*, feature development, bug fixing, refactoring). Additionally, the development and operational workflow of the team developing the project are supported by additional tools such as Continuous Integration (CI), Continuous Development (CD), issue trackers, and bots of different natures (*e.g.*, Dependabot),[3] among others.

5.2 MOTIVATION & CURRENT SOLUTIONS

To understand the importance of assisting library maintainers when introducing BCs and assessing their impact on client code, we use an example brought to our attention by a core maintainer of Spoon,[4] a source code analysis and transformation library for Java developed on GitHub. Spoon developers are very concerned about stability and therefore avoid BCs. When they do want to introduce a BC, they first use deprecation to warn clients before proceeding with the change. In pull request PR#2683, a new feature to pretty-print Java import statements was rolled in, making a previous implementation—the `ImportScanner` interface and corresponding `ImportScannerImpl` implementation and `ImportScannerTest` tests—obsolete. Therefore, the author of the PR labeled both types as deprecated. Two months later, in PR#3184, Spoon developers removed a set of deprecated types and methods from the code, notably the two types mentioned above. The PR was merged, making its way into Spoon's main branch. However, three months later, Spoon's developers noticed that these changes broke two important clients (Astor [106] and DSpot [31]) and realized that the `ImportScanner` interface and implementation were used by clients and thus reconsidered the deprecation. They finally reintroduced them in PR#3266.

Clearly, having to revert changes after introducing them is not ideal. If other changes had been made based on the original changes, they would have had to be reverted too, triggering a nasty ripple effect within the library's code. To avoid this undesirable situation, there are, to the best of our knowledge,

3 https://github.com/dependabot/
4 https://github.com/INRIA/spoon/

three approaches that enable library maintainers to evaluate the impact of the BCs they introduce.

REGRESSION TESTING The most common approach is to use a regression test suite, launched after each change. This test suite contains test cases that ensure that the usage scenarios of the library, as intended by its maintainers, work as expected. With a strong enough test suite, BCs should make some test cases fail and, thus, help library maintainers assess the impact of their changes. However, regression test suites suffer one severe drawback. Since it is neither possible nor desirable to build a test suite covering every possible library usage, developers typically enforce the usage scenarios they deem important to their library using their own subjective opinion. These cases may not accurately represent real usage scenarios of the library and therefore convey a false sense of security by omitting some popular usages or by including scenarios that seldom happen in practice. In our example case, Spoon's test suite did notice the BCs introduced in PR#3184. However, the maintainers estimated that it did not represent valid usages of the library any longer and removed the corresponding test cases (the `ImportScannerTest` class).

STATIC ANALYSIS Another popular approach is to employ dedicated static analysis tools to systematically search for BCs. Various tools can statically scan two versions of a library to output the list of BCs between them [70]. For instance, Guava relies on JDiff [56] while the Apache foundation uses `japicmp` [59]. The main drawback of this approach is that it might result in an overly conservative approach when dealing with BCs introduction. Indeed, static analysis tools issue a warning whenever a BC is introduced. However, many changes, including seemingly innocuous ones (*e.g.*, simply inserting a new method in a class), are potentially breaking and therefore labeled as BCs, even though they are safe for the vast majority of clients (*cf.* Chapter 3). This typically results in static analysis reports containing dozens of warnings about BCs, with only a small fraction of them having

Listing 5.1: Excerpt of `japicmp`'s output for PR#3184

```
1 ---! REMOVED INTERFACE: PUBLIC(-) ABSTRACT(-) spoon.reflect.visitor
      .ImportScanner  (not serializable)
2 ---  REMOVED ANNOTATION: java.lang.Deprecated
3
4 ---! REMOVED CLASS: PUBLIC(-) spoon.reflect.visitor.
      ImportScannerImpl  (not serializable)
5 ---! REMOVED INTERFACE: spoon.reflect.visitor.ImportScanner
6 ---  REMOVED ANNOTATION: java.lang.Deprecated
7
8 [...]
```

actual usages in client projects. Ultimately, maintainers must again resort to their subjective opinion about the BCs detected by these tools to decide whether they may significantly impact clients.

Listing 5.1 shows an excerpt of `japicmp`'s output artificially produced for PR#3184 (Spoon maintainers do not use this tool in their development process). It signals the deleted `ImportScanner` and `ImportScannerImpl` types, together with other BCs. However, it does not bring any more information about the impact of the changes than what the regression test suite already reported. One can assume that Spoon maintainers would still have proceeded with the change despite the warnings.

REVERSE DEPENDENCY COMPATIBILITY TESTING A third and less common approach—the only one able to evaluate the impact on real clients—is Reverse Dependency Compatibility Testing (RDCT) [26, 174]. The idea of RDCT is to identify a set of relevant clients, retrieve their source code, inject the new version of the library in their dependencies, and finally build them and run their test suite. If any error is detected during the process, maintainers can analyse it by browsing through the resulting log files. This approach is popular in the realm of programming languages: Scala and Rust, for example, periodically perform RDCT on their clients (standard library). The Coq project[5] goes a step further by directly integrating RDCT as part of its CI process

5 https://coq.inria.fr/

so that any BC impacting a client is detected right away [174]. Although examples are rare, some regular libraries also employ RDCT as part of their evolution process (*e.g.*, Spoon).[6]

RDCT is an effective technique to assess whether changes impact real clients, but it suffers from several issues in practice. First, selected clients must be up-to-date with the latest version of the library, must successfully build, and must have a test suite that passes so BCs impact can be effectively detected. Ensuring all these conditions are met and setting up an environment to make it happen requires considerable time and resources. Even with a suitable environment, clients might be in a state of development where they are not buildable or where their test suites are not passing, making them useless. Second, RDCT takes considerable time and computational resources to build and test every client. For instance, the Rust project reports requiring up to a week to check 74,234 clients using `crater`,[7] as of September 2019. Third, RDCT uses the results of compilation and individual tests (pass or fail) to spot BCs in clients. However, it is very common for maintainers to introduce several BCs simultaneously. Pinpointing which ones are the problematic BCs for a given client requires analysing the resulting log files and diagnosing the source of the error. Unfortunately, the resulting log files are verbose, forcing the maintainers to filter irrelevant information to understand the root cause. Last, whenever a BC is merged, the impacted clients can no longer be used for subsequent RDCT until their affected code is repaired. This might not be a problem for BCs with a small impact, but for BCs with a significant impact, it can result in the impossibility of performing RDCT until all clients have adopted the new version of the library. For this reason, Coq's maintainers resort to fixing the clients themselves to keep their CI process running, which requires considerable effort [174].

In our example case, Spoon's RDCT runs every day on 13 projects. Following PR#3184, it ran into errors for the Astor and DSpot clients. An analysis of the log file of Astor (Listing 5.2), containing 266 errors among 704 log statements, revealed

6 https://ci.inria.fr/sos/

7 https://github.com/rust-lang/crater/

Listing 5.2: Extract of Astor's build log file (704 lines in total)

```
1  [ERROR] Failed to execute goal org.apache.maven.plugins:maven-
       compiler-plugin:3.0:compile (default-compile) on project astor:
        Compilation failure: Compilation failure:
2  [ERROR] /builds/workspace/astor/src/main/java/fr/inria/astor/
       approaches/scaffold/scaffoldgeneration/libinfo/LibParser.java
       :[23,29] cannot find symbol
3  [ERROR] symbol:   class ImportScanner
4  [ERROR] location: package spoon.reflect.visitor
5  [ERROR] /builds/workspace/astor/src/main/java/fr/inria/astor/
       approaches/scaffold/scaffoldgeneration/libinfo/LibParser.java
       :[24,29] cannot find symbol
6  [ERROR] symbol:   class ImportScannerImpl
7  [ERROR] location: package spoon.reflect.visitor
8  [ERROR] /builds/workspace/astor/src/main/java/fr/inria/astor/
       approaches/scaffold/scaffoldgeneration/libinfo/LibParser.java
       :[53,9] cannot find symbol
9  [ERROR] symbol:   class ImportScanner
10 [ERROR] location: class fr.inria.astor.approaches.scaffold.
       scaffoldgeneration.libinfo.LibParser
11 [ERROR] /builds/workspace/astor/src/main/java/fr/inria/astor/
       approaches/scaffold/scaffoldgeneration/libinfo/LibParser.java
       :[53,43] cannot find symbol
12 [ERROR] symbol:   class ImportScannerImpl
13 [ERROR] location: class fr.inria.astor.approaches.scaffold.
       scaffoldgeneration.libinfo.LibParser
```

that the deleted types (`ImportScanner` and `ImportScannerImpl`) raised missing symbol errors at compile time. Thanks to RDCT, Spoon developers discovered the impact of the BC. Based on this information, they decided to revert the removal of these two types. However, Astor's RDCT build was already failing for three months before the revert took place, making it unusable for detecting the impact of other BCs in this period.

In summary, while RDCT helps analyse the impact of BCs on clients, it has several issues that severely hinder its usability and adoption. In the next section, we present a novel and lightweight approach based on static analysis to evaluate the impact of syntactic BCs on clients that can easily integrate with the library maintainers' workflow. Naturally, RDCT still goes beyond our

approach regarding semantic BCs, at the cost of building and running the client test suites.

5.3 STATIC RDCT

In this section, we introduce *static* RDCT, an approach that supports library maintainers in the evolution of their libraries by analysing what changes are introduced in their APIs, and where and how they impact client code. This section focuses on the principles and main components of the approach, independently from a particular programming language, software forge, or development workflow. We introduce BREAKBOT later in Section 5.4 as an implementation of the static RDCT approach for Java libraries hosted on GitHub.

Approach Overview

Static RDCT aims to collect factual information from clients regarding their usage of a library and the impact the library's evolution may have. Our proposal for static RDCT solves the three main issues identified with classical RDCT: (i) it does not require clients to be healthy (*i.e.*, to compile and have a passing test suite); (ii) it drastically reduces the amount of resources needed to analyse the clients by employing static analysis; (iii) it uses dedicated reports to feed the information to maintainers, rather than piggybacking on inadequate formats such as build logs, and; (iv) it enables clients that are not up-to-date or that have recently been impacted by BCs to remain usable in the compatibility checking process.

Figure 5.1 depicts an overview of the static RDCT approach, which involves three main components orchestrating the whole solution: the *client explorer*, the *static impact analyser*, and the *impact reporter*. The approach considers a software forge hosting a set of repositories. Of particular interest are the repository of the studied library and the client repositories that depend on it. Zooming into one particular repository, the *library repository*, we

observe an excerpt of the commit tree, where $commit_{j+1}$ points to its predecessor $commit_j$. Each commit also holds a pointer to a *snapshot* of the source code and additional repository files.

To perform API evolution and impact analysis, one must first pick two versions of the library and the corresponding snapshots. According to the library maintainers' needs, these two versions may be two subsequent commits, the latest commits from two different branches, the latest commit and a reference commit (*e.g.*, the latest stable release), or any other combination of arbitrary commits. Then, the client explorer gathers snapshots of relevant client repositories that depend on the target library. The static impact analyser takes as inputs the two snapshots of the library to generate a Δ-*model* between the two library versions (which lists the BCs introduced in between), and the set of client projects to generate a set of *impact models* which pinpoint the locations in the clients' source code that are impacted by the changes identified in the Δ-model. These models are input to the impact reporter, which generates insightful impact reports fed back into the library development workflow. Hereafter, we provide more details on how the three main components of static RDCT operate.

Client Explorer

The *client explorer* is responsible for discovering, selecting, and fetching client projects of interest for the library under study. Naturally, the complete list of clients depending on a library is unknown: not every client is publicly available online, and it is difficult to identify all those that are. Several software forges and ecosystems come with their own means to discover the reverse dependencies of a library, *e.g.*, the Maven Dependency Graph [13] and the GitHub Dependency Graph,[8] which can be queried to identify clients. In other cases, the library maintainers may come up with a custom list of clients—as is already the case with classical RDCT, *e.g.*, in Coq and Spoon.

8 https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph/

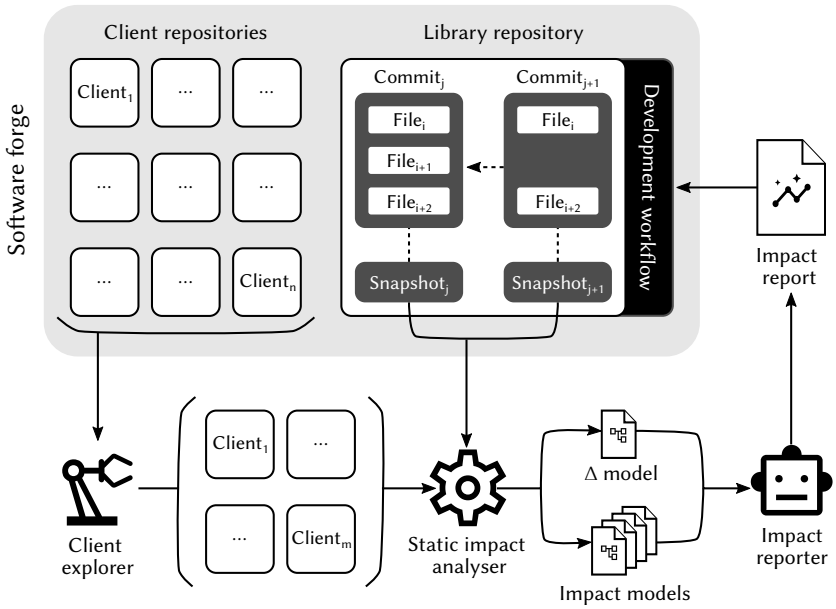


Figure 5.1: Overview of the static RDCT approach

As it is not desirable nor possible to exhaustively analyse all clients of a library (popular libraries on GitHub and npm typically reach hundreds of thousands of clients [37, 122]), the client explorer honors additional criteria to reduce the analysis space. As an illustration, BreakBot enables library maintainers to limit the analysis to the top m most popular clients according to GitHub's stars score, where m is an arbitrary number (*cf.* Section 5.4). Other possible criteria include coverage of the library's API by the clients (to avoid analysing several clients using the library in the same way and to make sure that the library's API is adequately covered by the selected ones); diversity and representativeness of the clients w.r.t. chosen dimensions [119]; nature of the library, in particular, if it is a engineered software project or "noise" [118]; update date, or; importance of the client to library maintainers.

Note that, for the approach to be accurate, clients do not necessarily need to depend on the latest version of the library under study. For instance, if maintainers remove in version `2.0`

a method `m()` that was introduced in version `1.1`, any client depending on version $v \in [1.1, 2.0)$ and using the method `m()` will be impacted when it upgrades to $v' \in [2.0, \infty)$. Naturally, clients that have not been updated in a long time (*e.g.*, clients that depend on version `2.x` while the library is about to release version `4.1`) may not be as interesting to analyse since they may never update to the latest version and, when they do, they will be faced with other major changes. Ultimately, the decision is up to the maintainers, who should be given the means to implement their own policy.

