# Verificação de Conformidade Arquitetural com Testes de Design - Um Estudo de Caso

**Izabela Melo[1], João Brunet[1], Dalton Serey[1], Jorge Figueiredo[1]**

[1]Laboratório de Práticas de Software
Departamento de Sistemas e Computação
Universidade Federal de Campina Grande (UFCG)
Campina Grande – PB – Brasil

izabela@copin.ufcg.edu.br; (jarthur,dalton,abrantes)@dsc.ufcg.edu.br

***Abstract.*** *Architectural conformance checking has been acknowledged as a key technique to assure that an implementation of a system conforms to its planned arquitecture. In this paper, we describe an experience of the application of design tests, an automatic approach to architectural conformance checking, in a real setting. The approach efficiently revealed divergences among documented, planned and implemented architectural rules. It also revelead that historically accumulated architectural violations, which could be easily dealt with at the moment they appeared, are difficult to address, due to the functional stability of the system.*

***Resumo.*** *A verificação de conformidade arquitetural tem sido reconhecida como uma das principais técnicas para garantir que a implementação de um sistema de software atenda às regras de design e de arquitetura planejadas. Este artigo descreve a experiência de aplicar testes de design, uma proposta de abordagem automática para a verificação de conformidade arquitetural, em um projeto real. A abordagem revelou com eficiência as divergências entre as regras arquiteturais documentadas, planejadas e implementadas. E que as violações arquiteturais acumuladas historicamente, que seriam de fácil resolução no momento em que surgiram, são de difícil resolução face à estabilidade funcional do sistema.*

## 1. Introdução

À medida em que o software é desenvolvido, se faz necessário desempenhar atividades que monitorem e controlem essa evolução para que o produto final esteja de acordo com suas especificações. Em particular, um dos desafios enfrentados durante a evolução e manutenção de software é verificar se sua estrutura está de acordo com as regras arquiteturais previamente estabelecidas – verificação arquitetural de software [CLEMENTS et al. 2003]. Essa atividade é de extrema importância no contexto de desenvolvimento de software, pois é através dela que a equipe detecta violações arquiteturais, isto é, divergências entre a arquitetura planejada e a implementada.

A detecção prévia de violações arquiteturais pode capacitar a equipe a reduzir os efeitos da erosão arquitetural de software, fenômeno causado pelo acúmulo de violações arquiteturais [PERRY and WOLF 1992] ao longo do tempo. Entre outros efeitos nocivos provenientes da erosão arquitetural estão a perda de estrutura do software, que impacta consideravelmente na sua complexidade e dificuldade de manutenção [GURP and BOSCH 2002][BROOKS 1975].

Este trabalho relata a experiência da aplicação de verificação arquitetural em um sistema real entitulado e-Pol – Sistema de Gestão das Informações de Polícia Judiciária,

desenvolvido em parceria pela Polícia Federal do Brasil e a Universidade Federal de Campina Grande. Apesar de haver outras formas de realizar a verificação arquitetural, como a linguagem DCL (Dependency Constraint Language), optamos por utilizar Testes de Design [BRUNET et al. 2009][BRUNET et al. 2011], uma abordagem baseada em testes que permite a escrita e a verificação automática de conformidade entre regras arquiteturais e uma dada implementação. Essa escolha foi apoiada na praticidade da abordagem por combinar a prática de verificação arquitetural à de testes automáticos. Isso permite integrar os procedimentos de verificação arquitetural às suítes de testes funcionais automatizados, sem a adição de nenhum outro ferramental ou processo.

O processo consistiu em coletar as regras arquiteturais junto à equipe de desenvolvimento, implementá-las na forma de testes de design e executar os testes. Ao final, analisamos e discutimos as violações arquiteturais junto à líder da equipe. Como era esperado, um número significativo de violações arquiteturais foi encontrado — 12 das 22 regras arquiteturais planejadas não são respeitadas na implementação. Este resultado corrobora outros anteriormente publicados [BRUNET et al. 2011] e pode ser explicado pela cultura em relação às regras de design e de arquitetura. Decisões arquiteturais não são adequadamente documentadas nem compartilhadas entre os desenvolvedores. Ao longo do tempo, tendem a ser esquecidas o que aumenta a chance de serem desrespeitadas. E a inexistência de ferramentas de apoio à verificação piora ainda mais o cenário, por dificultar a imediata detecção de violações durante o desenvolvimento e evolução do sistema.

Este artigo está organizado da seguinte maneira. A Seção 2 apresenta os conceitos fundamentais de verificação arquitetural e testes de design. A Seção 3 apresenta o estudo realizado e as lições aprendidas. A Seção 4 discute trabalhos relacionados. Por fim, a Seção 5 apresenta nossas conclusões.

