Uma Metodologia e Estudo de Caso para a Avaliação de Dependências no Desenvolvimento de Sistemas

Carlos Magno Barbosa[1], **Luan Luiz Gonçalves**[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, luanlg.cco@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

O presente artigo se trata de uma versão expandida do artigo [Barbosa and Schiavoni 2020] publicado na IV Escola Regional de Engenharia de Software, realizada em Maringá-PR entre os dias 11 e 13 de Novembro de 2020. Nesta versão expandida de nossa comunicação no evento, acrescentamos diversas figuras que ilustram nossas decisões e também o método proposto em nosso trabalho, conforme sugerido pelos participantes de nossa sessão e dos revisores de nosso trabalho. Algumas seções deste documento, sobretudo a que trata o estudo de caso, ganharam mais conteúdo puderam ilustrar melhor o que foi realizado em nossa pesquisa. Também adicionamos uma seção que apresenta as contribuições deste trabalho.

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 de forma genérica 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 desenvolvimento e manutenção do 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 por tentar se adequar a uma biblioteca. 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 necessário 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 e complexidade de manutenção 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.

Além disso, o processo de reutilização exige mudanças no ciclo de vida de desenvolvimento, o que acarreta em diversas adaptações gerenciais e a inclusão de atividades como análise de domínio[1], avaliação da maturidade e confiabilidade, análise das licenças, análise das dependências das bibliotecas, portabilidade e adequação do componente com os requisitos do projeto [Gimenes and Huzita 2005]. Por fim, a existência de uma biblioteca não garante que a mesma possa ser integrada de maneira eficaz e adequada a uma nova aplicação [Pressman and Maxim 2016]. Por esta razão a escolha de bibliotecas pode ser uma etapa muito importante no desenvolvimento de um sistema.

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

[1] Análise de domínio pode ser definida como o processo de catalogar toda informação gerada e utilizada no desenvolvimento de um projeto [Pressman and Maxim 2016].

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 2017b, Schiavoni et al. 2017, Schiavoni and Gonçalves 2017a], que estava descontinuada e encontrava-se não funcional devido a problemas em suas dependências. Espera-se que a abordagem proposta auxilie na escolha de bibliotecas, garantindo uma maior longevidade para o sistema onde foi aplicada.

As demais seções deste artigo estão organizadas da seguinte maneira: a Seção 2 descreve o cenário desta pesquisa. Na Seção 3 é apresentada a metodologia. Na Seção 4 é apresentado um estudo de caso da aplicação da metologia aplicada na recuperação da ferramenta Harpia, depois renomeada para Mosaicode. Na Seção 5 são apresentados alguns trabalhos relacionados e na Seção 7 são abordados as considerações finais e conclusão.

2. Fundamentação Teórica

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 ou por meio de ferramentas de instalação de dependências. 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 [Sandy and Schiavoni 2018, Markiewicz and de Lucena 2001].

Entendemos neste contexto que um componente de software é um artefato de software reutilizável no estilo caixa-preta. Um componente possui uma interface bem definida que define quais dados devem lhe ser passados e qual saída é obtida a partir destes dados. 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. Um exemplo bastante comum de componentes é o conjunto de *widgets* de interface gráfica com o usuário (GUI) [Lau and Wang 2007].

Já um *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 Shala 2007, Johnson 1997]. Um framework pode ainda ser horizontal ou vertical e trazer soluções para diversos domínios ou específica para um determinado domínio.

Quando uma aplicação faz reúso de um artefato, como, por exemplo, uma biblioteca, componente, e/ou framework, 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, Manolescu et al. 2007, Gamma et al. 1994] 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.

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 [Gamma et al. 1994], 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 [Vidal et al. 2016, Rosik et al. 2011, Knodel et al. 2006]. 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 [Pressman and Maxim 2016]. 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. O Método Proposto

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: adotar uma solução já existente ou implementar uma solução própria, conforme apresentada na Figura 1.