Static Impact Analyser

The static impact analyser is in charge of both analysing the changes introduced between two versions of a library and the impact these changes have on a particular client project. After the client explorer has fetched the client repository snapshots, the *static impact analyser* takes the client sample (C) and two snapshots of the target library (l_1 and l_2) corresponding to two different commits in the repository. These commits may be adjacent or not. Adjacent commits are commits that are directly connected in the commit tree, where one commit (*e.g.*, $\mathtt{commit}_j$) is the predecessor of the other one (*e.g.*, $\mathtt{commit}_{j+1}$). In either case, the two commits can be in the same or different branches within the VCS. Based on these inputs, the static impact analyser performs a static API evolution analysis $\Delta\langle l_1, l_2\rangle$ generating a Δ-model that contains all BCs introduced between l_1 and l_2. Afterward, it performs a static impact analysis $I\langle\Delta_{l_1 l_2}, c\rangle$, $\forall c \in C$ building a set of impact models containing pointers to broken client code after the upgrade from l_1 to l_2. To produce these assets, the static impact analyser creates an Abstract Syntax Tree (AST) of each library and client snapshot, avoiding the need to build and compile the software projects. Then, it traverses the library ASTs to identify the language-specific BCs. This information is later used when traversing the clients' ASTs to mimic the behaviour of the host language tools (*e.g.*, compiler, linker) and identify broken client code.

Impact Reporter

Ultimately, the *impact reporter* takes the Δ-model and the impact models as inputs to generate an impact report. This report provides library developers with insights into the BCs introduced between two commits: what kinds of BC affects which declaration, in which location in the source code, and how such changes break client code. The report is later fed back to the development workflow providing actionable information about the library evolution to the project maintainers. Based on this information, library maintainers can opt for totally or partially including the introduced changes between the two commits or rejecting them. They can also design strategies to support clients in their upgrade process and better enforce their own policies regarding backward compatibility and library evolution.

5.4 BREAKBOT

BreakBot is our implementation of static RDCT for Java libraries hosted on GitHub and following a pull-based development workflow [58]. It is implemented as a GitHub App that analyses the PRs of Java libraries hosted on GitHub. In particular, it pinpoints the BCs introduced in the PR and their impact on a configurable set of client repositories. This information is fed back into the development workflow in the form of a report attached to the PR to support the code review process (see `https://github.com/break-bot/spoon-before-bc/pull/4/checks/` for an example report on the scenario presented in Section 5.2). In this section, we present how we implemented the three components of static RDCT for BreakBot.

Client Explorer

In BreakBot, clients of a library hosted on GitHub are extracted from the GitHub Dependency Graph. As there is currently no dedicated API to retrieve clients of a repository, we implemented

Listing 5.3: An example BreakBot configuration file

```yaml
clients:
  top: 10
  stars: 5
  repositories:
    - repository: foo/bar
      sha: 6c3c1e
      module: bar-core
    - repository: bar/baz
      branch: 3.x
```

a crawler that extracts the list of clients from the *Dependency Graph* webpage of the library under study, together with associated popularity metrics (number of stars, number of forks). Users of BreakBot can configure which of these clients should be taken into account during the impact analysis phase through a dedicated YAML configuration file `breakbot.yml` that must be placed in a `.github` directory at the root of the library's repository, following GitHub's conventions. In this file, users can define a threshold of popularity and a maximum number of clients to be checked, as illustrated in Listing 5.3. Given this configuration, the client explorer computes the list of clients ordered by popularity and returns the top n. In addition, users may provide a custom list of client repositories that should always be checked. As an example, the configuration of Listing 5.3 will instruct BreakBot to automatically fetch and analyse the ten clients using the library with the most stars and a minimum of five stars, together with the repository `foo/bar` (at commit `6c3c1e`) and the `bar-core` Maven module of the `bar/baz` repository on branch `3.x`.

Static Impact Analyser

The static impact analyser is implemented as a standalone tool named Maracas. Maracas is a static analysis tool written in Java that performs API changes and impact analysis on Java

projects.[9] In particular, Maracas takes two versions of a library (as source code or bytecode) and the source code of a client project that depends on the first version of the library as inputs to conduct the static analysis. In earlier work, we implemented a first version of Maracas in Rascal [84] to analyse hundreds of thousands of Java ARchivess (JARs) hosted in Maven Central Repository (MCR) (*cf.* Chapter 3). To use it in concert with BreakBot, we re-implemented Maracas in pure Java to improve the tool's performance and better integrate it with required Java libraries; *e.g.*, `japicmp` for Δ-model computation, and the Spoon library [128] for source code analysis.

It consists of three core components: (i) Maracas `core` performs the static API change and impact analysis; (ii) Maracas `forges` handles the communication with software forges, enabling Maracas to analyse source code hosted on remote repositories, and; (iii) Maracas `rest` makes it easy for third-party clients (such as BreakBot) to leverage the analysis capabilities of Maracas through a REpresentational State Transfer (REST) API.

To detect the BCs introduced between two arbitrary versions of a library, Maracas builds the corresponding snapshots using Maven to produce their JARs. The two JARs are then passed through `japicmp`, which returns the declarations affected by a BC and the kind of change (*e.g.*, a method removal or a change in a method's return type). The list of BCs is stored in a Δ-model.

To detect the impact a Δ-model has on a particular client, Maracas first builds an AST of the client's source code. Then, for each BC in the Δ-model, Maracas instantiates a dedicated visitor of the appropriate BC kind, configured by the API declaration that is affected. We implemented one kind of visitor per BC type. Each visitor subscribes to the AST node types that may be affected: a `MethodRemovedVisitor`, for instance, looks for method invocations and method overrides in the client's AST that point towards the now-removed method in the library. The set of all visitors for a particular Δ-model is then combined into a single one so that the whole analysis runs efficiently in a single pass. Each element in the client's AST that is impacted by a BC (*e.g.*, a

9 https://github.com/alien-tools/maracas/

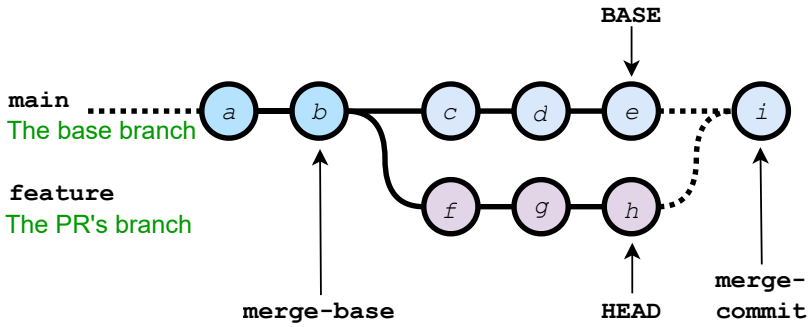


Figure 5.2: By default, BreakBot compares the `merge-base` and the `HEAD` commits when analysing a PR

method invocation, field access, or interface implementation) is called a *broken use* and is associated with its path, location, and kind of use. Together, the set of broken uses for a particular client forms its impact model.

Reporter

Our BreakBot reporter is designed to act on GitHub PRs and is implemented as a GitHub App that reacts to PRs events by subscribing to GitHub webhooks.[10] When a PR is created or updated, a notification is sent to BreakBot. By default, BreakBot selects the `merge-base` commit and the `HEAD` commit as the two versions to be compared, as illustrated in Figure 5.2. Although other strategies could be implemented (*e.g.*, using the `merge-commit` or an arbitrary reference commit), this one is the closest to the maintainers' workflow as it picks the same commits as Git's three-dots diff strategy, which is the default strategy implemented in GitHub when visualizing the changes introduced by a PR.

Then, BreakBot reads the `.github/breakbot.yml` configuration file and fetches the appropriate client repositories to conduct the BCs and impact analyses. The resulting Δ-model and impact models are processed to produce a markdown report attached to

10 https://github.com/alien-tools/breakbot/

Breaking changes

Declaration	Kind	Status	Impacted clients	Broken Uses
`spoon.reflect.cu.CompilationUnit.beginOfLineIndex(int)` (diff)	`METHOD_REMOVED`	✔	None	None
`spoon.reflect.cu.CompilationUnit.getTabCount(int)` (diff)	`METHOD_REMOVED`	✔	None	None
`spoon.reflect.cu.CompilationUnit.nextLineIndex(int)` (diff)	`METHOD_REMOVED`	✔	None	None
`spoon.reflect.cu.CompilationUnit` (diff)	`ANNOTATION_DEPRECATED_ADDED`	✘	4 (KTH/xPerturb, SpoonLabs/astor, SpoonLabs/nopol, SpoonLabs/spooet)	14

(a) Reported BCs

SpoonLabs/astor

Location	Breaking declaration	Kind	Use Type
`import spoon.reflect.visitor.ImportScannerImpl;`	`spoon.reflect.visitor.ImportScannerImpl`	`SUPERCLASS_REMOVED`	`IMPORT`
`super.visitCtFieldWrite(fieldWrite)`	`spoon.reflect.visitor.ImportScannerImpl`	`SUPERCLASS_REMOVED`	`METHOD_INVOCATION`
`new spoon.reflect.visitor.ImportScannerImpl()`	`spoon.reflect.visitor.ImportScannerImpl`	`SUPERCLASS_REMOVED`	`TYPE_DEPENDENCY`
`importContext`	`spoon.reflect.visitor.ImportScanner`	`SUPERCLASS_REMOVED`	`TYPE_DEPENDENCY`

(b) Impacted code in Astor

Figure 5.3: Excerpt of the BreakBot report for Spoon's PR#3184 with the BCs and their impact on clients

Impact on clients

Client	Status	Broken Uses
STAMP-project/AssertFixer	✅	0
KTH/xPerturb	❌	3
Spirals-Team/casper	✅	0
SpoonLabs/coming	❌	88
STAMP-project/dspot	❌	13
SpoonLabs/gumtree-spoon-ast-diff	❌	3
SpoonLabs/metamutator	✅	0
SpoonLabs/astor	❌	57
SpoonLabs/nopol	❌	12
SpoonLabs/CoCoSpoon	✅	0
SpoonLabs/npefix	❌	3
SpoonLabs/spooet	❌	16
—	❌	195

Figure 5.4: Excerpt of the BreakBot report for Spoon's PR#3184 with the clients' overview

the *Checks* tab of the PR, easily accessible to maintainers during the code review process. The markdown report is enriched with clickable links that allow the library maintainers to directly navigate to the locations in the PR diff that introduce a BC, as well as to the locations in client code that are impacted by the changes.

Figure 5.3 and Figure 5.4 depict an excerpt from a BreakBot report. It displays an overview of the detected BCs together with their status (red if any client is impacted, green otherwise), the number of impacted clients, and the number of broken uses in all clients (*cf.* Figure 5.3a). Then, it summarizes the impact on each individual client (*cf.* Figure 5.4). Finally, it lists, for each impacted client, which locations are impacted by broken uses (*cf.* Figure 5.3b).

5.5 EVALUATION

In this section, we evaluate BreakBot and its support for static RDCT. Concretely, we aim at assessing both (i) its *usefulness* as the information generated by the approach and its corresponding implementation must be meaningful and insightful for library developers and; (ii) its *accuracy*, which comes hand in hand with the *usefulness* evaluation aspect: to be meaningful, results must reflect reality allowing library developers to make informed decisions. Thus, we pose the following research questions:

Q1 How useful is BreakBot for the detection of BCs impact in a pull-based development workflow?
Q2 How accurate is BreakBot in detecting the impact of BCs on clients?

Henceforth, we address each one of these research questions, discussing our corpora, methodology, main results, and analyses. We discuss the threats to the validity of both studies at the end of the section.

Usefulness Evaluation

Our objective in this evaluation is to evaluate whether static RDCT—and, in particular, its instantiation in BreakBot—is valuable to library maintainers. A privileged approach to perform this evaluation would be to conduct a field experiment [148] where a selected few library maintainers would install and configure BreakBot and use it as part of their development process for a given period of time. However, this methodology would also come with two drawbacks. First, it would hurt the generalizability of our findings. Second, within a limited time span, there is no guarantee that the variety of evolution scenarios for which BreakBot is helpful would be covered adequately.

Instead, to access a large and diverse set of evolution scenarios, we conduct an experimental simulation [148] where we harvest the sheer number of library evolution cases publicly available as PRs in popular GitHub repositories [78]. Indeed, in every PR, contributors (who may or may not be the project's maintainers) propose a set of changes to be merged into the library's code. Maintainers and reviewers evaluate the contributions as part of the (possibly tool-assisted) code review process and decide whether the changes should be incorporated.

For the sake of this evaluation, we first build a corpus of relevant PRs from popular Java libraries on GitHub (*cf.* Corpus section) and analyse them with BreakBot to compute their Δ-models and impact models on a list of popular clients. This first quantitative evaluation portrays how often and in which way selected PRs of popular Java libraries include BCs that impact their clients (*cf.* Quantitative Results section). Then, we qualitatively review the content of a selected set of PRs and the associated BreakBot reports to discuss their usefulness for maintainers (*cf.* Qualitative Exploratory Results section).

In summary, we evaluate usefulness along two main dimensions: (i) *applicability*: is it common for PRs to introduce BCs and, when they do, do they have an impact? And (ii) *relevance*: when a PR introduces BCs, does BreakBot provide valuable information to assist the maintainers' decision?

Table 5.1: Selection criteria for GitHub repositories and PRs

Criteria	**Repository**
Language	Java
Build system	Maven
Buildable	✓
Stars	$\geqslant$ 500
Clients	$\geqslant$ 100
Last push	$\leqslant$ 1 month
Last PR merge	$\leqslant$ 1 month
Fork	✗
Mirror	✗
Empty	✗
Archived	✗
Disabled	✗
Locked	✗
Criteria	**Pull Request**
Updated	$\leqslant$ 1 month
Closed	✗
Merged	✗
Locked	✗
Impacted Java files	$\geqslant$ 1

CORPUS To assemble our corpus, we query the GitHub GraphQL API to search for relevant PRs recently created or updated on popular and active Java libraries. Table 5.1 details the criteria we use while considering the characteristics of the repositories in GitHub (*e.g.*, a repository is not necessarily a project, projects tend to be inactive) as described by Kalliamvakou et al. [78].

On the one hand, a repository is relevant for our study if it uses Java as a programming language, contains a Maven Project Object Model (POM) file in its root folder (*is a project* and *is buildable*), has at least 500 stars (*is popular*) and 100 clients (*is a library*), and has been updated in the past month (*is active*). We pick a high minimum number of 500 stars for two reasons. First, the analysis would not be realistically possible for every library on GitHub, so we must pick a subset of all libraries.

Table 5.2: Descriptive statistics of the 230 studied repositories

Dimension	Min	Q1	Median	Mean	Q3	Max
Age (months)	34	75	114	118.3	149	259
Commits	159	2,323	4,561	8,325	10,367	61,716
Releases	0	0	9	76.8	40	4,883
Stars	512	1,186	2,973	6,529	7,881	38,184
Clients	127	1,589	3,272	26,750	22,544	1,380,374
Analysed PRs	1	2	5	16.5	11	390
Breaking PRs	1	1	2	4.6	5	40
Impactful PRs	1	1	1	2.2	3	11

Second, we hypothesize that libraries with a high number of stars and clients are more likely to implement a proper code review process and to care about their clients. Library clients are extracted from the GitHub Dependency Graph, considering every package offered in the repository. Packages and their dependencies on other packages are identified via manifest files (*e.g.*, POM files) or lock files hosted on the repository [77]. Repositories with less than 100 clients per package are still considered as long as the total number of clients for all packages is above this threshold. Finally, we do not consider repositories that are forks, mirrors, empty, archived, disabled, or locked. We ran the query on the 25th of November, 2022, and obtained a total of 230 repositories. Among these are well-known libraries such as `google/guava`, `apache/commons-lang`, `apache/activemq`, and many other `apache` libraries, `spring` libraries, as well as lesser-known libraries. Table 5.2 presents some descriptive statistics of the resulting corpus of libraries.