## 2. Fundamentação Teórica

### 2.1. Arquitetura como conjunto de regras arquiteturais

Devido à sua importância na indústria e academia, o conceito de arquitetura de software tornou-se um tópico de interesse em Engenharia de Software. Nos últimos 20 anos, particularmente, vários pesquisadores se esforçaram em produzir uma definição para esse conceito [CLEMENTS et al. 2003, BUDGEN 2003, GARLAN and PERRY 1995, BASS et al. 2003, JANSEN and BOSCH 2005]. Essas definições abordam, por exemplo, aspectos dinâmicos, estáticos e de fluxo de dados do software. Embora a literatura contenha muitos trabalhos nesse sentido, é possível encontrar um denominador comum nessas definições – o fato da arquitetura especificar os componentes de um software e como eles devem se comunicar [BASS et al. 2003].

Nesse contexto, uma arquitetura de software pode ser vista como um conjunto de decisões/regras arquiteturais que estabelecem relações entre os componentes de uma aplicação [JANSEN and BOSCH 2005]. Por exemplo, suponha um sistema em que o arquiteto utilizou camadas para separar a apresentação, da lógica de negócio e os objetos de acesso aos dados. Um exemplo de regra arquitetural nesse sistema é o fato da camada de apresentação não poder depender diretamente dos objetos de acesso aos dados.

### 2.2. Testes de design

Teste de design é uma abordagem de verificação arquitetural. Através de testes de design, o desenvolvedor/arquiteto é capaz de especificar regras arquiteturais em formato de testes

automatizados e verificar se a implementação está de acordo com essas regras. Por exemplo, vamos novamente considerar a restrição de acesso entre o componente da camada de apresentação (e.g. pacote GUI) e o componente de acesso a dados (e.g. pacote DAO). O Algoritmo 1 ilustra o teste de design para essa regra.

Como podemos observar no Algoritmo 1, o teste consiste em obter informações a respeito dos módulos acessados por GUI e verificar se DAO não pertence a esse conjunto de módulos. Em tempo, um módulo é uma agregação de entidades, sejam elas classes, pacotes ou interfaces.

**Algoritmo 1. Pseudocódigo da implementação de um teste de design**

```
1  componenteGUI = getPackage("GUI")
2  componenteDAO = getPackage("DAO")
3  calledPackages = componenteGUI.getCalleePackages()
4  assertFalse(calleePackages.contais(componenteDAO))
```

Testes de design detectam violações arquiteturais. Por exemplo, o teste do Algoritmo 1, quando executado tendo como sistema a ser verificado o exemplo do Algoritmo 2 detectaria uma violação arquitetural na linha 4. Uma violação arquitetural pode ser representada como uma tripla (*caller*, *callee*, *type*), em que *caller* indica o módulo que viola uma regra arquitetural, *callee*, o módulo que é usado pelo caller, e *type* o tipo da violação (chamada a método, acesso a variável, generalização, etc). A violação apresentada no Algoritmo 2, por exemplo, pode ser representada por: (*GUI*, *DAO*, "chamada de método")

**Algoritmo 2. Pseudocódigo de uma violação arquitetural**

```
1   package gui;
2   class MainWindow() {
3       show() {
4               DAO.getData();
                  ...
5       }
6   }
7
8   package dao;
9   class DAO() {
10      getData() {...}
11  }
```

## 3. Estudo de Caso: Aplicação de testes de design

Neste estudo relatamos a experiência da aplicação de verificação arquitetural baseada em testes de design em um sistema real. O sistema em questão é o e-Pol – Sistema de Gestão das Informações de Polícia Judiciária, que vem sendo desenvolvido em parceria pela Polícia Federal do Brasil e a Universidade Federal de Campina Grande desde 2010. O projeto visa desenvolver uma plataforma integrada de investigação e inquérito para a Polícia Federal. No atual estágio, o sistema consiste em aproximadamente 57.000 linhas de código, 539 classes e 31 pacotes. Atualmente, conta com 10 desenvolvedores, embora um total de 41 tenham atuado no projeto, desde seu início. O processo de desenvolvimento do e-Pol é baseado em métodos ágeis. Nesse sentido, o processo adotado foca pouco em atividades de documentação e de verificação arquitetural.

