Avaliação de Dependências no Desenvolvimento de Sistemas

Carlos Magno Barbosa[1], **Flávio Luiz Schiavoni**[1]

[1] Arts Lab in Interfaces, Computers, and Everything Else - ALICE
Federal University of São João del-Rei - UFSJ
São João del-Rei - MG

cmagnobarbosa@gmail.com, fls@ufsj.edu.br

Abstract. *The library re-usage is one of the most common practices to increase the efficiency of software development and the quality of developed systems. However, this re-usage of libraries can be harmful if it adds problematic dependencies to system maintenance. This paper brings an approach of topics and possible issues that must be taken in considerations during software development and maintenance according to dependencies and used libraries. This analysis can help the dependency optimization relating it with longevity, stability, maturity and maintenance of the system. To exemplify thus purpose it is presented a study of case that describes the analyze on the recovery process in the Harpia / Mosaicode application.*

Resumo. *O reúso de bibliotecas é uma das práticas mais comuns para aumentar a eficiência do processo de desenvolvimento e a qualidade dos sistemas desenvolvidos. Contudo, esse reúso pode ocorrer de forma não apropriada e adicionar dependências externas problemáticas à manutenção do sistema. Diante desse contexto, este artigo tem o objetivo de propor uma abordagem de tópicos e questões que devem ser levadas em consideração durante o desenvolvimento ou manutenção de um sistema com o intuito de analisar a dependência desse sistema para com as bibliotecas utilizadas. Essa análise tem como objetivo a otimização das dependências e está relacionada com a longevidade, estabilidade, maturidade e manutenibilidade do sistema. Para exemplificar a proposta, um estudo de caso que descreve a análise no processo de recuperação da ferramenta Harpia / Mosaicode.*

1. Introdução

Reutilizar trechos de código (reuso) tornou-se uma prática comum no desenvolvimento de software para possibilitar a entrega de aplicações complexas de forma eficiente em termos de tempo e custo [Shiva and Abou Shala 2007]. Nesse contexto, bibliotecas de software são alguns dos artefatos mais reutilizados, pois são soluções comumente fornecidas por diversos desenvolvedores, em diversas linguagens de programação para os mais variados problemas. Neste artigo, o termo biblioteca de software é utilizado para descrever trechos de códigos reutilizáveis, como APIs de funções, componentes ou *frameworks*.

Os desenvolvedores também podem criar as suas próprias bibliotecas para reutilizá-las em diferentes projetos. Isso permite a reutilização de trechos de código para criar novos sistemas de software. A reutilização dessas bibliotecas pode resultar em um ganho de produtividade dos desenvolvedores e da qualidade do código-fonte dos sistemas [Caldiera and Basili 1991].

Apesar das vantagens mencionadas, a utilização de bibliotecas provenientes de terceiros pode trazer alguns problemas para o código de um sistema. Caso as bibliotecas sejam muito genéricas ou específicas demais, os desenvolvedores precisam realizar adaptações que podem resultar em um código-fonte pouco eficiente. Outro ponto a ser avaliado é que a reutilização de uma biblioteca de terceiros não garante a integração dessa biblioteca ao projeto de maneira adequada ao sistema. [Pressman and Maxim 2016]. Pra garantir uma integração eficaz e adequada é necessária uma avaliação da maturidade e confiabilidade, análise das licenças, análise das dependências, verificação da portabilidade e adequação das bibliotecas com os requisitos do sistema [Gimenes and Huzita 2005].

Além desse problema, a utilização de uma biblioteca em um projeto pode gerar uma dependência da aplicação resultante do projeto para com esse artefato. Consequentemente, a descontinuidade dessa biblioteca pode comprometer o funcionamento e a qualidade de uma aplicação que dependa da mesma. Além disto, depender de diversas bibliotecas pode resultar em um aumento de custo de manutenção e no aumento da complexidade de um projeto. Por esta razão, a reutilização de código por meio da utilização de bibliotecas de terceiros deveria passar por uma análise criteriosa de escolha que ajude a decidir quais e como as bibliotecas devem ser reutilizadas em um determinado sistema, de modo que essa reutilização traga benefícios reais para o projeto.