From these repositories, we then retrieve relevant PRs. Our criteria are that the PR should have been created and/or updated in the past month and must be active (*i.e.*, it has not been closed, merged, or locked). We consider both PRs that are in a `DRAFT` state (*i.e.*, work in progress) and those that are not. A `DRAFT` PR is a PR labeled as "work in progress": it signals that the code is not yet ready to be reviewed and merging into any other branch is blocked. Moreover, since we are interested in PRs where a

contributor potentially introduces BCs, we filter out PRs that do not affect Java files. This new query returned a total of 3,786 PRs, which constitute our final corpus.[11]

METHOD Using BreakBot and Maracas, we analyse our corpus of PRs to perform the static RDCT via BreakBot. However, to avoid having to install BreakBot on each of the repositories, we query the GitHub API to search for interesting PRs recently created on selected Java libraries (*cf.* Corpus section). We only use Maracas API to perform this first assessment efficiently. PRs, or sometimes the main development branch, might be in an unstable state and cannot be built properly to obtain the library JARs necessary to construct the Δ-models. Whenever an impacted package cannot be built, it is skipped and the analysis proceeds on the packages that can be built. We obtain, then, for each analysed PR: the list of Java files that are modified, the list of impacted packages (which, in our case, are Maven modules), Δ-models for each impacted package (list of BCs), a list of relevant clients for each package, and the impact model for each Δ-model as a list of broken uses on these clients. We call *breaking PR* a PR that introduces at least one BC in one of the packages it impacts, and *impactful PR* a PR that introduces at least one BC that has an impact on at least one client project (*i.e.*, at least one broken use).

We consider a PR to be interesting if it is both breaking and impactful. When we find an interesting PR, we fork the original library in our own GitHub organization,[12] where BreakBot is installed, push the corresponding YAML file with BreakBot's configuration, and copy the original PRs there. BreakBot then automatically builds the reports for these PRs (just as it would have done in the original repository). These reports are later manually inspected to report on interesting cases that can inform us of the relevance of our approach and implementation. In particular, we look for comments and information in the PR associated with BCs introduction. For that, we perform a textual

11 Our dataset and software artefacts are available at the companion Zenodo repository: https://zenodo.org/record/7475823/.

12 https://github.com/breakbot-playground/

Table 5.3: Descriptive statistics of the 3,786 studied PRs

Dimension	Min	Q1	Median	Mean	Q3	Max
			All PRs			
Modified Java files	1	2	4	10.5	11	>100
Modified packages	0	1	1	2.2	2	97
Breaking changes	0	0	0	8.5	0	17,826
Analysis time (sec)	1	31	54	147.3	123	28,661

search on the PR title, description, and comments based on a set of keywords, namely *break*, *broken*, *compatible*, *backward*, and *change*. Notice that some interesting evolutionary scenarios might be disregarded, such as cases where no BC is introduced, but the library maintainer fears to impact clients.

While maintainers would typically fine-tune their own library-specific BreakBot configuration (cf. Listing 5.3), we resort to an intuitive default configuration that makes the analysis both feasible and meaningful: for each package impacted by a PR, we retrieve the list of all its clients in descending order of stars and pick the first hundred with a minimum of 5 stars per client. As we found out that a large number of repositories marked as clients in the GitHub Dependency Graph are, in fact, forks of the original library—and therefore very likely to be impacted by changes, we filter out repositories that are explicitly marked as forks. However, a few unofficial forks still slip into the analysis as they are created manually as new repositories under the umbrella of a different organization, without any explicit link to the source repository on GitHub, and cannot be filtered automatically. Nevertheless, the name of the original repository can be used to trace back some of the unofficial forks that preserved it. Such cases are removed from the corpus to reduce further impact in the final results.

QUANTITATIVE RESULTS On the one hand, Table 5.3 presents some descriptive statistics of the obtained dataset. In particular,

Table 5.4: Descriptive statistics of the breaking PRs

Dimension	Min	Q1	Median	Mean	Q3	Max
Breaking changes	1	1	3	68.8	9	17,826
Analysed clients	0	1	12	37.4	100	337
Broken clients	0	0	0	1.0	0	91
Broken uses	0	0	0	25.8	0	2693

it shows the number of modified Java files per PR[13] (median of 4), the number of modified packages per PR (median of 1), the number of introduced BCs per PR (median of 1), and the time taken by BreakBot to perform the analysis on the repository (median of 54 sec). Overall, 465 out of 3,786 PRs are breaking (12.28%), and 42 are impactful (2.19% of all PRs, 17.85% of breaking ones). Interestingly, 391 of the 3,786 PRs are in a `DRAFT` state (10.33%), yet they account for 16.56% of the breaking ones (77/465).

On the other hand, Table 5.4 depicts some descriptive statistics related to the analysis of breaking PRs. It presents the number of BCs per breaking PR (median of 3), the number of analysed clients per breaking PR (median of 12), the number of broken clients per breaking PR (median of 0), and the number of broken uses per breaking PR (median of 0). Note that, in some cases, no client is analysed even though the PR is breaking: this is because the packages impacted by the PR do not have clients that meet our criteria, even though the library does have more than 100 clients.

We draw the following lessons from this first quantitative analysis. First, there is indeed an opportunity for BreakBot to detect breaking PRs on GitHub, even when studying popular and mature libraries with many clients. Second, BreakBot finds impactful PRs even when considering a number of up to 100 clients per package, and we believe this can still be improved (*cf.*

13 The maximum number of modified Java files is labeled as >100 due to the GitHub API endpoint pagination for PR files. We would need to navigate the result pages to get the exact number of modified files, impacting the querying time. As the current study is not aiming at performing a thorough characterization of the chosen PRs, we opt for using the >100 to get a glimpse of the PRs sizes.

Section 5.6). Third, BreakBot is quite efficient in its analysis (median of 54 and mean of 147.3 seconds, which includes identifying the impacted packages, building the corresponding JAR versions, computing the Δ-models, fetching the clients, and computing the impact models). Fourth, however, the median of detected broken clients and broken uses for breaking PRs is 0. These results require further investigation to identify how to improve BreakBot's design so it can fit the library development process and not appear as a silent bot that impacts maintainers' confidence in the tool. Interestingly, these results are of the same order of magnitude (for breaking and impactful PRs) as those obtained when studying semantic versioning in the Maven ecosystem (*cf.* Chapter 3). In the next section, we conduct a more qualitative analysis of its usefulness to library maintainers.

QUALITATIVE EXPLORATORY RESULTS In this section, we perform a qualitative exploratory study to get a glimpse of the potential relevance of BreakBot for library maintainers. We analyse the reports ourselves, and in some pertinent cases, we leave a comment on the PR asking for feedback from PR reviewers. The comment contains a link to the fork of the library in our GitHub organization—which contains the BreakBot report—and a link to a 5-minutes pilot survey complying with GDPR and approved by the ethical board of the Eindhoven University of Technology (TU/e).

The survey contains questions about the role of the respondent in the project, the project context, the selected PR and BreakBot's report, and interest in participating in a follow-up study (*cf.* Appendix A). The pilot survey faced a low response rate (just a handful of people replied) and a mixed reception from maintainers, prompting us not to distribute it further. Indeed, we realized that our comments were interfering with the maintainers' workflow of reviewing PRs—although we carefully selected the PRs, manually reviewed the content of every report, and limited our comments to one per repository. Concisely, some maintainers considered the comment disturbing as it was unsolicited and unexpected, and in some cases they found it irrelevant, arguing

that the nature of the introduced changes was already identified by them. Still, we received interesting comments from the respondents, which we use as additional insights to complement our own manual inspection. The reception of some library maintainers led us to stop the study after the publication of 15 comments and a total of 3 responses. Therefore, we disclaim that these results are preliminary and exploratory; by no means they should be considered conclusive. Additional research needs to be conducted to draw conclusions on the usefulness of our tool.

After manually reviewing the reports, we identify four evolution scenarios, namely (i) introduction of deprecations anticipating broken client code; (ii) introduction of BCs impacting both internal and external clients; (iii) introduction of unexpected BCs, and; (iv) introduction of non-impactful BCs. This list of evolution scenarios is by no means complete. We briefly describe the aforementioned scenarios and present some real examples with pointers to the BreakBot reports.

Scenario I: Introduction of deprecations anticipating broken client code. The Java `@Deprecated` annotation (or the corresponding `@deprecated` Javadoc tag introduced in Java 1.1) is used to discourage the use of some part of the API. This is done primarily because it might be insecure, inefficient, buggy, or obsolete, and will likely be considered for future removal [57]. Several BreakBot reports announced the forthcoming impact of removing such elements on client code. To illustrate, PR#9523 in `trinodb/trino` deprecated the `getTableProperties()` method in interface `io.trino.spi.connector.Connector`. Four of the analysed client projects implement this interface. BreakBot accurately reports the impact on these clients should this method be removed in the future.[14]

Scenario II: Introduction of BCs impacting both internal and external clients. BreakBot helped identify impacted internal and external clients. An internal client is a repository hosted within the same GitHub organization as the library, while an external one is owned by a different one. For in-

14 https://web.archive.org/web/https://github.com/breakbot-playground/trino/pull/4/checks/

stance, PR#562 from `spring-cloud/spring-cloud-commons` removed a method from a public class. The change impacted an internal client `spring-cloud/spring-cloud-consul`—within the same organization—which was overriding the method. Library maintainers later reported failing tests for this client. BreakBot detected such a scenario and linked the BCs introduced in the library with the impacted code in the client, providing traceability between the change and the subsequent issue.[15]

Scenario III: Introduction of unexpected BCs. BreakBot can bring awareness when unexpected BCs are introduced. One of our survey respondents stated that even though "*[the analysed] pull request is against a new major version [anonymised] and [they] expect to introduce more breaking API changes*", they underestimated the impact of such a PR on client code. Moreover, as an additional example, project `apache/flink` uses a PR template that explicitly asks contributors whether introduced changes potentially affect the public API of classes annotated with `@Public(Evolving)`. PR#19986 removed three public constants from the public API. Although the owning class was not labeled with the `@Public(Evolving)` annotation, the change impacted four client projects, as stated in the BreakBot report.[16] Defining what the public API is is up to the library developers, but these changes might be of interest to the maintainers when reviewing the PR.

Scenario IV: Introduction of non-impactful BCs. As reported before, 17.85% of breaking PRs impact client code, meaning that most breaking PRs are harmless for their clients. Furthermore, some impactful PRs can even have different types of BCs from which a subset impacts client code, and the other does not. This information can be relevant for library maintainers to corroborate their decisions regarding the acceptance of certain changes in their repositories' PRs.

15 https://web.archive.org/web/https://github.com/breakbot-playground/spring-cloud-commons/pull/1/checks/

16 https://web.archive.org/web/https://github.com/breakbot-playground/flink/pull/6/checks/

As final remarks, to improve BreakBot's output, library maintainers who responded to our survey and/or PR comments and that are concerned about BCs impact on client code suggested that deprecations must be treated differently to how BCs are reported: "*What [a deprecated annotation] does is signal that the code should no longer be used, and that migrating away from it is required for future compatibility*." When both kinds of changes are reported at the same level, the information might be interpreted as misleading: "*I'm not sure if I agree with the signals* BreakBot *is raising. We did deprecate a factory method but the behavior is still consistent*." Respondents' reactions also suggest that maintainers need a dedicated fine-tuned configuration for their libraries, rather than the generic one we used in this evaluation, for instance regarding deprecation, the development version against which the PRs should be checked, and the clients to analyse.

ANALYSIS In summary, this first exploratory analysis shows that BreakBot can successfully identify interesting evolution scenarios on GitHub and applies to many libraries. Concretely, 12.18% of the studied PRs are breaking, but only 2.19% are impactful (17.85% of the breaking PRs). Given these non-negligible figures, we conclude that PRs do introduce BCs that might be impactful. This problem can, therefore, be addressed via an approach such as static RDCT. Moreover, from the manual inspection of BreakBot reports, we spot a list of four evolution scenarios that give us early evidence of the applicability and relevance of BreakBot.

Q1. How useful is BreakBot for the detection of BCs impact in a pull-based development workflow?
12.18% of the studied PRs are breaking but only 2.19% are impactful (17.85% of the breaking PRs). Although BreakBot can identify these cases and provide insightful information to library maintainers (as shown in different evolution scenarios), more research needs to be conducted to prove how useful it actually is and how valuable maintainers find it. The analysis of the reports

generated on breaking PRs highlights that BreakBot can provide valuable information for various evolution scenarios, which might complement the maintainers' expertise.

THREATS TO VALIDITY In this section, we identify threats that can impact the internal, external, and construct validity of the usefulness evaluation of our approach. Regarding the *internal validity* of the study, we measure applicability in terms of the number of breaking and impactful PRs. Although these figures let us know that such PRs do exist on Java repositories hosted on GitHub, they are just a proxy to sense the applicability of the approach in a pull-based development workflow. This information can only accurately be obtained by directly asking for the library maintainers' opinion. In the future, the relevance of the approach should be directly assessed with its potential users—that is, once again, library maintainers.

The *external validity* of the study is threatened by the criteria used in the usefulness evaluation; *e.g.*, selecting GitHub as an ecosystem and considering only Java projects that are public, buildable, popular, and active. These criteria do not allow us to generalize our results to other ecosystems, programming languages, or projects with divergent characteristics. Moreover, the manual selection of interesting reports is arbitrary and introduces a selection bias: only interesting cases might be included, and irrelevant reports are kept aside. The selected reports, thus, should be used only as a qualitative indicator that illustrates the figures presented in the first part of the assessment.

Lastly, the *construct validity* of the study is threatened by the existing issues and bugs of the underlying tools (*e.g.*, GitHub querying tools and Maracas). To reduce such threats, we conduct an accuracy evaluation presented in the *Accuracy Evaluation* section. Undetected issues introduced in our dataset-gathering pipelines can also impact the validity of the study. We performed unit tests and exploratory data analysis to mitigate this threat.

Accuracy Evaluation

To answer **Q2**, we need to evaluate the accuracy of MARACAS, the static impact analyser implementation of our solution and, therefore, the component in charge of detecting BCs between two snapshots of a library together with their impact on client code. We opt to frame the evaluation as a *laboratory experiment* [148], which allows us to study, with high precision, whether MARACAS succeeds at identifying impacted and non-impacted client code in well-defined API evolution cases. An API evolution case consists of a BC introduced in an API member (*i.e.*, type, method, or field) and a use of such a member in client code. The latter can be used in a breaking or non-breaking manner on the client side. In concrete, we conduct a *benchmark* study that considers two synthetic corpora.

CORPORA Our accuracy evaluation considers two synthetic corpora, namely the `comp-changes` corpus developed by the authors of this chapter, and the *API evolution data* corpus, designed and implemented by Jezek and Dietrich [70]. We describe both corpora below. (In the following text, consider l_1, l_2, and c defined in Section 5.3.)