O estudo realizado é de natureza exploratória em que procuramos identificar questões de pesquisa, mais que respondê-las.

| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | GUI | | | | | | x | x | x | | x | x | x | x | * |
| 2 | Action | * | | x | x | | x | | | | x | x | x | x | |
| 3 | Converter | x | x | | x | x | x | x | x | x | x | x | x | x | x |
| 4 | Validator | x | x | x | | x | x | x | x | x | x | x | x | x | x |
| 5 | Listagem | x | | x | x | | x | | | | x | x | x | x | |
| 6 | Cenario | x | x | x | x | x | | | | x | x | x | x | x | |
| 7 | DAO | x | x | x | x | | | | x | x | x | x | x | x | * |
| 8 | Session | x | x | x | x | x | x | | | | x | x | x | x | |
| 9 | Util | x | x | x | x | x | x | x | | | x | x | x | x | |
| 10 | Test | x | | | | | | | | | | | | | |
| 11 | Test.unit | x | x | | | | * | x | | | | | | | |
| 12 | Test.integration | x | | | | | | | | | | | | | |
| 13 | Log | x | x | x | x | x | x | x | x | x | x | x | x | | x |
| 14 | Model | x | x | x | x | x | x | x | * | x | x | x | x | x | |

**Figura 1. DSM dos relacionamentos entre os módulos do e-Pol.**

### 3.1. Metodologia

A metodologia utilizada para conduzir o estudo foi dividida em três etapas. Na primeira – Documentação arquitetural – coletamos as regras arquiteturais do projeto. Produzimos: 1) uma lista dos módulos em que consiste o projeto; 2) o mapeamento entre os módulos e os pacotes/classes da implementação; e 3) as regras arquiteturais planejadas, expressas na forma de relacionamentos não admissíveis entre módulos. Os dados foram obtidos através de entrevistas com a líder de desenvolvimento do projeto.

Na segunda – Verificação arquitetural – implementamos as regras arquiteturais na forma de testes de design e executamos os testes para verificar a conformidade arquitetural. A implementação dos testes foi feita por um dos autores deste artigo em concordância com a líder de projeto. Foi produzida uma lista de violações arquiteturais para cada uma das regras identificadas.

Por fim, na terceira etapa – Análise das violações – analisamos as violações arquiteturais do projeto junto à líder da equipe. Embora o método usado tenha sido em boa parte *ad-hoc*, de modo geral, o objetivo era o de confrontar a líder de desenvolvimento com as violações. Para cada violação identificada, pedimos à líder que avaliasse, confirmasse e explicasse.

### 3.2. Resultados

**Documentação arquitetural** O primeiro passo consistiu em identificar os módulos do sistema. Embora os 12 módulos que compõem o sistema tenham sido identificados facilmente, o mapeamento das entidades do código aos módulos não foi tarefa fácil. Em diversos momentos, percebemos que houve dúvidas por parte da líder em relação ao módulo a que pertencem várias das entidades. Em particular, a divisão em pacotes do sistema não reflete bem a decomposição em módulos identificada. Há vários casos em que as classes de um mesmo pacote foram mapeadas para diferentes módulos. A tarefa, contudo, claramente obrigou à reflexão e crítica da arquitetura do sistema, por parte da líder de desenvolvimento.

Uma vez coletados os módulos que compõem o sistema, coletamos as restrições arquiteturais planejadas. A DSM apresentada na Figura 1 sumariza as restrições arquiteturais identificadas. Na DSM, cada "x" representa um relacionamento proibido – o módulo da linha não pode depender do módulo da coluna. E cada "*" representa uma proibição parcial – existem exceções para os quais a dependência é permitida.

A atividade seguinte foi organizar a informação na DSM em regras arquiteturais. Identificamos um total de 22 regras, das quais 20 são do tipo *módulo A não pode depender*

*do módulo B*. As outras duas regras são semelhantes, mas de menor granularidade, e proibem dependências entre conjuntos de entidades específicas contidas pelos módulos (classes ou mesmo métodos). São, portanto, regras do tipo *módulo A não pode depender dos métodos* $M_1$, $M_2$, ..., $M_n$.

**Verificação arquitetural** Para cada uma das regras arquiteturais identificadas, implementamos os testes de design correspondentes, utilizando o DesignWizard [BRUNET et al. 2009] e o JUnit. A título de exemplo, o Algoritmo 3 corresponde ao pseudocódigo do teste de design para a regra *nenhum módulo pode ter acesso direto ao módulo Test*. Os testes de design foram executados na última versão estável do e-Pol e as violações arquiteturais detectadas foram armazenadas. No total, foram identificadas 298 violações arquiteturais referentes a 12 das regras. A implementação, portanto, está em conformidade com apenas 10 das 22 regras (45%) coletadas. A distribuição das violações foi bastante concentrada tanto por regra (160, 53, 31, 17, 12, 5, 5, 5, 4, 3, 2, 1), quanto por módulo em que ocorre (160, 54, 37, 17, 12, 7, 6, 5, 0, 0, 0, 0). A semelhança dos vetores de distribuição é consequência da lógica de formação das regras: cada regra trata, praticamente, das restrições de um módulo. A título de exemplo, o processo de verificação revelou que o módulo Model, cujas regras estabeleciam que *Model não pode depender de nenhum outro módulo*, acumula um total de 54 violações.