Diante desse contexto, este artigo tem o objetivo de propor uma abordagem de tópicos e questões que podem ser levadas em consideração durante o processo de desenvolvimento ou de manutenção de um sistema. O intuito dessa abordagem é analisar a dependência desse sistema para com as bibliotecas utilizadas, visando reduzir o impacto negativo que este reúso pode causar na evolução desse sistema. Este artigo apresenta ainda um caso de uso em que a abordagem proposta foi aplicada no processo de refatoração da ferramenta Harpia / Mosaicode [Schiavoni and Gonçalves 2017], que estava descontinuada e encontrava-se não funcional devido a problemas em suas dependências.

2. Background

Bibliotecas de software são trechos de códigos previamente construídos que podem ser utilizados para auxiliar na implementação da funcionalidade de um sistema. Algumas bibliotecas são fornecidas em conjunto com os compiladores das linguagens de programação e fornecem soluções para problemas simples e recorrentes, como, por exemplo, entrada e saída da dados, números randômicos, funções matemáticas e obtenção de dados do sistema operacional. Há também bibliotecas obtidas separadamente, comumente baixadas a partir do site do desenvolvedor/fornecedor. Normalmente, esse último tipo de biblioteca fornece componentes ou *frameworks* que auxiliam na solução de problemas mais complexos, como por exemplo, a manipulação de bancos de dados ou a construção da arquitetura do sistema.

Um componente de software é um artefato de software reutilizável no estilo caixa-preta. Ele possui uma interface bem definida que define quais dados devem lhe ser passados e qual saída é obtidas. Diferentemente de bibliotecas, um componente resolve um problema pontual sendo que diversos componentes podem atuar em conjunto para resolver um problema mais complexo podendo formar bibliotecas de componentes. O exemplo mais comum de componentes são os *widgets* de interface gráfica com o usuário (GUI).

Framework é um artefato de software que implementa a funcionalidade de um

domínio sendo formado por um conjunto de classes, ou componentes, que possuem uma dependência entre si. Um *framework* pode ser reutilizado na forma de uma caixa-preta, caixa-branca (herança de seus classes), ou um meio termo entre esses dois últimos (chamado de caixa-cinza). Outra diferença, em comparação às bibliotecas de funções e de componentes, é que os *frameworks* assumem o controle da execução da funcionalidade, o que nos permite dizer que é o *framework* que utiliza o código do sistema, não o contrário [Abi-Antoun 2007, Shiva and Abou Shala 2007].

Quando uma aplicação faz reúso de um artefato, como, por exemplo, uma biblioteca, cria-se uma dependência dessa aplicação para com esse artefato. Essa dependência pode ser amenizada por meio de um projeto de software flexível e coeso que oculta os detalhes dos artefatos reutilizados da implementação da funcionalidade que a reutiliza. Uma forma de se conseguir isso é por meio de padrões de projeto de software.

Padrões de software são modelos de descrição de problemas recorrentes e suas soluções [Pressman and Maxim 2016]. Eles possibilitam ao desenvolvedor reutilizar a experiência e os procedimentos que foram adotados por outros desenvolvedores em situações semelhantes. Assim, ao invés de criar um solução própria que possivelmente não será a ideal, o desenvolvedor pode adotar a solução proposta por um padrão de software, que é reconhecida como a mais adequada [Group 2017].

Existem padrões para diferentes níveis do desenvolvimento de software. Entre eles, os padrões de projeto são os mais conhecidos e estão relacionados com a construção de código flexível e de manutenção facilitada [Manolescu et al. 2007]. Alguns deles objetivam diminuir a dependência entre as classes da aplicação, como, por exemplo, os padrões Abstract Factory, Decorator, Observer e Strategy , permitindo que classes possam ser removidas, modificadas e/ou inseridas sem que sejam necessárias alterações nos demais módulos da aplicação.

Não existe um consenso sobre o que é arquitetura de software. Em geral, a arquitetura está relacionada com a definição do objetivo e das interfaces de cada componente/módulo do software [Rosik et al. 2011]. A organização dos componentes de um software, seja ele construído com bibliotecas, *frameworks* e projetado ou não com o uso de padrões e outros recursos, compõe a arquitetura desse software. A arquitetura de software é um dos artefatos mais importantes no ciclo de vida de um sistema. Ela interfere nos objetivos de negócios, objetivos funcionais e na qualidade do sistema [Bessa et al. 2016, Melo et al. 2016].