`comp-changes` corpus is a set of synthetic projects consisting of three main Java projects, namely `old`, `new`, and `client`. The former two represent the old and new releases of a synthetic library (*i.e.*, l_1 and l_2, respectively) that has been designed with the sole purpose of introducing BCs. The library is split into a set of packages, each addressing a different BC. In total, the library covers 41 types of BCs as defined by the `japicmp`[17] project, which in turn is aligned with the JLS [57]. Moreover, each package contains a set of classes representing different scenarios where BCs manifest. The newest release of the library, `new`, is in charge of introducing the corresponding BCs. The `client` project (*i.e.*, c) depends on the oldest release of the library, `old`. This is the project that might be impacted (or not) by the BCs once it upgrades to

17 https://siom79.github.io/japicmp/

the newest release of the library. To date, the corpus reports 378 API evolution cases.

API evolution data corpus is a synthetic data set introduced by Jezek and Dietrich to compare tools in charge of identifying BCs introduction (*e.g.*, `japicmp`, `clirr`). The corpus also consists of three projects, namely `lib-v1`, `lib-v2`, and `client`. `lib-v1` is the baseline release of a library, and `lib-v2` is its modified version (*i.e.*, l_1 and l_2, respectively). The `client` project (*i.e.*, c) is an executable application that depends on and uses the first release of the library. The data set revolves around three dimensions that aim to cover possible API evolution cases: (i) *what* has changed (*e.g.*, access modifier, type); (ii) *where* has it changed (*e.g.*, in a class, in a method), and; (iii) *how* the code has changed. The latter dimension depends on the combination of the two previous ones, but, in general, a code element can be added, removed, or modified. We add some modifications to the original corpus, specifically: (i) we transform the three projects into Maven projects for later validation; (ii) we remove deprecated code to avoid noisy compilation messages. In particular, we replace the deprecated use of the `Integer` constructor with the static method `valueOf` of the `Integer` class, and; (iii) we add a missing set of cases addressing fields visibility increase. In the end, the corpus results in 186 API evolution cases.

METHOD To answer **Q2**, we set two benchmarks based on the above-mentioned synthetic corpora, namely the `comp-changes` and the *API evolution data* corpora. Each one simulates the introduction of diverse BCs by defining a set of API evolution cases. These cases are used as ground truth to compute the accuracy of MARACAS.

We developed a component in MARACAS to evaluate the tool's accuracy, namely the `validator` project. The project takes two releases of a library (l_1 and l_2) and a client project (c) that depends on the oldest release of the library (l_1). The three projects are Maven projects, and therefore, the dependency of c on l_1 is explicitly stated on the POM file. Then, MARACAS is used to compute the Δ-model between l_1 and l_2, and the impact model

of c. The latter contains a set of broken uses caused by BCs introduction. It is, then, the target of our accuracy evaluation: we aim at tracking true positive (TP), false positive (FP), and false negative (FN) uses. With these measures, we compute the accuracy of Maracas in terms of precision and recall. *Precision* refers to the ratio of retrieved cases that are indeed relevant (*cf.* Equation 5.1). Conversely, *recall* refers to the ratio of relevant cases that are retrieved (*cf.* Equation 5.2).

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{5.1}$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{5.2}$$

However, to compute Maracas accuracy, we need to compare Maracas output against a ground truth set. To do so, the `validator` project is in charge of automatically upgrading the dependency of the client project c to the library release l_2. This is achieved by injecting the new version of the library dependency in the POM file of the client project. Then, the standard Maven compiler (`javax.tools.JavaCompiler`) is used to try to build the client project using Java 8 and gather the whole list of compilation errors or warnings obtained after upgrading the library version. This list of compilation messages is treated then as the ground truth set. Finally, to verify Maracas accuracy, all compilation messages related to the upgrade are matched against the set of broken uses reported by Maracas. The matching is performed based on the location of the client's affected element in the broken use and the location reported by the compilation message. For the location, we consider both the path to the file and the source code line. Matched pairs of broken uses and compilation messages are labeled as TP cases; unmatched broken uses are labeled as FP cases, and; unmatched compilation messages are labeled as FN cases. This project can be extended to consider other corpora, build systems, and matching criteria.

After performing the automatic matching of the compilation messages against the broken uses reported by Maracas for both

Table 5.5: MARACAS accuracy measures and metrics

Benchmark	Cases					Metrics	
	TP	FP	FN	**Disc.**	**Total**	**Precision**	**Recall**
`comp-changes`	356	13	19	11	388	0.96	0.95
API evolution data	170	15	1	12	186	0.92	0.99

corpora, we manually validate the results. Concretely, we go through all FP and FN cases and verify if these cases need to be reclassified or discarded. FP are reclassified as TP cases when the Maven compiler does not report an expected broken use in the form of a compilation message. Possible cases include illegal casting, unhandled checked exceptions, references to removed class members, and references to members of deprecated transitive supertypes. For instance, when invoking a method via an object of a removed class. In like manner, FN cases can be discarded if they address cases not meant to be handled by MARACAS. In particular, we discard cases triggered by unsupported BCs, redundant compilation messages pointing to two different locations (*e.g.*, removing a method can trigger a compilation message at the class declaration and the overriding method), and errors triggered by accessing a static field via an object. After this semi-automatic classification, the number of TP, FP, and FN cases are used to calculate the precision and recall for each benchmark.

RESULTS Table 5.5[18] summarizes the results obtained for MARACAS accuracy evaluation. The table shows results for both the `comp-changes` and the *API evolution data* benchmarks. Afterwards, the number of TP, FP, and FN cases are reported, together with the number of discarded cases after manual reclassification and the total number of disclosed cases. The precision and recall of the evaluation are also included for each benchmark.

On the one hand, the `comp-changes` benchmark outputs a total of 388 cases, out of which 356 are TP cases, 13 are FP cases, 19

18 "Disc." stands for "Discarded".

are FN cases, 26 cases are manually reclassified, and 11 cases are discarded. Based on these cases, MARACAS reports a precision of 0.96 and a recall of 0.95. On the other hand, the *API evolution data* benchmark reports 186 cases, out of which 170 are TP cases (16 more than before), 15 are FP cases (16 less than before), 1 is a FN case (12 cases less than before), 16 cases are manually reclassified, and 12 cases are discarded. MARACAS reports a precision of 0.92 and a recall of 0.99.

Lastly, after performing the manual verification of the evaluation, we find out that both FP and FN cases are triggered for the following reasons:

GENERICS. `japicmp` does not report BCs related to generics. This is due to type erasure, which removes the type parameters information from the source code to produce a binary file that is backward compatible with previous versions of Java [125].

INHERITANCE HIERARCHY. MARACAS only has access to information in the library releases and the client projects. Thus, it can only build part of the inheritance hierarchy for some types.

THE `strictfp` AND `native` MODIFIERS. `japicmp` does not report BCs related to the `strictfp` and `native` modifiers [70]. The `strictfp` modifier is used to restrict floating-point calculations and ensure the same result on every platform. The `native` modifier informs that the associated method is implemented in native code using the Java Native Interface (JNI).

(UN)BOXING TYPES. `japicmp` does not always report the new type of a construct when a primitive value is converted to its corresponding object wrapper class (boxing) or vice versa (unboxing). This is the case of a change in the type of a method parameter. `japicmp` reports such a case as a method removal, losing information about the new types of the parameters.

Although reflection has not been included in the evaluation, we are aware that it is not considered by MARACAS during analysis. That is, the use of reflection can result into broken uses that will pass undetected by the tool.

ANALYSIS After performing the accuracy evaluation on both benchmarks, we observe that MARACAS scores more than 0.92 in precision (0.96 and 0.92 for the `comp-changes` and *API evolution data* benchmarks, respectively) and recall (0.95 and 0.99 for the `comp-changes` and *API evolution data* benchmarks, respectively). FP and FN cases are due to the loss of information when dealing with binary code (*i.e.*, *generics* and *inheritance hierarchy*), and limitations coming from underlying tools (*i.e.*, lack of information about `strictfp` *and* `native` *modifiers* and *(un)boxing types*). These results give us confidence that (i) noise or misleading information is seldom reported on MARACAS models, and (ii) few cases slip unnoticed by the tool. Nevertheless, some important threats to validity must be considered in future research. Such threats are further described in the next section.

Q2. How accurate is BreakBot in detecting the impact of BCs on clients?
MARACAS reports a precision of 0.96 and a recall of 0.95 on our synthetic `comp-changes` benchmark, and a precision of 0.92 and a recall of 0.99 on the *API evolution data* one. In both benchmarks, the accuracy metrics score more than 0.9.

THREATS TO VALIDITY Some threats might impact the internal, external, and construct validity of our accuracy evaluation. First, *internal validity* might be impacted as we do not control the implementation decisions of the Maven compiler, which is used to generate the ground truth of the study. In particular, the expected behaviour might differ from the one showcased by the compiler. To cope with this threat, we manually inspect the FP and FN cases. Based on the expected output, we reclassify incorrect FP cases as TP cases and discard the incorrect FN

from the analysis. Nevertheless, this triggers an additional internal threat: reclassification is a manual process and is, therefore, error-prone.

Second, the *external validity* of the study is threatened by our decision to report results only for Java 8, and to choose synthetic corpora that, due to its design and implementation, might favor the accuracy test of MARACAS. Results cannot be blindly generalized to other Java projects, versions of Java, and languages. Lastly, *construct validity* can be impacted by bugs introduced in the `validator` project, particularly when performing the broken uses and compiler messages matching. To decrease the effects of this threat, we manually inspect the FP and FN cases to verify that the output was correct. However, TP cases are disregarded from this process.

5.6 DISCUSSION

The desire to preserve stability and to prioritize the sense of community are generalized concerns among all projects in most ecosystems [18]. Therefore, deciding whether and when to introduce BCs is no trifling matter. The decision depends on values and constraints established at the ecosystem and library levels. To enact stability, libraries tend to assume that BCs are always harmful [68]. This reasoning has led to the misconception that stability is equivalent to change stagnation [18, 21, 109]. However, we claim that stability is the capability to not break clients even under the presence of change. What truly affects stability are *changes that break clients*, not BCs themselves. This distinction is essential: it has been shown that, for instance, in MCR, most BCs do not impact the clients whatsoever (*cf.* Chapter 3). In this chapter, we confirm that this is also the case for BCs in PRs of popular Java libraries on GitHub.

But, what if introduced BCs do impact client code? Do modifications need to be reverted? Once again, the decision will depend on the library's values: if innovation and rapid access are valued, both breaking and non-breaking changes are more likely to slip into new releases of the library; whilst, if stability and

compatibility are values appreciated by the library maintainers, chances to revert modifications that potentially break client code increase [18]. If such changes find their way in a new library release, evidence about how changes impact clients should be communicated, or else the mistrust in the community raises.

Some evolving libraries opt for communicating change via their API, versioning conventions such as semantic versioning, or official documentation such as release notes and changelogs. These artifacts are seen as promises that, when broken, result in further mistrust. Unfortunately, tooling to support the generation of such artifacts is lacking, forcing library maintainers to define them manually, and making them error-prone. We foresee that shared tooling is required to enforce or encourage the announcement of changes and their actual impact [18]. As our study suggests, the static RDCT approach and BreakBot can be valuable when plugged into a library development workflow. However, further support can be offered to (i) improve the selection of client samples to assess the impact of BCs in client code; (ii) support the configurability of static RDCT approaches; (iii) praise the community value by using the reports to define client upgrade policies and BCs-coping strategies, and; (iv) improve the generation of existing BC communication channels (*e.g.*, semantic versioning, release notes).

DISCOVERING CLIENTS Knowing how BCs impact relevant clients is key when deciding to apply or revert changes. Some projects, such as Coq and Spoon, have a precise list of relevant clients, but this is only the case for a few libraries. Even when such a list exists, it might be incomplete and hide important evidence. Automatically discovering a *diverse and representative sample* [119] of clients should give a clearer picture of the library's usage. These capabilities must be integrated into the *client explorer* component of the static RDCT approach (*cf.* Section 5.3). To do so, one must first pinpoint which versions of the library are affected by a given BC, and thus which clients may be affected. As it is likely that clients' usage of a library will overlap [64], and to provide an efficient solution, it is essential to avoid analysing

similar clients multiple times. It is thus needed to find a sample of clients that represent the overall usage of the API, *e.g.*, by clustering clients around their use of the library.

CONFIGURABILITY Policies to deal with backward compatibility range from permissive (*e.g.*, npm, favoring *innovation*) to restrictive (*e.g.*, Eclipse, favoring *compatibility*) [18]. Regardless of their posture, projects should be given the necessary information to decide whether and how to introduce a change. Bots such as BREAKBOT must therefore be highly configurable to account for project-specific policies [51, 157]. *What are*, then, the configuration properties that impact analysis tools should capture? For instance, defining the type and number of allowed BC, the parts of the API subject to analysis (*e.g.*, non-experimental interfaces), and the key set of relevant clients (*e.g.*, commercially-related projects, popular projects), is of foremost importance.

UPGRADE POLICIES AND STRATEGIES BREAKBOT generates reports for library maintainers that pinpoint clients that will be impacted. This information can be leveraged to define upgrade policies and BCs coping strategies on the client side. To do so, the library and its community must decide how to distribute the upgrade costs associated with the library's evolution. A caring library can, for instance, provide documentation that helps clients upgrade their impacted code, or patch their clients themselves—as the Coq project does. On the contrary, a library that prioritizes time to market might push the upgrade effort to its client projects: client developers need to find ways to patch the broken code and share such solutions in adequate community channels.

COMMUNICATION CHANNELS When a new library version is released to the public, clients may consider announcement mechanisms such as the semantic version number, annotations, and naming conventions to pinpoint unstable releases or foresee the introduction of BCs. They can also browse through the associated *release notes* or *changelogs* to learn about new features,

BCs, or migration guides. Unfortunately, this information is not always available, and if present, it might be erroneous and, therefore, misleading. For instance, Wu et al. identify that the absence of information related to BCs is the prime complaint of users when dealing with release notes [161]. Specifically, the authors note that "*it can be difficult for release notes producers to correctly locate and highlight breaking changes*". Tools that support previous manual mechanisms, as well as the generation of community artefacts, can be valued by developers. For instance, creating tooling that suggests a semantic version number [90]; identifies unstable parts of the library API, and; produces accurate and relevant release notes and changelogs might be of great use. We believe that BreakBot and Maracas provide such capabilities and should be tuned in the future to support, for instance, the automatic generation of release notes in new or existing tools such as ARENA [115] and DeepRelease [75].

5.7 Related work

API evolution and BCs have been the subject of active research for a long time [18]. Multiple papers have studied why and how evolution manifests at the level of individual libraries and entire ecosystems [12, 21, 34, 66, 122, 164]. BCs are largely considered a pain point for library users who must keep up to avoid technical lag and for library maintainers who must be cautious when introducing them [18, 170].

In a recent and extensive review of the literature, Lamothe, Guéhéneuc, and Shang identify unsolved challenges in API evolution, some of which are particularly relevant to our work: "*Tools that mine usage data help API developers improve APIs*", "*Tools to help API developers deal with API migration, not just users*", and "*Determining API migration and API recommendation impacts*" [91]. We see static RDCT, and its implementation in BreakBot, as first steps towards addressing these challenges.