3.1. 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

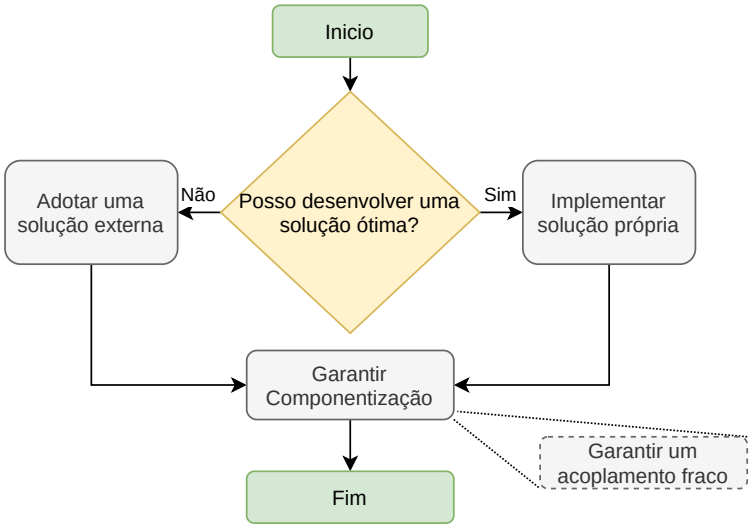

Figura 1. Primeira decisão a ser tomada na metodologia para a decisão sobre dependências. Devemos desenvolver uma solução ou adotar uma solução existente?

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 ou a aquisição do código-fonte juntamente com os componentes no caso de uma biblioteca fechada.

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 possivelmente 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.

3.2. 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.

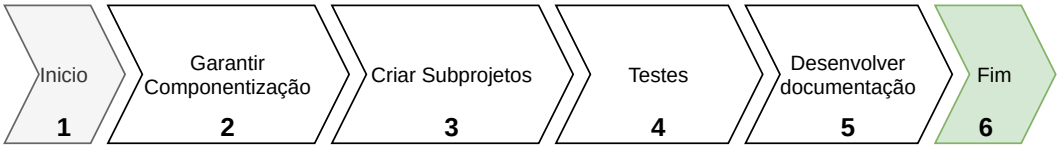

Figura 2. Decisões importantes caso deseja-se desenvolver uma solução própria para o problema comum.

Realizando esses passos, como ilustrados na Figura 2, 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 um dos critérios que serão apresentados a seguir. Outro ponto a considerar é que as características a serem avaliadas na escolha pela adoção de bibliotecas de terceiros, como Portabilidade, Compatibilidade de Licenças, Independência de outras bibliotecas, e Legibilidade e documentação, também devem ser levadas em consideração quando optamos por desenvolver uma solução própria.

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 no sistema. 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 [Gamma et al. 1994]. 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 (e.g., verificação de independência de outras bibliotecas). A Figura 3 apresenta um resumo da aplicação do método.

A aplicação deste método pode ser feita a partir do seguinte questionário:

1. Podemos solucionar o problema com uma solução própria?
    a) Qual o custo alto de implementar uma solução própria?
    b) Uma solução própria implica em reimplementações de soluções existentes?
    c) Uma solução própria traria problemas de portabilidade?
    e) Uma solução própria esta fora do escopo do projeto?
2. A nossa solução é ótima?
    a) Temos experiência sobre o domínio da aplicação?
    b) Temos condições de manter esta solução?
3. Existe uma solução madura para o problema encontrado?
    a) Será que outras pessoas encontraram o mesmo problema e já propuseram alguma solução?
    b) Essa solução atende as demandas do projeto e as estratégias de desenvolvimento do sistema?
4. Podemos adicionar essa solução de maneira a ter um acoplamento com uma dependência fraca com nosso sistema?
5. Essa solução pode ser alterada futuramente sem grandes prejuízos para o projeto?
6. O código-fonte está disponível permitindo a sua manutenção e auditoria?
7. O responsável pelo código-fonte é confiável e estável?
8. A adoção desta solução irá obrigar o sistema a adotar outras bibliotecas?
    a) Quais dependências esta dependência por trazer?