3. Método

O presente trabalho propõe uma metodologia para a análise de dependências na adoção de bibliotecas em projetos de software. A metodologia proposta passa por alguns passos ou etapas, conforme será apresentado. Esta metodologia pode ser utilizada ao encontrarmos uma determinada característica do sistema que é comum a outros sistemas computacionais ou um problema comum a ser resolvido. Diante deste problema, há duas possibilidades: implementar algo que já pode estar implementado ou adotar uma solução já existente. Primeiramente, será analisada a possibilidade de desenvolver uma solução própria.

3.1. Desenvolvendo soluções próprias

A possibilidade de desenvolver uma solução própria é viável, principalmente, no caso de esta solução ser ótima e atender a todos os requisitos do projeto. Certamente espera-se que os requisitos estejam muito bem elucidados e que haja experiência na equipe com o tipo de problema que a solução pede. Desse modo, alguns passos podem ser dados:

- Garantir a componentização da solução e/ou o desenvolvimento da solução como uma biblioteca que atenda diretamente ao problema comum e que poderá ser reutilizada em outros projetos para resolver o mesmo problema.
- Documentar e incluir exemplos desta biblioteca de forma a garantir que seu uso por terceiros seja facilitado.
- Realizar testes e incluí-los na biblioteca.
- Disponibilizar esta biblioteca em um repositório próprio incluindo o código-fonte e uma licença de software (livre).
- Divulgar a sua biblioteca e convidar outras pessoas a ajudarem no projeto.

Realizando esses passos, é possível conseguir uma solução própria que atenda os requisitos da metodologia, garantindo um manutenibilidade futura para essa biblioteca. Vale lembrar que o desenvolvimento de uma solução própria pode ser a única decisão a ser tomada caso a adoção de uma biblioteca de terceiro não seja possível por qualquer dos critérios que serão apresentados a seguir.

3.2. Adotando bibliotecas de terceiros

No caso de ser decidido adotar uma biblioteca de terceiro, um processo de avaliação das características desejáveis desta biblioteca deve ser iniciado. Esta avaliação deve ser feita para tentar reduzir os impactos negativos da adição de uma biblioteca externa ao projeto e evitar problemas como, por exemplo, de portabilidade e obsolescência.

Manutenção e auditoria
A distribuição de uma biblioteca pode ser feita por meio de arquivos binários ou por meio de seu código-fonte. A distribuição do código-fonte não é comum em bibliotecas proprietárias. Por isto, essas bibliotecas não podem ser auditadas ou adaptadas pelos desenvolvedores do Sistema e, apesar de muitas vezes serem funcionais e aparentemente inofensivas, podem trazer problemas de desempenho ou de segurança para o sistema. Consequentemente, elas se tornam um ponto fraco no sistema, uma vez que o desenvolvedor não consegue garantir o funcionamento desejado. Já bibliotecas *Free Libre Open Source Software* (FLOSS) são distribuídas com seu código-fonte. Isso permite que as mesmas sejam alteradas, modificadas, distribuídas e auditadas pela equipe desenvolvedora do Sistema [Mancinelli et al. 2006]. Por este motivo, para garantir a manutenção e auditoria do código do sistema, recomenda-se a utilização de bibliotecas de código aberto.

Longevidade
A longevidade do Sistema é diretamente influenciada pela longevidade de suas bibliotecas, pois, se um produto de software depende de uma biblioteca que não possui mais atualização, o mesmo poderá se tornar obsoleto. Neste ponto, é importante que um projeto possua dependências apenas de códigos de terceiros que sejam distribuídos por entidades reconhecidas, sejam elas empresas ou comunidades de software, e que a velocidade de adequação destas bibliotecas às novas necessidades do mercado sejam adequada e de

acordo com os requisitos do Sistema. Caso uma biblioteca esteja sem sofrer atualizações há muito tempo ou se sua comunidade de desenvolvimento e/ou empresa responsável parece inativa, a utilização desta biblioteca poderá trazer problemas futuros ao sistema que dela depender. Para analisar a longevidade de um projeto é possível analisar o tempo de vida do projeto, versão em que ele se encontra e número de atualizações do mesmo ao longo de sua existência.