Over time, many tools have been developed to help maintainers and users identify BCs in new library versions. We decided to base Maracas on `japicmp`, which has been shown to be the best-

performing tool for BCs detection in Java libraries [70]. Although we believe it is equally important to identify the impact of these BCs in client code, there is little support in the Java ecosystem to support this analysis [90]. Our tool Maracas is intended to fill this gap, similar to work achieved in other ecosystems. For instance, Møller, Nielsen, and Torp developed Tapir to identify locations in JavaScript code potentially affected by BCs [114], and Coccinelle has been used for more than a decade to automatically co-evolve source code in the Linux kernel [93].

There has been significant work in dealing with API BCs through automatic API migration [45, 92, 120, 168, 169]. While these approaches are most valuable, we attack the problem from a different angle. We claim that API evolution could lead to less friction if API designers and maintainers had the means to understand and anticipate the impact of their changes. We see the two approaches as complementary: getting a better understanding of the impact of API changes should lead to fewer BCs and help to improve the accuracy of automatic migration tools leveraging factual usage data extracted from tools such as BreakBot.

5.8 conclusion

In this chapter, we introduce the static RDCT approach, together with BreakBot, its implementation for Java projects hosted on GitHub. The static RDCT approach aims to assist maintainers in the evolution of their libraries by providing information about how the changes they introduce impact client code. Specifically, its implementation in BreakBot reports introduced syntactic BCs on GitHub's PRs and their impact on a subset of client projects. This information is fed back to the development workflow of the library maintainers for further analysis and assistance in code review.

We evaluate static RDCT and BreakBot in terms of usefulness and accuracy. Analysing thousands of PRs on popular and active Java libraries hosted on GitHub, we identify BCs and their impact on a sample of their most popular clients also hosted on GitHub. We conclude that BreakBot identifies many breaking PRs, some

of which impact clients, and produces meaningful reports for a subset of the library's clients. However, further research needs to be conducted to asses BreakBot's potential value in real contexts. Second, the accuracy evaluation performed on two synthetic benchmarks (*i.e.*, `comp-changes` and *API evolution data*) reports a precision of 0.96 and a recall of 0.95 on the `comp-changes` benchmark, and a precision of 0.92 and a recall of 0.99 on the *API evolution data* benchmark. We expect our framework and its implementation in BreakBot to empower library maintainers, allowing them to fight technical debt and have a purposeful picture of the nature of introduced changes in PRs and the impact they have on selected clients.

Part IV

TODO CAMBIA

Cambia lo superficial
Cambia también lo profundo
Cambia el modo de pensar
Cambia todo en este mundo

Cambia el clima con los años
Cambia el pastor su rebaño
Y así como todo cambia
Que yo cambie no es extraño

– Julio Numhauser (1982)

6

CONCLUSION

In this thesis, we address the library-client co-evolution problem. First, we consider the *nounal* view to understand the nature of the phenomenon. Concretely, we investigate (i) best practices to define dependencies to prevent the propagation of BCs, and; (ii) BCs introduction with regards to semantic versioning practices and their real impact on client projects. Second, we considered the *verbal* view to come up with new processes, methods, and tools that can better support the processes associated with the library-client co-evolution phenomenon. To this aim, we introduce (i) Maracas a static analysis tool that implements our *static impact analysis* approach. The latter aims at statically detecting BCs between two versions of a Java library and their impact on client code, and; (ii) BreakBot, a GitHub bot that assists library evolution by reporting insights into backwards compatibility and impact analysis on a list of relevant clients. BreakBot is a prototype that implements the *static RDCT* approach. In the rest of the chapter, we report on the main findings of the thesis as a whole (*cf.* Section 6.1). We close up the thesis by describing future research directions (*cf.* Section 6.2).

6.1 MAIN FINDINGS

In this chapter, we present the main conclusions to answer the three main research questions introduced in Section 1.3. **Q1** addresses the nounal view as its study gives us insights into how experts manage dependencies and to what extent such practices

are followed by Eclipse developers. **Q2** addresses both the nounal and the verbal view as the first version of MARACAS was implemented to conduct the empirical study on BCs impact in Maven. Lastly, **Q3** focuses on the verbal view by providing a method and tool to assist library evolution.

Research Question 1: Dependency Management Best Practices

We revisit **Q1**, which is addressed in Chapter 2.

Q1: What dependency management best practices are advised and followed and what observable effect do they have on software projects?

As main findings we discover that experts promote the adoption of 11 best practices including exposing only the public API and hiding implementation details to clients, defining all required dependencies (even if they seem to be available by default during development), explicitly using dependency versions and versioning conventions such as semantic versioning, among others. Six out of 11 best practices are further studied to evaluate if they are being used in the Eclipse ecosystem and their observable effect on such projects. We find out that most of the selected best practices are not widely followed. Moreover, one-third of the studied best practices, reduce the classpath size of the software projects, and have no statistically significant impact on their resolution time.

We conclude that advised dependency management best practices enforce the specification of robust and efficient APIs. In particular, experts suggest hiding implementation details, exposing and using only needed interfaces, and always using versioning mechanisms. The main goal of applying these best practices is to prevent the unintended propagation of BCs into client code.

Research Question 2: Semantic Versioning, Breaking Changes & Impact Analysis

We revisit **Q2**, which is explored in Chapter 3.

Q2: What is the real impact of BCs on clients?

Our results show that more than 83.4% of library upgrades comply with semantic versioning principles—that is, that libraries introduce syntactic BCs only in the expected releases (*i.e.*, major and initial development releases). Furthermore, the tendency to comply with semantic versioning principles has increased over time. Lastly, we discover that code introducing BCs is seldom used by any client in the studied ecosystem. Actually, less than 8% of all clients are impacted by BCs. In conclusion, we find that library and client projects (in Maven) are not "breaking bad", meaning that (i) library maintainers tend, to inform about the introduction of syntactic BCs via versioning mechanisms, and; (ii) the real impact of these changes on client code represents less than 10% of the potential impact. These findings increase the confidence on supporting an informed client-library co-evolution, where both technical debt and lag are reduced.

Research Question 3: Library Evolution Assistance

Finally, we revisit **Q3** addressed in Chapter 4 and Chapter 5.

Q3: How to assist library evolution?

We introduce the *static RDCT* approach to assist library maintainers with the evolution of their projects, and its implementation, BreakBot. BreakBot is a GitHub bot used in pull-based development Java repositories that depends on a set of static analyses tool (*e.g.*, Maracas, `japicmp`, Spoon) to provide insightful information. After installing the bot on a fork of the Spoon framework, we can validate the main features of the tool: (i) BreakBot

does not need to build client projects, instead, it parses client code and performs a static analysis on to of the generated ASTs, and; (ii) BreakBot summarises essential BCs impact information in a single report. Our evaluation reports that Maracas—BreakBot's underlying tool—has a precision and recall of more than 90% on synthetic benchmarks. We also identify that BreakBot, and, therefore, the static RDCT approach, is applicable in a pull-based development workflow. A first exploratory qualitative study suggests that BreakBot can be relevant for library maintainers when addressing diverse evolution scenarios.

6.2 Future Research Directions

To discuss the future research directions derived from this thesis, we consider our two methodological views (*i.e.*, the nounal and verbal view). Regarding the *nounal* view, we focus on studying the phenomenon of library-client co-evolution. As for the *verbal* view, we dive deeper into methods and tools that provide new ways of supporting processes related to the library-client co-evolution phenomenon.

Nounal View Future Research

In relation to the nounal view, we first consider research on *dependency management*. In particular, we aim at better understanding how poorly-managed dependencies impact *BCs propagation* in software ecosystems. Are clients more prone to be impacted by BCs in these scenarios? Are there design decisions that shield client code from being impacted by such changes? We also plan to further assess how specific ways of defining dependencies among software projects has an *impact on diverse extra-functional attributes* of an ecosystem. We shift from a client-library co-evolution scenario to an ecosystem evolution one. For instance, are software ecosystems whose projects follow certain dependency management practices more reliable, usable, efficient or maintainable [52]?

How do these practices affect the design of the APIs within such ecosystems?

Second, we intend to extend our research on BCs introduction and their impact analysis. Concretely, we would like to focus on better understanding the *why* of BCs introduction. Studying the *when* and *where* of BCs is a first step towards this goal (*cf.* Chapter 3). However, additional qualitative studies are required to understand the motivation and management of library evolution. How do developers cope with BCs? What are developers concerns when evolving a library?

Additionally, we plan to study how *programming languages design and their evolution* impact the *definition of syntactic BCs*. As new constructs are introduced in a language (*e.g.*, the `default` operator in Java 8 or the `record` data type in Java 15) new BCs appear. However, these new constructs can also be introduced as a way of (i) coping with backwards compatibility (*e.g.*, `default` methods support the evolution of interfaces), or; (ii) supporting the design of robust APIs (*e.g.*, the `module` construct was introduced in Java 9 as a way to cope with the limitations of access modifiers). The automatic identification of such BCs and its alignment with static analysis tools is thus essential to provide insightful evidence to the programming language users.

Another focal point for future research lays on the study of *clients reaction to BCs*. At the dependency definition level, we would like to explore how clients deal with the introduction of new releases of their libraries. Some studies on technical lag and outdated dependency management [36, 86, 87, 136] are stepping stones into reaching this understanding. We aim to study how clients react in the wild and which patterns can be identified from these reactions. How do clients react to specific BCs? Are there patterns that we can associate to specific instances of BCs? If we take a step back, we can start by studying how to isolate code modifications related to a specific BC [107], and, even further, how to perform an automatic migration or repair the code [106]. The answer to this question can contribute not only to the field of impact analysis and software upgrading, but also to other

fields where the isolation of data and control dependent code is required (*e.g.*, finding bugs, testing).

Finally, we would like to go beyond syntactic compatibility, and study *semantic compatibility*. Semantic incompatibility does not generate any compilation or linking-time error—at least in Java. Not to mention that the difficulty to identify this type of incompatibility originates from the problem of *program equivalence*, which is undecidable [54]. Hence, detection of semantic BCs is not trivial and in some scenarios can even be untraceable. Particularly, we want to understand *how to accurately identify the introduction of semantic BCs*, and *how to efficiently detect the impact of such BCs on client projects*. A straight-forward approach to detect semantic changes relies on using the tests suites of different software projects to identify the changes in behaviour of a library. For instance, Danglot et al. propose an approach to detect semantic changes (also called *behavioural* changes in the literature) on commits by generating a set of test methods derived from variations of existing test cases [30]. Similarly, Chen et al. and Mujahid et al. use the test suites of several clients of a library to detect the introduction of such changes [26, 117]. We foresee that achieving a good test coverage of the library and basing the identification of semantic BC on the library tests is a potential scalable solution. In particular, the tests of the old library release can be used to verify the output of the new release under the same conditions and inputs. Moreover, finding out what is the source that originates such changes and the meaning of the change is relevant information for the understanding of both library and client developers. Testing techniques such as mutation and random testing [160] would be beneficial to provide a robust approach.

Verbal View Future Research

Regarding the verbal view, we aim first at improving the design and capabilities of the methods and tools that we introduce in the thesis. In particular, to support the different policies and values of software ecosystem and projects [18], we need to extend the *configurability* of tools such as BreakBot. Knowing the configura-

tion properties required by library maintainers to perform static impact analysis is thus needed. For example, library maintainers can be interested in defining the number and type of allowed BCs in a commit, the parts of the interface they care about when facing software evolution, as well as the list of clients that are relevant to them. The latter aspect is of foremost importance: the *automatic discovery of relevant clients*. As a first goal, we aim at gathering a *diverse and representative sample* [119] of clients of a library. It is, however, important to avoid analyzing several times clients that use the library in a similar fashion [64]. For that, investigating how to represent the API usage print of a client is not trivial and essential to generate a representative set of clients. What do we understand by *similar API usage*? Which properties must be considered to perform this comparison? Answers to these questions will help us shape the required client-library representation to generate the expected sample.

Lastly, we aim at leveraging Maracas capabilities to also serve client developers. For instance, we intend to develop an improved version of Dependabot that informs client projects about the introduction of BCs in the libraries they are depending on, and their impact on their own code. We would also like to provide an *automatic upgrade* approach to help clients deal with the library-client co-evolution problem. For instance, we can leverage the wisdom of the crowd to identify upgrading patterns on client projects when facing a specific BC introduced in a library. We can also rely on the internal upgrades performed within the library to assist the client with its upgrading process.

Part V

APPENDIX

A

BREAKBOT SURVEY

Role in the Project

What is your experience working on open-source libraries? [Single-choice question]

- ☐ 0-2 years
- ☐ 3-6 years
- ☐ 7-10 years
- ☐ >10 years

Project Context

When reviewing non-trivial pull requests for this library (internal or external contributors), are you concerned about their impact on client code? [Single-choice question]

- ☐ Not concerned at all
- ☐ Moderately concerned
- ☐ Very concerned

When reviewing non-trivial pull requests for this library (internal or external contributors), are you confident in your estimate of their impact on client code? [Single-choice question]

☐ I do not attempt to estimate

☐ I am not confident in my estimate

☐ I am confident in my estimate

☐ Other: _____

Does the library use any of the following approaches to deal with breaking changes and their impact? [Multiple-choice question]

☐ Semantic versioning

☐ Regression testing

☐ Code review

☐ Static analysis tools

☐ None

☐ Other: _____

Can you give us more information on how such approaches are integrated into the development process of this library? [Open question] _____

PR & BreakBot

Do you think the potential impact of the changes in this pull request is a concern? [Single-choice question]

☐ Yes

☐ No

☐ Not sure

Before seeing the BreakBot report, did you think the pull request introduced breaking changes (regardless of their impact on clients)? [Single-choice question]

- ☐ Yes
- ☐ No
- ☐ I did not know
- ☐ I did not think about it
- ☐ No answer

Does the BreakBot report help you better assess the breaking changes introduced in this pull request? [Single-choice question]

- ☐ Yes, I did not think about breaking changes
- ☐ Yes, I had trouble estimating the amount of breaking changes
- ☐ Yes, I underestimated the amount of breaking changes
- ☐ Yes, I overestimated the amount of breaking changes
- ☐ No, the report describes the situation I was expecting
- ☐ Other: _____

Before seeing the BreakBot report, did you think these changes impacted client code? [Single-choice question]

- ☐ Yes
- ☐ No
- ☐ I did not know
- ☐ I did not think about it

Does the BreakBot report help you better assess the impact of the changes introduced in this pull request on client code? [Single-choice question]

- ☐ Yes, I did not think about the impact
- ☐ Yes, I had trouble estimating the impact of breaking changes
- ☐ Yes, I underestimated the impact of breaking changes
- ☐ Yes, I overestimated the impact of breaking changes
- ☐ No, the report describes the situation I was expecting
- ☐ Other: _____

Do you have any feedback on the BreakBot report (comments, problems, improvements)? [Open question] _____

What effect will the BreakBot report have on your original decision on this pull request (acceptance, refusal, or modification requests)? [Single-choice question]

- ☐ It will boost my confidence about my decision
- ☐ It will change my decision
- ☐ It will have no effect about my decision

Why? [Open question] _____

Would you be interested to include a tool such as BreakBot as part of this library development process?