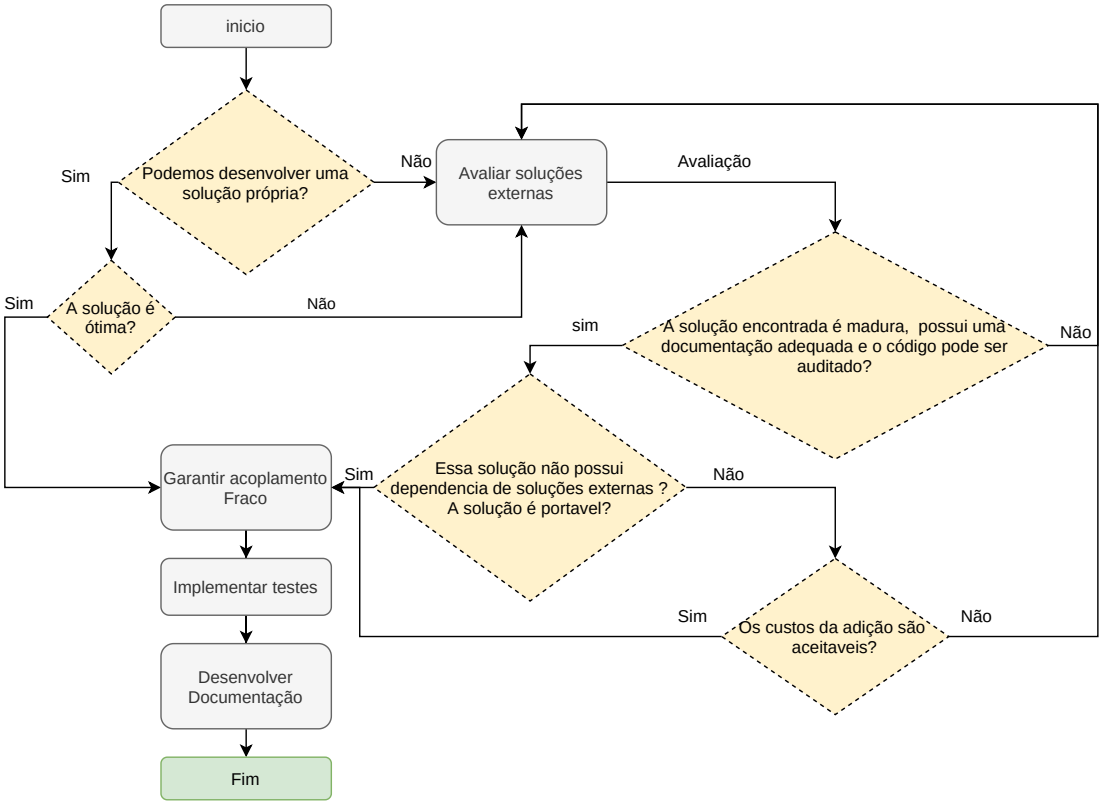

Figura 3. Resumo aplicação do método.

9. A solução é portável para as arquiteturas onde o sistema irá ser executado?
10. A solução é documentada e exemplificada?

Estes questionamentos podem nos auxiliar na decisão de projeto no momento de incluir ou não mais uma dependência em um sistema.

4. Estudo de Caso: Harpia / Mosaicode

O estudo e avaliação das dependências apresentado neste trabalho surgiu em nosso grupo de pesquisa durante o processo de refatoração e empacotamento de uma ferramenta já existente chamada Harpia, um ambiente de programação visual que permite a geração de aplicações/código-fonte para o domínio de Visão Computacional [Gonçalves and Schiavoni 2020]. Durante este processo de refatoração do Harpia, surgiram várias das perguntas que estão presente em nosso método e é difícil separar o método aqui apresentado do processo de refatoração que o gerou. Este processo envolveu vários alunos de nosso grupo de pesquisa e levou os autores deste trabalho a pensar sobre a sistematização das decisões tomadas neste momento. Assim, este processo será apresentado como um estudo de caso do método proposto neste trabalho (Seção 3) para ilustrar a utilização do mesmo.

O Harpia é uma ferramenta FLOSS e foi incluída nos repositórios Debian e Ubuntu por um bom tempo, mas em determinado momento deixou de estar disponível devido ao abandono da manutenção de seu código – aproximadamente em 2009. 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. 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. 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. A Figura 4 é um screenshot da GUI do Harpia.