Maturidade
Uma biblioteca madura e testada em diversos projetos tende a ter menos falhas do que um código criado exclusivamente para resolver um único problema. Isso porque:

1. diversos desenvolvedores já analisaram esse código;
2. após ter sido utilizada em inúmeros sistemas, a biblioteca já teve diversos problemas identificados e solucionados. Algo improvável em um projeto iniciante;
3. alguns testes, como o de carga e estresse, só são possíveis após os usuários entrarem em contato com o sistema.

Por esta razão, a incorporação de uma biblioteca madura ao projeto pode adicionar ao projeto uma solução de qualidade testada e melhorada. Esta característica nos leva a acreditar que é melhor adotar de bibliotecas que tenham sido amplamente utilizadas por outros projetos e a evitar a adoção de bibliotecas novas que foram pouco testadas.

Portabilidade
A portabilidade de um Sistema é sua capacidade de funcionar em diversas plataformas. Para que isto ocorra, é necessário que as bibliotecas que o mesmo utiliza como dependência estejam disponíveis nas plataformas as quais o software será utilizado. Por esta razão, a escolha das bibliotecas deve ser embasada no suporte da plataforma de destino a essas bibliotecas. Caso a análise aqui mencionada não seja feita nos instantes iniciais do desenvolvimento do projeto, o custo na portabilidade do software poderá ser muito alto, pois será necessário portar também todas as dependências externas do mesmo.

Compatibilidade de Licenças
Um ponto que deve ser observado ao compilar, integrar e distribuir uma biblioteca externa juntamente com o Sistema é a sua licença. Quando se deseja distribuir um compilado ou fechar o código de um produto, as permissões das dependências utilizadas devem ser avaliadas pois podem haver incompatibilidades entre a licença do Sistema e as licenças de suas bibliotecas. Algumas licenças não podem ser usadas simultaneamente em um projeto e possuem restrições de compilação [German and Hassan 2009]. Portanto deve ser avaliado se as permissões de uso dessas bibliotecas atendem aos objetivos do projeto.

Independência de outras bibliotecas
Uma biblioteca pode atender aos requisitos de um sistema inicialmente mas possuir dependências de outras bibliotecas que não atendem a estes requisitos. Ao assumir a dependência de uma biblioteca que possui dependências de outras bibliotecas é necessário fazer esta mesma análise com todas as dependências sucessivamente [Kula et al. 2014]. Por esta razão, deve-se evitar criar dependência de uma biblioteca que depende de muitas bibliotecas devido ao aumento da complexidade que esta adoção pode trazer ao Sistema.

Legibilidade e documentação
A adoção de uma biblioteca aumenta a complexidade do código-fonte de um Sistema pois para entender o Sistema pode ser necessário entender a API da biblioteca em questão. Por esta razão, para garantir a legibilidade do código e o aprendizado do mesmo é necessário que a biblioteca possua documentação disponível para o aprendizado de sua API e preferencialmente possua exemplos de código para facilitar o seu aprendizado.

3.3. Integrando a solução

Independentemente de ter desenvolvido uma solução própria na forma de uma biblioteca ou ter adotado bibliotecas de terceiros, uma série de decisões podem ser tomadas na integração da solução ao projeto. Estas decisões estão associadas ao nível do acoplamento da biblioteca ao código implicar em uma dependência forte ou fraca.

Abordagem arquitetural
Para reduzir o acoplamento de um código, é possível utilizar uma abordagem arquitetural que consiga isolar a dependência. Um exemplo é a utilização de uma arquitetura em camadas onde apenas uma camada irá ter contato com a biblioteca a ser acoplada. Isso evita que a dependência seja propagada por todo o projeto e permite a substituição da dependência sem grandes custos ao projeto.

Padrões de projeto
Na programação orientada a objetos há diversos padrões de projeto que permitem isolar componentes de um sistema de maneira a criar uma interface para os mesmos, como por exemplo, o Facade ou o Bridge, e isolar a complexidade de um sistema. Com o uso desses padrões de projeto é possível encapsular a dependência em uma classe do próprio sistema e isolá-la de maneira que a substituição desta biblioteca seja possível sem grandes refatorações de código no sistema. Apesar de padrões de projeto serem uma metodologia de desenvolvimento voltada para programação orientada a objetos, este isolamento também é possível em linguagens de programação não orientadas a objetos.