**Algoritmo 3. Pseudocódigo de um teste de design no e-Pol**

```
1  testaAcessoAoModuloTeste() {
2
3    moduloTeste = DesignWizard.getPackage("Test");
4    classesModuloTeste = moduloTeste.getAllClasses();
5
6    for (classNode in classesModuloTeste) {
7      for (caller in classNode.getCallerClasses()) {
8        Assert.assertTrue("Ha violacao",
9              caller.getPackage().equals(moduloTeste));
10     }
11   }
12 }
```

**Análise das violações** O relatório de violações identificadas foi analisado em conjunto com a líder de projeto. Como se havia antecipado, a líder já esperava que algumas violações fossem encontradas, dada a natureza ágil do processo de desenvolvimento e à indisponibilidade de ferramentas para apoiar o processo de verificação arquitetural. O número de violações encontrado, contudo, ainda foi surpreendente. Todas as violações mencionadas foram confirmadas[1] pela líder de projeto. Houve uma ressalva, contudo, em relação às violações dos módulos Model e GUI. Segundo a líder, um número significativo dessas violações são decorrentes de mudanças em decisões arquiteturais ao longo do tempo que implicaram no acúmulo de violações não corrigidas. Logo, embora não fossem violações às regras arquiteturais vigentes no momento do desenvolvimento, são violações à arquitetura atual. Há, portanto, um débito arquitetural, embora as razões pelas quais

[1]De fato, o processo de coleta foi iterativo. Apresentamos à líder de desenvolvimento, relatórios preliminares de violações, a partir dos quais houve a oportunidade de refletir sobre as regras arquiteturais e reescrevê-las ou aperfeiçoá-las.

tenha sido criado sejam compreensíveis. A correção das violações, contudo, pode ter um alto custo para o projeto, dado que, do ponto de vista funcional, o sistema se encontra estável. Trata-se, portanto, da identificação da necessidade de um significativo processo de refatoramento.

Um aspecto que chamou a atenção é a distribuição concentrada das violações em determinados módulos e/ou regras – 71% das violações ocorrem em apenas dois módulos do sistema (GUI e Action), sendo que 53% delas estão em um único módulo (GUI). O módulo GUI é responsável pela apresentação do sistema para o usuário e, portanto, precisa ter acesso a várias entidades. Por vezes, a divisão entre entidades possíveis de serem acessadas e entidades proibidas pode não ficar tão clara para os desenvolvedores, acarretando em violações arquiteturais. O módulo Action, por sua vez, é responsável pelas regras de negócio e pela disponibilização de informações para a camada de apresentação. Logo, o módulo Action separa a camada de visão da camada de modelo. Claramente, ter testes de design incorporados às suítes de testes automáticos do sistema permitiria evitar a adição de novas violações durante a evolução do sistema. Para o caso de haver explicações plausíveis para aceitar a violação, os testes permitem manter sob controle as situações em que as regras serão ignoradas. Em particular, a simplicidade do uso de testes de design foi percebida pela própria líder de desenvolvimento como um motivo para adotar o método para o controle e a verificação arquiteturais.

### 3.3. Lições aprendidas

Duas lições foram aprendidas durante o estudo de caso: 1) conformidade arquitetural requer apoio ferramental, e 2) Decisões arquiteturais mudam ao longo do tempo.

**Conformidade arquitetural requer apoio ferramental** A verificação de conformidade arquitetural deve ter papel relevante nos processos de desenvolvimento para garantir a qualidade do software. Contudo, a falta de ferramentas que automatizem, ainda que parcialmente, o processo torna a atividade impraticável. Técnicas de inspeção e revisão de código, programação em pares, checklists de padrões ajudam a amenizar os problemas. Contudo, como são essencialmente manuais estão sujeitas aos problemas já conhecidos: diferenças nas interpretações, erros na execução e custos elevados. Nesse sentido, defendemos que testes de design são uma forma barata e simples de incorporar a verificação de conformidade arquitetural a qualquer processo de desenvolvimento. Em particular, permite que a verificação arquitetural seja integrada ao processo de testes automáticos e que, assim, se monitore continuamente o entendimento e a atuação da equipe em relação às regras de design e de arquitetura. Por elevar a documentação das regras à condição de executáveis, reduz significativamente as falhas de comunicação da equipe em termos de decisões de design e de arquitetura. Uma vez implementadas como testes, as regras podem ser integradas ao repositório e podem ser usadas para compor as suítes de testes pré-commit (testes que condicionam a aceitação de cada commit feito pelos desenvolvedores). Além disso, por serem testes, o feedback proporcionado ao desenvolvedor em caso de falha é concreto: indica uma violação em termos dos módulos envolvidos e da regra desrespeitada.