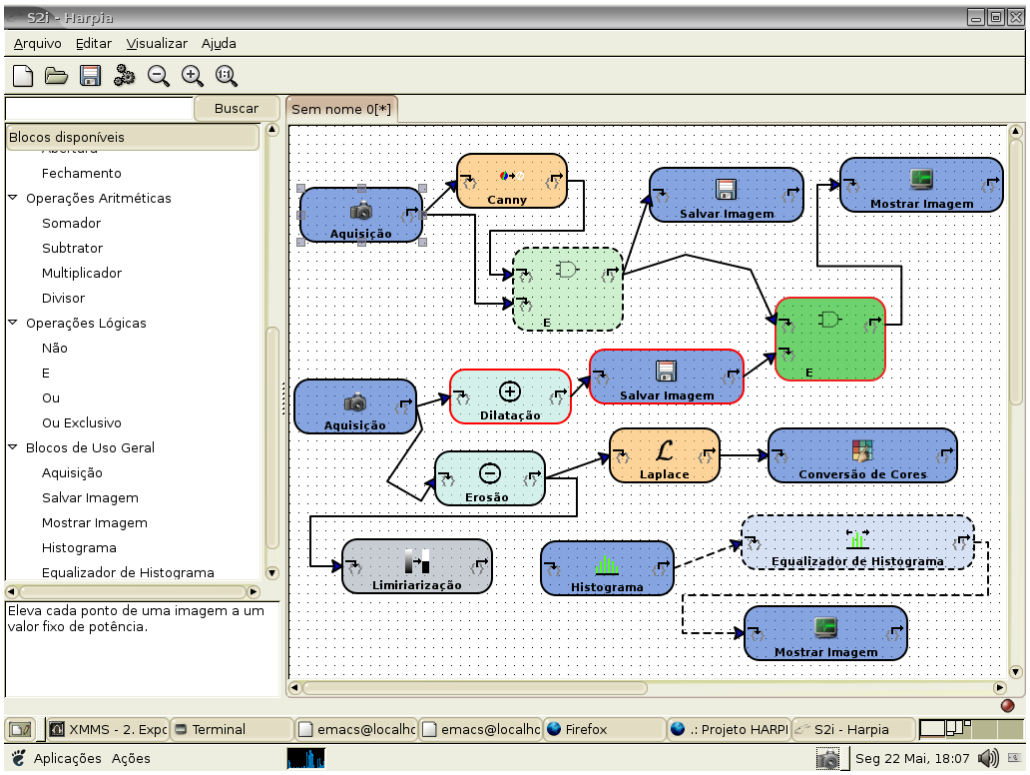

Figura 4. GUI do Harpia.

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 a) teria um custo alto para o projeto, b) implicaria em reimplementar soluções existentes, c) poderia trazer erros já solucionados em códigos de terceiros, d) traria problemas de portabilidade e 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**[2];
- Os componentes de GUI gerenciados anteriormente pela ferramenta **Glade** foi substituído pelo **PyGObject - GTK**[3];

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 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 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 [Gamma et al. 1994]. Desta maneira, caso a dependência quebre, somente será necessário realizar a manutenção na camada de abstração da persistência.

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[Krasner et al. 1988]. 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, o que não era possível na ferramenta inicial.

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, fornecendo 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

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

a dependência e segue o principio de reduzir o número de funções de cada componente, modularizando o sistema o máximo possível.

A Figura 5 ilustra a arquitetura da ferramenta, que utiliza o padrão MVC e separa o desenvolvimento do software em camadas, afim de desacoplar a GUI da parte funcional da aplicação. A arquitetura contém as seguinte camadas [Gonçalves and Schiavoni 2020]:

- **Camada de Persistência**: responsável por salvar e carregar artefatos, metadados e valores de propriedades do software. Toda a persistência XML é isolada nesta camada, utilizando a camada de Modelo e seus métodos são chamados exclusivamente pela camada Controle;
- **Camada de Controle**: responsável pelas ações no ambiente, essa camada intermedeia a interação entres as classes pertencentes às camadas distintas. Isso ocorre de forma a garantir um baixo acoplamento do código, onde todas as ações do ambiente estão definidas na camada de controle, que toma decisões sobre quais classes devem tomar parte de quais ações;
- **Camada de GUI**: a implementação da GUI é definida nessa camada, separando a interface em componentes gráfico que estendem uma classe Gtk e especializa esta classe para as necessidade do ambiente. Há também alguns componentes gráficos que baseiam-se na biblioteca Goocanvas;
- **Camada de Modelo**: permite a criação de classes modelos que definem a implementação de componentes/artefatos de software de forma abstrata. Dessa forma, os artefatos compõem um conjunto de classes com o Meta-Modelo do Sistema, que permite a modificação do sistema em tempo de execução – adicionando ou removendo instâncias destas classes.