3.4. Resumo do método

As características levantadas no método proposto permitem auxiliar a decisão de projeto quanto à adoção de código de terceiros no desenvolvimento de um Sistema. Parte do método proposto pode ser automatizada (eg. independência de outras bibliotecas).

4. Estudo de Caso: Harpia / Mosaicode

Este estudo e avaliação das dependências foi feito para auxiliar o processo de refatoração e empacotamento de uma ferramenta já existente chamada Harpia. Esta ferramenta tem o código aberto disponível e foi incluída nos repositórios Debian e Ubuntu por bastante tempo mas em determinado momento deixou de estar disponível devido ao abandono da manutenção de seu código. Apesar de não existir uma necessidade evidente de manutenção na funcionalidade da ferramenta, a mesma possuía dependências para com bibliotecas que tiveram seu desenvolvimento descontinuado. Por esse motivo, a ferramenta encontrava-se totalmente inoperante para os sistemas atuais. Eu sua refatoração, a ferramenta foi renomeada e passou a ser chamada de Mosaicode.

Dentre as dependências que impossibilitavam o funcionamento da ferramenta, destacam-se a biblioteca **Amara**, usada para a persistência de dados em XML, e a biblioteca **GLADE**, usada na construção da GUI da ferramenta.

Além disto, as bibliotecas Amara e Glade estavam entrelaçadas de maneira forte com o sistema, o que fez com que a refatoração tivesse que ser feita de maneira a alcançar diversas classes da aplicação. Com isto, também foi notado que seria mais adequado que a dependência ocorresse de maneira menos transversal ao código-fonte tornando uma futura modificação menos impactante.

4.1. Desenvolver ou adotar?

No instante inicial da refatoração do código do Harpia e com a ferramenta inoperante devido a dependências quebradas, surgiu a oportunidade de aplicarmos o método aqui apresentado. Diante da decisão imposta neste método, optamos por Adotar solução de terceiros. O desenvolvimento de soluções próprias para estes requisitos foi descartado pois existem soluções maduras e reconhecidas tanto para a implementação de persistência XML quanto para componentes gráficos e GUI.

Além disto, desenvolver uma solução própria teria um custo alto para o projeto, implicaria em reimplementar soluções existentes, poderia trazer erros já solucionados em códigos de terceiros, traria problemas de portabilidade e estava fora do escopo do projeto. Desse modo, foi decidido utilizar bibliotecas de terceiros e que seria necessário uma criteriosa análise de dependências para reduzir o impacto da obsolescência futura dessas bibliotecas.

Após um levantamento inicial das bibliotecas disponíveis para solucionar os pontos que causavam a obsolescência do Sistema, optou-se pelas seguintes substituições:

- A persistência XML feita anteriormente pelo **Amara** foi substituído pela biblioteca **Beautiful Soup**[1];
- Os componentes de GUI gerenciados anteriormente pela ferramenta **Glade** foi substituído pelo **PyGObject - GTK**[2];

As bibliotecas destacadas BeautifulSoup e PyGObject - GTK possuem código aberto, uma grande comunidade de desenvolvimento e suas licenças são compatíveis com a licença original da ferramenta. Elas também são utilizadas por diversos projetos de software reconhecidos e são portáveis para os Sistemas Operacionais Windows, MacOS e Linux. Além disto, estas bibliotecas possuem mais de 10 anos de existência, ampla documentação disponível para o seu aprendizado e pouca ou nenhuma dependência de outras bibliotecas de terceiros.

Considerando que as bibliotecas utilizadas atenderam de forma satisfatória todas as as características listadas anteriormente, se optou por isolar essas dependências e torná-las dependências fracas cuja substituição no futuro não implicará na reescrita de todo o sistema.

Persistência XML

Os trechos de código que liam e escreviam arquivos XML estavam espalhados por diversas classes da ferramenta. A refatoração do código e a adoção da nova biblioteca para

[1] `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`
[2] `https://pygobject.readthedocs.io/en/latest/`

esta funcionalidade iniciou-se pelo isolamento de todos os métodos de persistência em algumas classes responsáveis pela persistência.