- ☐ Not interested at all
- ☐ Moderately interested
- ☐ Very interested

Is there any other comment or information you would like to share with us about BreakBot? [Open question] _____

Follow-up Study

If you would like to know about the results of our study, please leave us your email [Open question] _____

Would you agree to participate in a follow-up study (in this case, make sure to leave us your email address)? [Single-choice question]

- ☐ Yes
- ☐ No
- ☐ No answer

BIBLIOGRAPHY

[1] Pietro Abate, Roberto Di Cosmo, Louis Gesbert, Fabrice Le Fessant, Ralf Treinen, and Stefano Zacchiroli. "Mining Component Repositories for Installability Issues." In: *12th Working Conference on Mining Software Repositories*. Piscataway: IEEE, 2015, pp. 24–33. ISBN: 978-0-7695-5594-2. DOI: 10.1109/MSR.2015.10.

[2] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. "Dependency Solving: A Separate Concern in Component Evolution Management." In: *Journal of Systems and Software* 85.10 (2012), pp. 2228–2240. ISSN: 0164-1212. DOI: 10.1016/j.jss.2012.02.018.

[3] OSGi Alliance. *Guidelines*. https://goo.gl/kT4FU6. n.d.

[4] OSGi Alliance. *OSGi Developer Certification – Professional*. https://goo.gl/TftHFF. n.d.

[5] The OSGi Alliance. *OSGi Core Release 6 Specification*. 2014. URL: https://docs.osgi.org/download/r6/osgi.core-6.0.0.pdf.

[6] The OSGi Alliance. *OSGi Core Release 8 Specification*. 2021. URL: https://docs.osgi.org/specification/osgi.core/8.0.0.

[7] Lowell Jay Arthur. *Software Evolution: The Software Maintenance Challenge*. 1st ed. John Wiley & Sons, 1988.

[8] Roland Barcia, Tim deBoer, Jeremy Hughes, and Alasdair Nottingham. *Developing OSGi enterprise applications*. https://dokumen.tips/documents/developing-osgi-enterprise-applications-2016-08-18-developing-osgi-enterprise.html. 2010.

[9] Neil Bartlett and Peter Kriens. *bndtools: mostly painless tools for OSGi*. https://www.slideshare.net/mfrancis/osgi-community-event-2010-rapid-bundle-development-with-bndtools-for-eclipse. 2010.

[10] Bas Basten, Mark Hills, Paul Klint, Davy Landman, Ashim Shahi, Michael J. Steindorfer, and Jurgen J. Vinju. "M3: A General Model for Code Analytics in Rascal." In: *1st IEEE International Workshop on Software Analytics*. IEEE Computer Society, 2015, pp. 25–28. DOI: 10.1109/SWAN.2015.7070485.

[11] Veronika Bauer and Lars Heinemann. "Understanding API Usage to Support Informed Decision Making in Software Maintenance." In: *16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 435–440. ISBN: 978-0-7695-4666-7. DOI: 10.1109/CSMR.2012.55.

[12] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. "The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache." In: *International Conference on Software Maintenance*. IEEE Computer Society, 2013, pp. 280–289. DOI: 10.1109/ICSM.2013.39.

[13] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. "The Maven Dependency Graph: a Temporal Graph-based Representation of Maven Central." In: *16th International Conference on Mining Software Repositories*. IEEE & ACM, 2019, pp. 344–348. DOI: 10.1109/MSR.2019.00060.

[14] Amine Benelallam, Nicolas Harrand, César Soto Valero, Benoit Baudry, and Olivier Barais. *Maven Central Dependency Graph*. last access 09.04.2022. 2018. DOI: 10.5281/zenodo.1489120. URL: https://zenodo.org/record/1489120#.YfyzCNso9H4.

[15] Stephen M. Blackburn et al. "Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century."

In: *Commun. ACM* 51.8 (2008), pp. 83–89. ISSN: 0001-0782. DOI: 10.1145/1378704.1378723.

[16] Kelly Blincoe, Francis Harrison, Navpreet Kaur, and Daniela Damian. "Reference Coupling: An Exploration of Inter-project Technical Dependencies and Their Characteristics Within Large Software Ecosystems." In: *Information and Software Technology* 110 (2019), pp. 174–189. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2019.03.005.

[17] Christopher Bogart, Christian Kästner, James D. Herbsleb, and Ferdian Thung. "How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems." In: *24th International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 109–120. DOI: 10.1145/2950290.2950325.

[18] Christopher Bogart, Christian Kästner, James D. Herbsleb, and Ferdian Thung. "When and How to Make Breaking Changes: Policies and Practices in 18 Open Source Software Ecosystems." In: *ACM Trans. Softw. Eng. Methodol.* 30.4 (2021). ISSN: 1049-331X. DOI: 10.1145/3447245.

[19] Pierre Bourque and Richard E. Fairley, eds. *Guide to the Software Engineering Body of Knowledge*. Piscataway: IEEE Computer Society, 2014. ISBN: 978-0-7695-5166-1.

[20] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. "Why and How Java Developers Break APIs." In: *25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2018, pp. 255–265. DOI: 10.1109/SANER.2018.8330214.

[21] John Businge, Simon Kawuma, Moses Openja, Engineer Bainomugisha, and Alexander Serebrenik. "How Stable Are Eclipse Application Framework Internal Interfaces?" In: *26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2019, pp. 117–127. DOI: 10.1109/SANER.2019.8668018.

[22] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. "An Empirical Study of the Evolution of Eclipse Third-party Plug-ins." In: *Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*. New York: Association for Computing Machinery, 2010, 63–72. ISBN: 9781450301282. DOI: 10.1145/1862372.1862389.

[23] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. "Eclipse API Usage: the Good and the Bad." In: *Software Quality Journal* 23.1 (2013), pp. 107–141. ISSN: 1573-1367. DOI: 10.1007/s11219-013-9221-3.

[24] Gerardo Canfora and Aniello Cimitile. "Software Maintenance." In: *Handbook of Software Engineering and Knowledge Engineering*. Ed. by S. K. Chang. World Scientific Publishing Company, 2001, pp. 91–120. DOI: 10.1142/9789812389718_0005.

[25] Ned Chapin, Joanne E. Hale, Khaled Md. Kham, Juan F. Ramil, and Wui-Gee Tan. "Types of Software Evolution and Software Maintenance." In: *Journal of Software Maintenance* 13.1 (Jan. 2001), 3–30. ISSN: 1040-550X.

[26] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. "Taming Behavioral Backward Incompatibilities via Cross-Project Testing and Analysis." In: *42nd International Conference on Software Engineering*. New York: Association for Computing Machinery, 2020, 112–124. ISBN: 9781450371216. DOI: 10.1145/3377811.3380436.

[27] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. "Lags in the Release, Adoption, and Propagation of *npm* Vulnerability Fixes." In: *Empirical Software Engineering* 26.3 (2021), pp. 1–28. ISSN: 1573-7616. DOI: 10.1007/s10664-021-09951-x.

[28] Bradley E. Cossette and Robert J. Walker. "Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries." In: *20th International*

Symposium on the Foundations of Software Engineering. New York: Association for Computing Machinery, 2012. ISBN: 978-1-4503-1614-9. DOI: 10.1145/2393596.2393661.

[29] Ward Cunningham. "The WyCash Portfolio Management System." In: *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications*. New York: Association for Computing Machinery, 1992, 29–30. ISBN: 0897916107. DOI: 10.1145/157709.157715.

[30] Benjamin Danglot, Martin Monperrus, Walter Rudametkin, and Benoit Baudry. "An Approach and Benchmark to Detect Behavioral Changes of Commits in Continuous Integration." In: *Empirical Software Engineering* 25.4 (2020), pp. 2379–2415. ISSN: 1573-7616. DOI: 10.1007/s10664-019-09794-7.

[31] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. "Automatic Test Improvement with DSpot: A Study with Ten Mature Open-Source Projects." In: *Empirical Softw. Engg.* 24.4 (2019), 2603–2635. ISSN: 1382-3256. DOI: 10.1007/s10664-019-09692-y.

[32] Joe Darcy. *Kinds of Compatibility*. 2021. URL: https://wiki.openjdk.org/display/csr/Kinds+of+Compatibility.

[33] Alexandre Decan and Tom Mens. "Lost in Zero Space – An Empirical Comparison of o.y.z Releases in Software Package Distributions." In: *Science of Computer Programming* 208 (2021), p. 102656. ISSN: 0167-6423. DOI: 10.1016/j.scico.2021.102656.

[34] Alexandre Decan and Tom Mens. "What Do Package Dependencies Tell Us About Semantic Versioning?" In: *IEEE Trans. Software Eng.* 47 (2021), pp. 1226–1240. DOI: 10.1109/TSE.2019.2918315.

[35] Alexandre Decan, Tom Mens, and Maëlick Claes. "An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems." In: *24th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2017,

pp. 2–12. ISBN: 978-1-5090-5501-2. DOI: 10.1109/SANER.2017.7884604.

[36] Alexandre Decan, Tom Mens, and Eleni Constantinou. "On the Evolution of Technical Lag in the npm Package Dependency Network." In: *International Conference on Software Maintenance and Evolution*. IEEE, 2018, pp. 404–414. DOI: 10.1109/ICSME.2018.00050.

[37] Alexandre Decan, Tom Mens, and Eleni Constantinou. "On the Impact of Security Vulnerabilities in the npm Package Dependency Network." In: *15th International Conference on Mining Software Repositories*. New York: Association for Computing Machinery, 2018, 181–191. ISBN: 9781450357166. DOI: 10.1145/3196398.3196401.

[38] Alexandre Decan, Tom Mens, and Philippe Grosjean. "An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems." In: *Empirical Software Engineering* 24 (2019), pp. 381–416. DOI: 10.1007/s10664-017-9589-y.

[39] Jim Des Rivières. *Evolving Java-based APIs*. https://tinyurl.com/yyqguo34. last access 26.07.2019. 2007.

[40] Jens Dietrich, Kamil Jezek, and Premek Brada. "Broken Promises: An Empirical Study into Evolution Problems in Java Programs Caused by Library Upgrades." In: *Conference on Software Maintenance, Reengineering, and Reverse Engineering*. IEEE Computer Society, 2014, pp. 64–73. DOI: 10.1109/CSMR-WCRE.2014.6747226.

[41] Jens Dietrich, David J. Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. "Dependency Versioning in the Wild." In: *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 2019, 349–359. DOI: 10.1109/MSR.2019.00061.

[42] Danny Dig and Ralph E. Johnson. "The Role of Refactorings in API Evolution." In: *21st International Conference on Software Maintenance*. IEEE Computer Society, 2005, pp. 389–398. DOI: 10.1109/ICSM.2005.90.

[43] Danny Dig and Ralph Johnson. "How Do APIs Evolve? A Story of Refactoring: Research Articles." In: *J. Softw. Maint. Evol.* 18.2 (2006), pp. 83–107. ISSN: 1532-060X. DOI: 10.1002/smr.v18:2.

[44] Bernhard Dorninger. *Experiences with OSGi in Industrial Applications*. https://www.slideshare.net/mfrancis/osgi-community-event-2010-experiences-with-osgi-in-industrial-applications. 2010.

[45] Mattia Fazzini, Qi Xin, and Alessandro Orso. "APIMigrator: An API-Usage Migration Tool for Android Apps." In: *7th International Conference on Mobile Software Engineering and Systems*. New York: Association for Computing Machinery, 2020, 77–80. ISBN: 9781450379595. DOI: 10.1145/3387905.3388608.

[46] Apache Felix. *OSGi Frequently Asked Questions*. https://felix.apache.org/documentation/tutorials-examples-and-presentations/apache-felix-osgi-faq.html. 2013.

[47] Apache Felix. *Dependency Manager – Background*. https://felix.apache.org/documentation/subprojects/apache-felix-dependency-manager/guides/background.html. 2015.

[48] Ulf Fildebrandt. *Structuring Software Systems with OSGi*. https://www.slideshare.net/mfrancis/structuring-software-systems-with-os-gi-ulf-fildebrandt. 2011.

[49] Thomas Forster, Thorsten Keuler, Jens Knodel, and Michael-Christian Becker. "Recovering Component Dependencies Hidden by Frameworks–Experiences from Analyzing OSGi and Qt." In: *17th European Conference on Software Maintenance and Reengineering*. USA: IEEE Computer Society, 2013, 295–304. ISBN: 978-0-7695-4948-4. DOI: 10.1109/CSMR.2013.38.

[50] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-48567-2.

[51] Amir Ghorbani, Nathan W. Cassee, Derek Robinson, Adam Alami, Neil Ernst, Alexander Serebrenik, and Andrzej Wasowski. "Autonomy Is An Acquired Taste: Exploring Developer Preferences for GitHub Bots." In: *45th IEEE/ACM International Conference on Software Engineering*. 2022.

[52] Martin Glinz. "A Risk-Based, Value-Oriented Approach to Quality Requirements." In: *IEEE Software* 25.2 (2008), pp. 34–41. DOI: 10.1109/MS.2008.31.

[53] Michael W. Godfrey and Daniel M. German. "On the Evolution of Lehman's Laws." In: *Journal of Software: Evolution and Process* 26.7 (2014), pp. 613–619. DOI: 10.1002/smr.1636.

[54] Robert Goldblatt and Marcel Jackson. "Well-Structured Program Equivalence is Highly Undecidable." In: *ACM Trans. Comput. Logic* 13.3 (2012). ISSN: 1529-3785. DOI: 10.1145/2287718.2287726.

[55] Jesus M. Gonzalez-Barahona, Paul Sherwood, Gregorio Robles, and Daniel Izquierdo. "Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is." In: *Open Source Systems: Towards Robust Practices*. Ed. by Federico Balaguer, Roberto Di Cosmo, Alejandra Garrido, Fabio Kon, Gregorio Robles, and Stefano Zacchiroli. Cham: Springer, 2017, pp. 182–192. ISBN: 978-3-319-57735-7. DOI: 10.1007/978-3-319-57735-7_17.

[56] Google. *API Differences between Guava 30.0-jre and Guava 30.1.1-jre*. 2021. URL: https://guava.dev/releases/30.1.1-jre/api/diffs/.

[57] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Oracle America, Inc., 2015.

[58] Georgios Gousios, Martin Pinzger, and Arie van Deursen. "An Exploratory Study of the Pull-Based Software Development Model." In: *36th International Conference on Software Engineering*. New York: Association for Computing

Machinery, 2014, pp. 345–355. ISBN: 9781450327565. DOI: 10.1145/2568225.2568260.

[59] Gary Gregory. *How We Handle Binary Compatibility at Apache Commons*. 2020. URL: https://garygregory.wordpress.com/2020/06/14/how-we-handle-binary-compatibility-at-apache-commons/.

[60] Alexander Grzesik. *TRESOR: the Modular Cloud – Building a Domain Specific Cloud Platform with OSGi*. https://es.slideshare.net/mfrancis/tresor-the-modular-cloud-building-a-domain-specific-cloud-platform-with-osgi-alexander-grzesik. 2013.