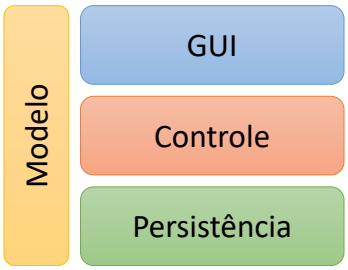

Figura 5. Arquitetura do Mosaicode inspirada no modelo em camadas e na arquitetura MVC.

Essa nova arquitetura permitiu a implementação de artefatos por meio de extensões de seu Meta-modelo do sistema, sendo também uma forma de reuso de código durante o desenvolvimento.

Da refatoração para a evolução

Durante a refatoração, o Harpia voltou a ser operante, mas foi notável que as modificações realizadas também resultou em uma evolução do ambiente. Assim, o que era para ser apenas uma refatoração para substituir códigos obsoletos se tornou uma oportunidade de modificar a ferramenta de forma mais profunda. Por definição, em edições de refatoramento, as estruturas de código do sistema são modificadas, mas seu comportamento externo é preservado. Apesar de isto ter ocorrido em um primeiro momento, o entendimento do código e a oportunidade de seguir modificando o mesmo se tornou um trabalho

maior que uma simples refatoração e que culminou na criação de outra ferramenta para o mesmo fim.

Nestas modificações, algumas premissas foram mantidas, mas o código passou a ser muito distinto da ferramenta inicial, não sendo mais exclusiva para a geração de aplicação de Visão Computacional; sendo possível estender a ferramenta para a geração de código para outros domínios e outras linguagens de programação, e adicionar novas funcionalidades ao sistema com a implementação de plugins. Por essa razão, a mesma foi rebatizada com o nome de Mosaicode – apresentada na Figura 6.

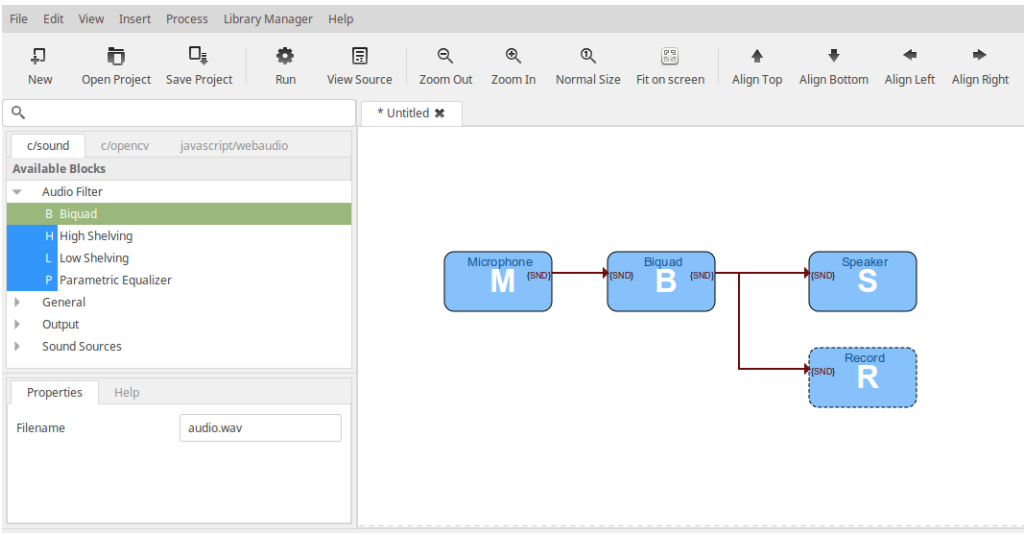

Figura 6. GUI do Mosaicode.

Um extensão no Mosaicode é o artefato de software que define uma linguagem de programação visual (Visual Programming Language – VPL), permitindo a geração de aplicações para um domínio específico (Domain-Specific (Programming) Language – DSL) e escritas uma linguagem de programação. Durante o desenvolvimento de uma aplicação, no ambiente de programação visual do Mosaicode, é possível combinar recursos providos de extensões distintas, mas apenas se as extensões utilizam a mesma linguagem de programação para o código-fonte gerado [Gonçalves and Schiavoni 2020].