Depois disto, foi implementada uma classe proxy que traz para o sistema todas as funcionalidades da nova dependência e todas as classes que trabalham com persistência chamam métodos desta classe do sistema. Desta maneira, caso a dependência quebre, somente será necessário realizar a manutenção na camada de abstração.

Nova GUI

No caso da interface gráfica, criar uma classe proxy resultaria em um esforço considerável de reescrita do código. Por isto, foi utilizada uma técnica para isolar a dependência baseada no padrão *Model-View-Controller* (MVC). Este padrão separa o desenvolvimento do sistema em camadas, facilitando a alteração da interface gráfica sem afetar a parte funcional. Assim, com a adoção do padrão MVC, a solução implementada separou os componentes de GUI da funcionalidade da ferramenta. Está abordagem permite inclusive que o sistema funcione sem GUI.

Além da adoção do padrão MVC, toda a utilização da API Gtk aconteceu por meio de classes que implementam componentes e estendem a API Gtk e fornecem uma interface simples para os demais componentes do sistema. Isto tornou as dependências do projeto mais fracas ou seja uma modificação de um componente implica na modificação de um trecho reduzido de código. Portanto o impacto de futuras modificações foi reduzido. A componentização dos elementos de GUI é a estratégia geral para enfraquecer a dependência e segue o principio de reduzir o número de funções de cada componente, modularizando o sistema o máximo possível.

5. Trabalhos Relacionados

Alguns trabalhos propõem abordagens de mineração de padrões de uso de APIs [Niu et al. 2017, Mendez et al. 2013]. Um padrão de uso de uma API documenta a sequência necessária de chamadas de métodos das classes dessa API para que a sua funcionalidade seja corretamente reutilizada. Esses padrões de uso são definidos por meio de processos de mineração que identificam as sequências de chamadas de métodos das classes das APIs [Zhong et al. 2009]. O intuito dessas abordagens é fazer com que as APIs sejam reutilizadas de maneira correta e eficaz pelo sistema.

Outro nicho de pesquisa aborda os desafios provenientes do reúso de software, como custo e dificuldades para a manutenção de bibliotecas, ausência de ferramentas de suporte e o custo para identificação e adaptação dessas bibliotecas [Hummel 2010]. Alguns trabalhos propõem como solução para esses desafios a reutilização sistemática de software por meio de ambientes integrados de reúso [Ahmed 2011]. Esse levantamento permitiu aos autores obter uma compreensão das atuais abordagens de reutilização sistemática e respectivos ambientes de reutilização, bem como identificar as deficiências correspondentes.

6. Conclusão

Este artigo apresentou um método para a avaliação de dependências no desenvolvimento de um Sistema partindo do entendimento que as escolhas das dependências são um passo essencial do desenvolvimento de um projeto e que essas escolhas podem ser decisivas

para o aumento de complexidade do código, custo de desenvolvimento, qualidade, portabilidade, longevidade, liberdade e produtividade de um sistema. Para que a decisão no momento de adotar ou não uma biblioteca tenha uma maior probabilidade de ser correta, este artigo se apresenta como um guia que pode auxiliar o desenvolvedor a tomar decisões quanto a criação de dependências externas em um código. Este estudo não traz uma resposta objetiva à questão da adoção de componentes de terceiros, mas pode auxiliar a equipe a considerar diversos fatores na decisão de gerar novas dependências com a adição de um componente externo.

Certamente, não utilizar dependências externas podem tornar o desenvolvimento de um Sistema mais custoso e até mesmo inviável. Por isto, apresentamos a possibilidade de garantir a manutenibilidade futura do Sistema por meio de adoção de técnicas de engenharia de software que permitem isolar bibliotecas de modo a enfraquecer a dependência do sistema em relação a códigos de terceiros. Com isto, é possível trazer para o projeto os ganhos e a maturidade da reutilização de bibliotecas de terceiros sem aumentar a complexidade do código, impacto e reduzir os custos de manutenção do sistema. Destacamos que o método proposto pode ser aplicado em sistemas legados e no desenvolvimento de novas aplicações.