[61] Brett Hackleman and James Branigan. *OSGi: the Best Tool in Your Embedded Systems Toolbox*. https://www.slideshare.net/bretth/osgi-best-tool-in-your-embedded-systems-toolbox. 2009.

[62] Bentley J. Hargrave and Peter Kriens. *OSGi Best Practices!* https://www.slideshare.net/mfrancis/osgi-best-practices-learn-how-to-prevent-common-mistakes-and-build-robust-reliable-modular-and-extendable-systems-using-osgi-technology-p-kriens-bj-hargrave. 2007.

[63] Bentley J. Hargrave and Jeff McAffer. *Best Practices for Programming Eclipse and OSGi*. https://www.eclipse.org/equinox/documents/eclipsecon2006/Best%20Practices%20for%20Programming%20Eclipse%20and%20OSGi.pdf. 2006.

[64] Nicolas Harrand, Amine Benelallam, César Soto-Valero, François Bettega, Olivier Barais, and Benoit Baudry. "API Beauty Is in the Eye of the Clients: 2.2 Million Maven Dependencies Reveal the Spectrum of Client-API Usages." In: *J. Syst. Softw.* 184 (2022), p. 111134. DOI: 10.1016/j.jss.2021.111134.

[65] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. "Software Ecosystem Call Graph for Dependency Management." In: *40th International Conference on*

Software Engineering: New Ideas and Emerging Technologies Results. Association for Computing Machinery, 2018, pp. 101–104. DOI: 10.1145/3183399.3183417.

[66] André Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, and Stéphane Ducasse. "How Do Developers React to API Evolution? A Large-Scale Empirical Study." In: *Software Quality Journal* 26.1 (2018), 161–191. ISSN: 0963-9314. DOI: 10.1007/s11219-016-9344-4.

[67] André Hora, Marco Tulio Valente, Romain Robbes, and Nicolas Anquetil. "When Should Internal Interfaces Be Promoted to Public?" In: *24th International Symposium on Foundations of Software Engineering*. New York: ACM, 2016, pp. 278–289. ISBN: 9781450342186. DOI: 10.1145/2950290.2950306.

[68] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. "Dependency Smells in JavaScript Projects." In: *IEEE Transactions on Software Engineering* (2021). DOI: 10.1109/TSE.2021.3106247.

[69] Dileepa Jayakody. *Building a Modular Server Platform with OSGi*. https://pt.slideshare.net/DileepaJayakody1/building-a-modular-server-platform-with-osgi. 2012.

[70] Kamil Jezek and Jens Dietrich. "API Evolution and Compatibility: A Data Corpus and Tool Evaluation." In: *J. Object Technol.* 16 (2017), 2:1–23. DOI: 10.5381/jot.2017.16.4.a2.

[71] Kamil Jezek, Jens Dietrich, and Premek Brada. "How Java APIs Break—An Empirical Study." In: *Inf. Softw. Technol.* 65 (2015), pp. 129–146. DOI: 10.1016/j.infsof.2015.02.014.

[72] Kamil Jezek, Lukas Holy, Antonin Slezacek, and Premek Brada. "Software Components Compatibility Verification Based on Static Byte-Code Analysis." In: *39th Euromicro Conference on Software Engineering and Advanced Applica-*

tions. USA: IEEE Computer Society, 2013, 145–152. ISBN: 9780769550916. DOI: 10.1109/SEAA.2013.58.

[73] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. "Developer mistakes in writing Android manifests: An empirical study of configuration errors." In: *14th International Conference on Mining Software Repositories*. Piscataway: IEEE, 2017, pp. 25–36. ISBN: 978-1-5386-1544-7. DOI: 10.1109/MSR.2017.41.

[74] Emily Jiang. *OSGi Application Best Practices*. https://www.slideshare.net/mfrancis/best-practices-for-enterprise-osgi-applications-emily-jiang. 2012.

[75] Huaxi Jiang, Jie Zhu, Li Yang, Geng Liang, and Chun Zuo. "DeepRelease: Language-agnostic Release Notes Generation from Pull Requests of Open-source Software." In: *28th Asia-Pacific Software Engineering Conference*. 2021, pp. 101–110. DOI: 10.1109/APSEC53868.2021.00018.

[76] Gerd Kachel, Stefan Kachel, and Ksenija Nitsche-Brodnjan. *Migration from Java EE Application Server to Server-side OSGi for Process Management and Event Handling*. https://www.slideshare.net/mfrancis/osgi-community-event-2010-migration-from-java-ee-application-server-to-serverside-osgi-for-process-management-and-event-handling. 2010.

[77] Maya Kaczorowski. *Secure at Every Step: How GitHub's Dependency Graph Is Generated*. 2020. URL: https://github.blog/2020-08-04-secure-at-every-step-how-githubs-dependency-graph-is-generated/.

[78] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. "The Promises and Perils of Mining GitHub." In: *11th Working Conference on Mining Software Repositories*. New York: Association for Computing Machinery, 2014, pp. 92–101. ISBN: 9781450328630. DOI: 10.1145/2597073.2597074.

[79] Vassilios Karakoidas, Dimitris Mitropoulos, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. "Generating the Blueprints of the Java Ecosystem." In: *12th Working Conference on Mining Software Repositories*. Piscataway: IEEE, 2015, pp. 510–513. ISBN: 978-0-7695-5594-2. DOI: 10.1109/MSR.2015.76.

[80] Taranjeet Kaur, Nisha Ratti, and Parminder Kaur. "Applicability of Lehman Laws on Open Source Evolution: A Case Study." In: *International Journal of Computer Applications* 93.18 (2014), pp. 40–46.

[81] Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. "Understanding Type Changes in Java." In: *28th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2020, pp. 629–641. DOI: 10.1145/3368089.3409725.

[82] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. "Structure and Evolution of Package Dependency Networks." In: *14th International Conference on Mining Software Repositories*. IEEE Press, 2017, 102–112. ISBN: 9781538615447. DOI: 10.1109/MSR.2017.55.

[83] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. "Systematic Literature Reviews in Software Engineering - A Systematic Literature Review." In: *Inf. Softw. Technol.* 51.1 (2009), pp. 7–15. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2008.09.009.

[84] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. "EASY Meta-programming with Rascal." In: *3rd International Summer School on Generative and Transformational Techniques in Software Engineering*. Ed. by João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 222–289. ISBN: 978-3-642-18023-1. DOI: 10.1007/978-3-642-18023-1_6.

[85] Raula Gaikovina Kula, Coen De Roover, Daniel M German, Takashi Ishio, and Katsuro Inoue. "Modeling Library

Dependencies and Updates in Large Software Repository Universes." In: *arXiv preprint arXiv:1709.04626* (2017).

[86] Raula Gaikovina Kula, Daniel M. German, Takashi Ishio, and Katsuro Inoue. "Trusting a Library: A Study of the Latency to Adopt the Latest Maven Release." In: *22nd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2015, pp. 520–524. DOI: 10.1109/SANER.2015.7081869.

[87] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. "Do Developers Update Their Library Dependencies?" In: *Empirical Software Engineering* 23.1 (2018), pp. 384–417. ISSN: 1573-7616. DOI: 10.1007/s10664-017-9521-5.

[88] Raula Gaikovina Kula, Ali Ouni, Daniel M. Germán, and Katsuro Inoue. "An Empirical Study on the Impact of Refactoring Activities on Evolving Client-used APIs." In: *Inf. Softw. Technol.* 93 (2018), pp. 186–199. DOI: 10.1016/j.infsof.2017.09.007.

[89] Lars Kühne, Vincent Massol, and Simon Kitching. *The Clirr Maven Plugin*. https://www.mojohaus.org/clirr-maven-plugin/. last access 08.04.2020. 2003.

[90] Patrick Lam, Jens Dietrich, and David J. Pearce. "Putting the Semantics into Semantic Versioning." In: *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. New York: Association for Computing Machinery, 2020, 157–179. ISBN: 9781450381789. DOI: 10.1145/3426428.3426922.

[91] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. "A Systematic Review of API Evolution Literature." In: *ACM Comput. Surv.* 54.8 (2021). ISSN: 0360-0300. DOI: 10.1145/3470133.

[92] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Chen. "A4: Automatically assisting Android API migrations using code examples." In: *arXiv preprint arXiv:1812.04894* (2018).

[93] Julia Lawall and Gilles Muller. "Coccinelle: 10 Years of Automated Evolution in the Linux Kernel." In: *USENIX Conference on Usenix Annual Technical Conference*. USA: USENIX Association, 2018, 601–613. ISBN: 9781931971447. DOI: 10.5555/3277355.3277413.

[94] Meir M. Lehman. "Programs, Cities, Students—Limits to Growth?" In: *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*. New York: Springer, 1978, pp. 42–69. ISBN: 978-1-4612-6315-9. DOI: 10.1007/978-1-4612-6315-9_6.

[95] Meir M. Lehman. "Programs, life cycles, and laws of software evolution." In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. DOI: 10.1109/PROC.1980.11805.

[96] Meir M. Lehman and Francis N. Parr. "Program Evolution and Its Impact on Software Engineering." In: *2nd International Conference on Software Engineering*. Washington: IEEE Computer Society Press, 1976, pp. 350–357.

[97] Meir M. Lehman and Juan F. Ramil. "Software Evolution - Background, Theory, Practice." In: *Information Processing Letters* 88.1 (2003), pp. 33–44. ISSN: 0020-0190. DOI: 10.1016/S0020-0190(03)00382-X.

[98] Meir M. Lehman, Juan F. Ramil, and Kahen G. "Evolution as a Noun and Evolution as a Verb." In: *Workshop on Software and Organisation Co-evolution*. London: IEEE Computer Society, 2000, p. 2.

[99] Meir M. Lehman, Juan F. Ramil, Paul Wernick, Dewayne E. Perry, and Wladyslaw M. Turski. "Metrics and Laws of Software Evolution—The Nineties View." In: *4th International Software Metrics Symposium*. IEEE Computer Society, 1997, p. 20. DOI: 10.1109/METRIC.1997.637156.

[100] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 9 Edition*. Oracle America, Inc., 2017.

[101] R. Murray Lindsay and Andrew S. C. Ehrenberg. "The Design of Replicated Studies." In: *The American Statistician* 47.3 (1993), pp. 217–228. DOI: 10.2307/2684982.

[102] Mircea Lungu, Michele Lanza, Tudor Gîrba, and Romain Robbes. "The Small Project Observatory: Visualizing Software Ecosystems." In: *Science of Computer Programming* 75.4 (2010), pp. 264–275. ISSN: 0167-6423. DOI: 10.1016/j.scico.2009.09.004.

[103] Salvatore Mamone. "The IEEE Standard for Software Maintenance." In: *SIGSOFT Softw. Eng. Notes* 19.1 (Jan. 1994), pp. 75–76. ISSN: 0163-5948. DOI: 10.1145/181610.181623.

[104] Konstantinos Manikas and Klaus Marius Hansen. "Software Ecosystems – A Systematic Literature Review." In: *Journal of Systems and Software* 86.5 (2013), pp. 1294–1306. ISSN: 0164-1212. DOI: 10.1016/j.jss.2012.12.026.

[105] Robert C. Martin. *Design Principles and Design Patterns*. Tech. rep. Object Mentor, 2000.

[106] Matias Martinez and Martin Monperrus. "ASTOR: A Program Repair Library for Java." In: *25th International Symposium on Software Testing and Analysis*. New York, 2016, 441–444. ISBN: 9781450343909. DOI: 10.1145/2931037.2948705.

[107] Matias Martinez and Martin Monperrus. "Coming: A Tool for Mining Change Pattern Instances from Git Commits." In: *41st International Conference on Software Engineering: Companion Proceedings*. 2019, pp. 79–82. DOI: 10.1109/ICSE-Companion.2019.00043.

[108] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. "Use at Your Own Risk: The Java Unsafe API in the Wild." In: *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2015, pp. 695–710. DOI: 10.1145/2814270.2814313.

[109] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. "An Empirical Study of API Stability and Adoption in the Android Ecosystem." In: *International Conference on Software Maintenance*. IEEE, 2013, pp. 70–79. DOI: 10.1109/ICSM.2013.18.

[110] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. "A History-based Matching Approach to Identification of Framework Evolution." In: *34th International Conference on Software Engineering*. IEEE, 2012, pp. 353–363. ISBN: 978-1-4673-1067-3. DOI: 10.1109/ICSE.2012.6227179.

[111] Samim Mirhosseini and Chris Parnin. "Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-date Dependencies?" In: *32nd International Conference on Automated Software Engineering*. IEEE Computer Society, 2017, pp. 84–94. DOI: 10.1109/ASE.2017.8115621.

[112] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. "The Bug Catalog of the Maven Ecosystem." In: *11th Working Conference on Mining Software Repositories*. New York: ACM, 2014, pp. 372–375. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597123.

[113] Jerome Moliere. *10 Things to Know You Are Doing OSGi in the Wrong Way*. https://www.slideshare.net/mfrancis/10-clues-showing-that-you-are-doing-os-gi-in-the-wrong-manner-jerome-moliere. 2011.

[114] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. "Detecting Locations in JavaScript Programs Affected by Breaking Library Changes." In: *Proc. ACM Program. Lang.* 4 (2020). DOI: 10.1145/3428255.

[115] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. "ARENA: An Approach for the Automated Generation of Release Notes." In: *IEEE Transactions on Software Engineering* 43.2 (2017), pp. 106–127. DOI: 10.1109/TSE.2016.2591536.

[116] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. "Experience Paper: A Study on Behavioral Backward Incompatibilities of Java Software Libraries." In: *26th International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 215–225. DOI: 10.1145/3092703.3092721.

[117] Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. "Using Others' Tests to Identify Breaking Updates." In: *17th International Conference on Mining Software Repositories*. New York: Association for Computing Machinery, 2020, 466–476. ISBN: 9781450375177.

[118] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. "Curating GitHub for Engineered Software Projects." In: *Empirical Softw. Engg.* 22.6 (2017), pp. 3219–3253. ISSN: 1382-3256. DOI: 10.1007/s10664-017-9512-6.

[119] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. "Diversity in software Engineering Research." In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. New York: Association for Computing Machinery, 2013, pp. 466–476. ISBN: 9781450322379. DOI: 10.1145/2491411.2491415.

[120] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. "A Graph-Based Approach to API Usage Adaptation." In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. New York: Association for Computing Machinery, 2010, 302–321. ISBN: 9781450302036. DOI: 10.1145/1869459.1869486.

[121] Lina Ochoa, Thomas Degueule, and Jean-Rémy Falleri. "BreakBot: Analyzing the Impact of Breaking Changes to Assist Library Evolution." In: *44th International Conference on Software Engineering: New Ideas and Emerging Results*. New York: Association for Computing Machinery, 2022,

pp. 26–30. ISBN: 9781450392242. DOI: 10.1145/3510455.3512783.

[122] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen J. Vinju. "Breaking Bad? Semantic Versioning and Impact of Breaking Changes in Maven Central." In: *Empirical Software Engineering* 27.3 (2022). DOI: 10.1007/s10664-021-10052-y.

[123] Marcel Offermans. *Automatically Managing Service Dependencies in OSGi*. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.645.3833&rep=rep1&type=pdf. 2005.

[124] Marcel Offermans. *Using Apache Felix: OSGi Best Practices*. 2006. URL: \url{https://goo.gl/jmZsYD}.