As extensões do Mosaicode podem escritas em XML ou Python. Os arquivos em XML são carregados de uma pasta específica, localizada no espaço de usuário. Em Python, é utilizado o pacote *setuptools*[4] para empacotamento da extensão, permitindo a distribuição no repositório *PypI*[5] e fácil instalação e remoção. Pacotes Python implementados com o *setuptools*, podem ser instalado em espaço de usuário ou de sistema.

O artefato Plugin permite expandir a ferramenta adicionando novas features. Os plugins também são implementados em Python, utilizando o *setuptools* para empacotamento e distribuição, com opções práticas de instalação e remoção, no espaço de sistema e usuário – utilizando o *pip*[6].

Com este estudo de caso foi possível obter um versão operante do sistemas, e

[4]https://pypi.org/project/setuptools/
[5]https://pypi.org/
[6]https://pypi.org/project/pip/

desenvolver um software com novos propósitos e features. As decisões tomadas, seguindo o método proposto, resultou na organização e otimização do código-fonte – atendendo os requisitos presentes na metodologia. Processos de software e tecnologias foram adotados na implementações de features, incluindo: teste de software, gerações de documentações, empacotamento, distribuição, padronização e internacionalização de software.

5. Trabalhos Relacionados

Alguns trabalhos propõem abordagens de mineração de padrões de uso de APIs [Niu et al. 2017, Mendez et al. 2013, Wang and Han 2004]. 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 relacionado a este tema 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, Li et al. 2008, Mili et al. 1998]. Alguns trabalhos propõem como solução para esses desafios a reutilização sistemática de software por meio de ambientes integrados de reúso [Ahmed 2011]. Mahmood et al. [Mahmood et al. 2013] apresentam um levantamento indicativo dos ambientes de reutilização propostos para a reutilização sistemática de artefatos de software. 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. Sobre a escolha de bibliotecas para o desenvolvimento de software, o trabalho de [Sandy and Schiavoni 2018] traz a proposta de um modelo para comparação e escolha baseado em requisitos não funcionais.

O trabalhos relacionados abordam a maneira correta e eficaz de reuso de APIs, considerando o custo e dificuldade de manutenção/evolução. Assim como a reutilização sistemática de artefatos de software e uma proposta de modelo para comparação e escolha de bibliotecas para desenvolvimento de baseado em requisitos não funcionais. Esses trabalhos completam o tema abordado neste artigo, auxiliando as etapas da metodologia aqui definida para a decisão de projeto quanto à adoção de código de terceiros no desenvolvimento de um sistema. A metodologia proposta lista questões importantes a se pensar durante o desenvolvimento de software, considerando características que resultam em boas práticas.

6. Contribuições deste trabalho

A primeira contribuição apresentada por este trabalho é uma lista de características que um sistema pode possuir que estão diretamente ligadas as características de suas dependências. Atentar-se a estas características pode auxiliar o desenvolvedor a ponderar e escolher uma biblioteca para solucionar uma determinada funcionalidade do sistema levando em consideração os requisitos do Sistema final.

A segunda contribuição deste trabalho é a apresentação de duas técnicas de desenvolvimento que podem auxiliar o desenvolvedor a diminuir a dependência do sistema

a códigos de terceiros. Tais técnicas não são exaustivas e certamente podem ser expandidas mas podem servir como ponto de partida para o desenvolvedor analisar como uma dependência pode ser incorporada ao Sistema.

Por fim, este trabalho traz mais uma contribuição que é um questionário, organizado como um método, que pode auxiliar o desenvolvedor a analisar entre o desenvolvimento da funcionalidade ou a escolha de uma biblioteca para isto e traz indagações que ajudam a escolher a melhor opção de biblioteca para um projeto.

7. 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 do Harpia, ferramenta 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.

A refatoração da Harpia mostrou ainda que, durante o desenvolvimento de um sistema, é necessário escolher apropriadamente as dependências para reduzir as chances desse sistema se tornar obsoleto devido à obsolescência de suas dependências externas. Este estudo de caso permitiu trazer esta ferramenta novamente a atividade e a refatoração desta ferramenta culminou na renomeação da mesma para 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.