Este artigo trouxe ainda um estudo de caso que utilizou o método aqui proposto na refatoração da ferramenta Harpia, que se encontrava inoperante devido a depreciação de suas dependências antigas. Essa depreciação da ferramenta Harpia permitiu ao grupo de pesquisa observar como as dependências estão diretamente relacionadas à longevidade de um Sistema e como uma dependência forte com uma biblioteca descontinuada pode resultar em um sistema não funcional e com alto custo de manutenção. Ao fim da refatoração de seu código, a ferramenta Harpia foi renomeada e passou se chamar Mosaicode.

Há uma tomada de decisão não elucidada neste trabalho que remete ao caso de nossa análise apontar para várias bibliotecas com características similares, que cumprem nossos requisitos e que atendem às necessidades. Como trabalhos futuros, pretendemos investigar metodologias que ajudem o desenvolvedor a escolher entre estas bibliotecas. Também é parte de nossos trabalhos futuros encontrar outras arquiteturas de sistemas e padrões de projetos que possam diminuir o acoplamento de bibliotecas e sistemas no que tange suas dependências de código externo.

Referências

Abi-Antoun, M. (2007). Making frameworks work: a project retrospective. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 1004–1018. ACM.

Ahmed, M. A. (2011). Towards the development of integrated reuse envirnoments for uml artifacts. In *Proceedings of the 6th International Conference on Software Engineering Advances*, pages 426 – 431.

Bessa, S., Valente, M. T., and Terra, R. (2016). Modular specification of architectural constraints. In *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pages 31–40.

Caldiera, G. and Basili, V. R. (1991). Identifying and qualifying reusable software components. *Computer*, 24(2):61–70.

German, D. M. and Hassan, A. E. (2009). License integration patterns: Addressing license mismatches in component-based development. In *Proceedings of the 31st International Conference on Software Engineering*, pages 188–198. IEEE Computer Society.

Gimenes, I. M. d. S. and Huzita, E. H. M. (2005). Desenvolvimento baseado em componentes: conceitos e técnicas. *Rio de Janeiro: Ciência Moderna*.

Group, T. H. (2017). Design patterns library.

Hummel, O. (2010). Facilitating the comparison of software retrieval systems through a reference reuse collection. In *Proceedings of the Workshop on Search-Driven Development: Users, Infrastructure, Tools and Evaluation*, pages 17–20.

Kula, R. G., De Roover, C., German, D., Ishio, T., and Inoue, K. (2014). Visualizing the evolution of systems and their library dependencies. In *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, pages 127–136. IEEE.

Mancinelli, F., Boender, J., Di Cosmo, R., Vouillon, J., Durak, B., Leroy, X., and Treinen, R. (2006). Managing the complexity of large free and open source package-based software distributions. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 199–208. IEEE.

Manolescu, D., Kozaczynski, W., Miller, A., and Hogg, J. (2007). The growing divide in the patterns world. *IEEE Software*, 24(4):61–67.

Melo, I., Santos, G., Serey, D. D., and Valente, M. T. (2016). Perceptions of 395 developers on software architecture's documentation and conformance. In *X Brazilian Symposium on Software Components, Architectures and Reuse*, pages 81–90.

Mendez, D., Baudry, B., and Monperrus, M. (2013). Empirical evidence of large-scale diversity in api usage of objected-oriented software. In *Proceedings of 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 43–52.

Niu, H., Keivanloo, I., and Zou, Y. (2017). Api usage pattern recommendation for software development. *Journal of Systems and Software*, 129:127 – 139.

Pressman, R. and Maxim, B. (2016). *Engenharia de Software-8ª Edição*. McGraw Hill Brasil.

Rosik, J., Gear, A. L., Buckley, J., Babar, M. A., and Connolly, D. (2011). Assessing architectural drift in commercial software development: a case study. *Software: Practice and Experience*, 41(1):63–86.

Schiavoni, F. L. and Gonçalves, L. L. (2017). Teste de usabilidade do sistema mosaicode. In *Anais [do] IV Workshop de Iniciação Científica em Sistemas de Informação (WICSI)*, pages 5–8, Lavras - MG - Brazil.

Shiva, S. G. and Abou Shala, L. (2007). Software reuse: Research and practice. In *Fourth International Conference on Information Technology (ITNG'07)*, pages 603–609. IEEE.

Zhong, H., Xie, T., Pei, P., and Mei, H. (2009). Mapo: mining and recommending api usage pattern. In *Proceedings of European Conference on Object-Oriented Programming*, page 318–343.