[125] Oracle. *The Java Tutorials: Type Erasure*. 2021. URL: https://docs.oracle.com/javase/tutorial/java/generics/erasure.html.

[126] David L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules." In: *Pioneers and Their Contributions to Software Engineering: sd&m Conference on Software Pioneers*. Ed. by Manfred Broy and Ernst Denert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 479–498. ISBN: 978-3-642-48354-7. DOI: 10.1007/978-3-642-48354-7_20.

[127] David L. Parnas. "The Secret History of Information Hiding." In: *Software Pioneers: Contributions to Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2002, 399–409. ISBN: 3540430814.

[128] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code." In: *Software: Practice and Experience* 46 (2015), pp. 1155–1179. DOI: 10.1002/spe.2346.

[129] Ricardo Pérez-Castillo, Ignacio García-Rodríguez de Guzmán, and Mario Piattini. "Knowledge Discovery Metamodel-ISO/IEC 19506: A Standard to Modernize Legacy Systems." In: *Comput. Stand. Interfaces* 33.6 (2011), pp. 519–532. ISSN: 0920-5489. DOI: 10.1016/j.csi.2011.02.007.

[130] Tom Preston-Werner. *Semantic Versioning 2.0.0*. https://semver.org/. last access 30.07.2019. 2013.

[131] Steven Raemaekers. *The Maven Dependency Dataset*. last access 09.04.2022. 2013. DOI: 10.4121/uuid:68a0e837-4fda-407a-949e-a159546e67b6. URL: \url{https://data.4tu.nl/articles/dataset/The_Maven_Dependency_Dataset/12698027/1}.

[132] Steven Raemaekers, Arie van Deursen, and Joost Visser. "Measuring Software Library Stability through Historical Version Analysis." In: *28th International Conference on Software Maintenance*. IEEE Computer Society, 2012, pp. 378–387. DOI: 10.1109/ICSM.2012.6405296.

[133] Steven Raemaekers, Arie van Deursen, and Joost Visser. "The Maven Repository Dataset of Metrics, Changes, and Dependencies." In: *10th Working Conference on Mining Software Repositories*. IEEE Computer Society, 2013, pp. 221–224. DOI: 10.1109/MSR.2013.6624031.

[134] Steven Raemaekers, Arie van Deursen, and Joost Visser. "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository." In: *14th International Working Conference on Source Code Analysis and Manipulation*. USA: IEEE Computer Society, 2014, pp. 215–224. ISBN: 978-1-4799-6148-1. DOI: 10.1109/SCAM.2014.30.

[135] Steven Raemaekers, Arie van Deursen, and Joost Visser. "Semantic Versioning and Impact of Breaking Changes in the Maven Repository." In: *Journal of Systems and Software* 129 (2017), pp. 140–158. ISSN: 0164-1212. DOI: 10.1016/j.jss.2016.04.008.

[136] Romain Robbes, Mircea Lungu, and David Röthlisberger. "How Do Developers React to API Deprecation? The Case of a Smalltalk Ecosystem." In: *20th International Symposium on the Foundations of Software Engineering*. New York: Association for Computing Machinery, 2012. ISBN: 9781450316149. DOI: 10.1145/2393596.2393662.

[137] Roman Roelofsen. *Very Important Bundles*. https://www.slideshare.net/romanroe/vib-very-important-bundles. 2009.

[138] Anand Ashok Sawant. "The Impact of API Evolution on API Consumers and How This Can Be Affected by API Producers and Language Designers." PhD thesis. Delft University of Technology, 2019.

[139] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. "On the Reaction to Deprecation of 25,357 Clients of 4+1 Popular Java APIs." In: *International Conference on Software Maintenance and Evolution*. IEEE Computer Society, 2016, pp. 400–410. DOI: 10.1109/ICSME.2016.64.

[140] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. "To React, or Not to React: Patterns of Reaction to API Deprecation." In: *Empirical Software Engineering* 24.6 (2019), pp. 3824–3870. ISSN: 1573-7616. DOI: 10.1007/s10664-019-09713-w.

[141] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Valentina Piantadosi, Michele Lanza, and Rocco Oliveto. "API Compatibility Issues in Android: Causes and Effectiveness of Data-driven Detection Techniques." In: *Empirical Software Engineering* 25.6 (2020), pp. 5006–5046. ISSN: 1573-7616. DOI: 10.1007/s10664-020-09877-w.

[142] Doreen Seider, Andreas Schreiber, Tobias Marquardt, and Marlene Brüggemann. "Visualizing Modules and Dependencies of OSGi-Based Applications." In: *Working Conference on Software Visualization*. IEEE, 2016, pp. 96–100. ISBN: 978-1-5090-3850-3. DOI: 10.1109/VISSOFT.2016.20.

[143] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. "Does Your Configuration Code Smell?" In: *13th International Conference on Mining Software Repositories*. New York: Association for Computing Machinery, 2016, pp. 189–200. ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2901761.

[144] Anas Shatnawi, Hafedh Mili, Ghizlane El Boussaidi, Anis Boubaker, Yann-Gaël Guéhéneuc, Naouel Moha, Jean Privat, and Manel Abdellatif. "Analyzing Program Dependencies in Java EE Applications." In: *14th International Conference on Mining Software Repositories*. Piscataway: IEEE, 2017, pp. 64–74. ISBN: 978-1-5386-1544-7. DOI: 10.1109/MSR.2017.6.

[145] Junji Shimagaki, Yasutaka Kamei, Shane McIntosh, David Pursehouse, and Naoyasu Ubayashi. "Why are Commits Being Reverted?: A Comparative Study of Industrial and Open Source Projects." In: *International Conference on Software Maintenance and Evolution*. IEEE, 2016, pp. 301–311. DOI: 10.1109/ICSME.2016.83.

[146] James Shore. "Fail Fast [Software Debugging]." In: *IEEE Software* 21.5 (2004), pp. 21–25. ISSN: 1937-4194. DOI: 10.1109/MS.2004.1331296.

[147] Megan Squire and David Williams. "Describing the Software Forge Ecosystem." In: *45th Hawaii International Conference on System Sciences*. IEEE, 2012, pp. 3416–3425. DOI: 10.1109/HICSS.2012.197.

[148] Klaas-Jan Stol and Brian Fitzgerald. "The ABC of Software Engineering Research." In: *ACM Trans. Softw. Eng. Methodol.* 27.3 (2018). ISSN: 1049-331X. DOI: 10.1145/3241743.

[149] Klaas-Jan Stol, Michael Goedicke, and Ivar Jacobson. "Introduction to the Special Section—General Theories of Software Engineering: New Advances and Implications for Research." In: *Information and Software Technology* 70

(2016), pp. 176–180. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2015.07.010.

[150] Jacob Stringer, Amjed Tahir, Kelly Blincoe, and Jens Dietrich. "Technical Lag of Dependencies in Major Package Managers." In: *27th Asia-Pacific Software Engineering Conference*. IEEE, 2020, pp. 228–237. DOI: 10.1109/APSEC51365.2020.00031.

[151] Rok Strniša, Peter Sewell, and Matthew Parkinson. "The Java Module System: Core Design and Semantic Definition." In: New York: Association for Computing Machinery, 2007, 499–514. ISBN: 9781595937865. DOI: 10.1145/1297027.1297064.

[152] Fangchao Tian, Tianlu Wang, Peng Liang, Chong Wang, Arif Ali Khan, and Muhammad Ali Babar. "The Impact of Traceability on Software Maintenance and Evolution: A Mapping Study." In: *Journal of Software: Evolution and Process* 33.10 (2021). DOI: 10.1002/smr.2374.

[153] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. "There and Back Again: Can You Compile that Snapshot?" In: *Journal of Software: Evolution and Process* 29.4 (2017). DOI: 10.1002/smr.1838.

[154] Ervin Varga. "Introduction." In: *Unraveling Software Maintenance and Evolution: Thinking Outside the Box*. Cham: Springer International Publishing, 2017, pp. 3–16. ISBN: 978-3-319-71303-8. DOI: 10.1007/978-3-319-71303-8_1.

[155] Tim Ward. *Best Practices for (Enterprise) OSGi Applications*. https://pt.slideshare.net/mfrancis/best-practices-for-enterprise-osgi-applications-tim-ward. 2012.

[156] Thomas Watson and Peter Kriens. *OSGi Component Programming*. https://slidetodoc.com/osgi-component-programming-thomas-watson-ibm-lotus-equinox/. 2006.

[157] Mairieli Wessel, Andy Zaidman, Marco A. Gerosa, and Igor Steinmacher. "Guidelines for Developing Bots for GitHub." In: *IEEE Software* (2022). DOI: 10.1109/MS.2022.3224813.

[158] James R. Williams, Davide Di Ruscio, Nicholas Matragkas, Juri Di Rocco, and Dimitris S. Kolovos. "Models of OSS Project Meta-information: A Dataset of Three Forges." In: *11th Working Conference on Mining Software Repositories*. New York: ACM, 2014, pp. 408–411. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597132.

[159] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Norwell: Kluwer Academic Publishers, 2000. ISBN: 0-7923-8682-5.

[160] Martin R. Woodward. "Mutation Testing—Its Origin and Evolution." In: *Information and Software Technology* 35.3 (1993), pp. 163–169. ISSN: 0950-5849. DOI: 10.1016/0950-5849(93)90053-6.

[161] Jianyu Wu, Hao He, Wenxin Xiao, Kai Gao, and Minghui Zhou. "Demystifying Software Release Note Issues on GitHub." In: *30th IEEE/ACM International Conference on Program Comprehension*. 2022.

[162] Wei Wu, Bram Adams, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "ACUA: API Change and Usage Auditor." In: *14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 89–94. DOI: 10.1109/SCAM.2014.33.

[163] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. "AURA: A Hybrid Approach to Identify Framework Evolution." In: *32nd ACM/IEEE International Conference on Software Engineering*. New York: Association for Computing Machinery, 2010, 325–334. ISBN: 9781605587196. DOI: 10.1145/1806799.1806848.

[164] Wei Wu, Foutse Khomh, Bram Adams, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "An Exploratory Study of API Changes and Usages Based on Apache and Eclipse Ecosystems." In: *Empir. Softw. Eng.* 21 (2016), pp. 2366–2412. DOI: 10.1007/s10664-015-9411-7.

[165] Laerte Xavier, Aline Brito, André C. Hora, and Marco Tulio Valente. "Historical and impact analysis of API breaking changes: A large-scale study." In: *24th International Conference on Software Analysis, Evolution and Reengineering*. IEEE Computer Society, 2017, pp. 138–147. DOI: 10.1109/SANER.2017.7884616.

[166] Laerte Xavier, André C. Hora, and Marco Tulio Valente. "Why Do We Break APIs? First Answers from Developers." In: *24th International Conference on Software Analysis, Evolution and Reengineering*. IEEE Computer Society, 2017, pp. 392–396. DOI: 10.1109/SANER.2017.7884640.

[167] Yaoguo Xi, Liwei Shen, Yukun Gui, and Wenyun Zhao. "Migrating Deprecated API to Documented Replacement: Patterns and Tool." In: *The 11th Asia-Pacific Symposium on Internetware*. ACM, 2019, 15:1–15:10. DOI: 10.1145/3361242.3361246.

[168] Zhenchang Xing and Eleni Stroulia. "API-Evolution Support with Diff-CatchUp." In: *Transactions on Software Engineering* 33.12 (2007), pp. 818–836. ISSN: 1939-3520. DOI: 10.1109/TSE.2007.70747.

[169] Shengzhe Xu, Ziqi Dong, and Na Meng. "Meditor: Inference and Application of API Migration Edits." In: *27th International Conference on Program Comprehension*. IEEE & ACM, 2019, pp. 335–346. DOI: 10.1109/ICPC.2019.00052.

[170] Oleksandr Zaitsev, Stéphane Ducasse, Nicolas Anquetil, and Arnaud Thiefaine. "How Libraries Evolve: A Survey of Two Industrial Companies and an Open-Source Community." In: *29th Asia-Pacific Software Engineering Conference (APSEC 2022)*. 2022.

[171] Ahmed Zerouali, Tom Mens, Jesús M. González-Barahona, Alexandre Decan, Eleni Constantinou, and Gregorio Robles. "A Formal Framework for Measuring Technical Lag in Component Repositories - and Its Application to npm." In: *J. Softw. Evol. Process.* 31 (2019). DOI: 10.1002/smr.2157.

[172] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. "Has My Release Disobeyed Semantic Versioning? Static Detection Based on Semantic Differencing." In: *37th IEEE/ACM International Conference on Automated Software Engineering*. 2022. DOI: 10.48550/ARXIV.2209.00393.

[173] Tianyi Zhang, Björn Hartmann, Miryung Kim, and Elena L. Glassman. "Enabling Data-Driven API Design with Community Usage Data: A Need-Finding Study." In: *CHI Conference on Human Factors in Computing Systems*. New York: Association for Computing Machinery, 2020, 1—13. ISBN: 9781450367080. DOI: 10.1145/3313831.3376382.

[174] Théo Zimmermann. "Challenges in the Collaborative Evolution of a Proof Language and Its Ecosystem. (Défis dans l'évolution collaborative d'un langage de preuve et de son écosystème)." PhD thesis. Paris Diderot University, France, 2019. URL: https://tel.archives-ouvertes.fr/tel-02451322.

CURRICULUM VITAE

Lina María Ochoa Venegas was born in Bogotá, Colombia. In 2010, she started her studies in Systems and Computation Engineering with extra courses on Art at Universidad de los Andes, Bogotá, Colombia. Afterwards, she obtained her master's degree in Software Engineering at the same university under the supervision of dr. Oscar González Rojas and prof.dr. Harold Castro Barrera. She moved to Amsterdam in 2017 to start a position as a Ph.D. candidate for the European Union project, CROSSMINER. During this time she was affiliated with the Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands. After the completion of the project, she moved to Eindhoven to work as a Ph.D. candidate at Eindhoven University of Technology (TU/e) supporting teaching and research activities. She finalized her Ph.D. studies in 2022 under the supervision of dr. Thomas Degueule, prof.dr. Jurgen Vinju, and prof.dr. Mark van den Brand.

IPA DISSERTATION SERIES

Titles in the IPA Dissertation Series since 2020.

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views*. Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Co-simulation Construction*. Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game Design & Enabling Live Programming*. Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models*. Faculty of Mathematics and Natural Sciences, UL. 2020-04

N. Naus. *Assisting End Users in Workflow Systems*. Faculty of Science, UU. 2020-05

J.J.H.M. Wulms. *Stability of Geometric Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2020-06

T.S. Neele. *Reductions for Parity Games and Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2020-07

P. van den Bos. *Coverage and Games in Model-Based Testing*. Faculty of Science, RU. 2020-08

M.F.M. Sondag. *Algorithms for Coherent Rectangular Visualizations*. Faculty of Mathematics and Computer Science, TU/e. 2020-09

D. Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic*. Faculty of Sciences, Department of Computer Science, VU. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components*. Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management*. Fac-

ulty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS*. Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning*. Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs*. Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical*. Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions*. Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition*. Faculty of Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing*. Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2022-10

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving*. Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution*. Faculty of Mathematics and Computer Science, TU/e. 2023-02