Pretendemos também pensar em como o acoplamento entre partes da mesma aplicação pode ser trabalhado de maneira a simplificar a manutenção de uma ferramenta e garantir com isto a sua longevidade. Por fim, é previsto como trabalho futuro ampliar nosso estudo de caso e fazer uma análise de dependências externas em diversas ferramentas utilizando para isto repositórios de software livre, correlacionando a sua longevidade com as dependências listadas. Este estudo pretende correlacionar as características das dependências listadas neste trabalho com a longevidade das ferramentas disponíveis nestes repositórios.

7.1. Agradecimentos

Os autores gostariam de agradecer ao CNPq (151975/2019-1), a FAPEMIG (APQ-02148-18), a Universidade Federal de São João del-Rei e aos membros do laboratório ALICE – Arts Lab in Interfaces, Computers, Education and Else – pelo apoio a esta pesquisa.

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.

Barbosa, C. M. and Schiavoni, F. L. (2020). Avaliação de dependências no desenvolvimento de sistemas. In *Anais da IV Escola Regional de Engenharia de Software*, pages 106–115, Porto Alegre, RS, Brasil. SBC.

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.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). Design patterns: elements of.

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*.

Gonçalves, L. L. and Schiavoni, F. L. (2020). Do harpia ao mosaicode a evolução de um ambiente de programação visual. In *Anais da IV Escola Regional de Engenharia de Software*, pages 316–324, Porto Alegre, RS, Brasil. SBC.

Gonçalves, L. L. and Schiavoni, F. (2020). Creating digital musical instruments with libmosaic-sound and mosaicode. *Revista de Informática Teórica e Aplicada*, 27(04):95–107.

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.

Johnson, R. E. (1997). Frameworks=(components+ patterns). *Communications of the ACM*, 40(10):39–42.

Knodel, J., Muthig, D., Naab, M., and Lindvall, M. (2006). Static evaluation of software architectures. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, pages 279–294. IEEE.

Krasner, G. E., Pope, S. T., et al. (1988). A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49.

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.

Lau, K.-K. and Wang, Z. (2007). Software component models. *IEEE Transactions on software engineering*, 33(10):709–724.

Li, Y., Zhang, L., Li, G., Xie, B., and Sun, J. (2008). Recommending typical usage examples for component retrieval in reuse repositories. In *Proceedings of 10th International Conference on Software Reuse: High Confidence Software Reuse in Large Systems*, pages 76–87.

Mahmood, S., Ahmed, M., and Alshayeb, M. (2013). Reuse environments for software artifacts: Analysis framework. In *Proceedings of IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)*, pages 35 – 40.

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.

Markiewicz, M. E. and de Lucena, C. J. (2001). Object oriented framework development. *XRDS: Crossroads, The ACM Magazine for Students*, 7(4):3–9.

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.

Mili, A., Mili, R., and Mittermeir, R. T. (1998). A survey of software reuse libraries. *Annals of Software Engineering*, 5:349 – 414.

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.

Sandy, J. M. d. S. and Schiavoni, F. L. (2018). Proposta de um modelo para comparação e escolha de frameworks para o desenvolvimento de software baseado em requisitos não funcionais. *Abakós*, 7:68–83.

Schiavoni, F. L. and Gonçalves, L. L. (2017a). Programação musical para a web com o mosaicode. In *Anais do XXVII Congresso da Associação Nacional de Pesquisa e Pós-Graduação em Música*, pages 1–6, Campinas - SP - Brazil.

Schiavoni, F. L. and Gonçalves, L. L. (2017b). 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.

Schiavoni, F. L., Gonçalves, L. L., and Gomes, A. L. N. (2017). Web audio application development with mosaicode. In *Proceedings of the 16th Brazilian Symposium on Computer Music*, pages 107–114, São Paulo - SP - 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.

Shiva, S. G. and Shala, L. A. (2007). Software reuse: Research and practice. In *Proceedings of the 4th International Conference on Information Technology – ITNG*, pages 603–609. IEEE.

Vidal, S., Guimaraes, E., Oizumi, W., Garcia, A., Pace, A. D., and Marcos, C. (2016). Identifying architectural problems through prioritization of code smells. In *X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pages 41–50.

Wang, J. and Han, J. (2004). Bide: efficient mining of frequent closed sequences. In *Proceedings of 20th International Conference on Data Engineering (ICDE)*, pages 79–90.

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.