**Decisões arquiteturais evoluem** À medida que se evolui no desenvolvimento do sistema ou que se compreende melhor os requisitos, é natural que as decisões arquiteturais

evoluam. O problema é que isso implica na situação revelada em nosso estudo de caso. Ao criar uma nova regra de design é necessário decidir o que fazer em relação aos trechos da implementação já concluídos que desrespeitam essa regra. Essa situação pode criar instantaneamente um débito arquitetural com o qual é necessário lidar. Se se optar por refatorar o sistema para deixá-lo conforme a nova regra, há que se considerar os possíveis impactos funcionais. Especialmente, quando se trata de porções estáveis do código (partes antigas do código com baixa frequência de alteração). Se se optar por não alterar o código antigo e fazer a regra valer apenas para as futuras modificações, há que se conviver com as violações arquiteturais já existentes, sem no entanto permitir que os desenvolvedores aumentem o número de violações em relação a essa regra[2]. Em ambos os casos, tomar uma decisão com clareza e de forma racional parece ser melhor alternativa do que manter o problema escondido, como ocorre quando não há um processo efetivo de verificação de conformidade arquitetural.

Da mesma forma que ocorreu nos outros estudos de caso que realizamos com testes de design para a verificação arquitetural, o líder de equipe viu-se motivado a adotar o método, tendo declarado isso explicitamente. Em geral, o único empecilho para adotar o método de imediato é a dificuldade em lidar com o débito arquitetural identificado. Embora isso demande um outro estudo, provavelmente a forma mais apropriada seja iniciar por um significativo refatoramento, para colocar o sistema em conformidade arquitetural.

## 4. Trabalhos Relacionados

A literatura relacionada à verificação de conformidade arquitetural é prolífica [BRUNET et al. 2009, BRUNET et al. 2011]. O objetivo desta seção é discutir os trabalhos que relatam a experiência da aplicação dessas abordagens ao longo do desenvolvimento de sistemas de software. Neste contexto, Murphy e Notkin descrevem a aplicação da abordagem Modelos de Reflexão no gerenciador de planilhas Excel da Microsoft [MURPHY and NOTKIN 1997, MURPHY et al. 2001]. O experimento foi executado como um exercício de reengenharia. Os resultados indicaram que o Excel, mesmo com uma documentação arquitetural vasta e constantemente atualizada, possuía uma grande quantidade de violações. Além desta constatação, como resultado da atividade de reengenharia, os desenvolvedores passaram a dominar mais os conceitos arquiteturais do sistema, uma vez que as violações eram detectadas e discutidas.

Unphon e Dittrich [UNPHON and DITTRICH 2010] realizaram um importante estudo com o objetivo de levantar conhecimento a respeito de como os *stakeholders* lidam com preocupações arquiteturais na prática. Como resultado desse estudo, os autores identificaram que usualmente existe um arquiteto chefe ou desenvolvedor central, intitulado *"walking architecture"*, que detém grande parte do conhecimento a respeito da arquitetura e que esse conhecimento é passado para os membros da equipe através de discussões pontuais. Além disso, esse estudo também confirma o fato de que a documentação arquitetural tende a não ser atualizada e, por esse motivo, torna-se obsoleta.

## 5. Conclusão

Neste estudo reiteramos a importância da verificação de conformidade arquitetural e, em particular, para reforçar a necessidade de ferramentas de apoio à atividade. Apesar de

[2]Nesse caso, é aconselhável manter a regra e tratar, explicitamente, as violações antigas como exceções. É possível escrever testes de design com exceções. Dessa forma, novas violações seriam acusadas, sem a necessidade de conviver com a reiterada indicação das violações antigas.

todo esforço por parte de uma equipe de desenvolvimento, é improvável que seja capaz de efetivamente detectar violações sem o apoio de ferramentas. Prova disso é o acúmulo de violações do mesmo tipo ao longo do desenvolvimento e a dificuldade das equipes em lidar com o chamado débito arquitetural (violações arquiteturais acumuladas no histórico do software).

Concluímos ainda que, embora restritos a decisões de design estruturais, testes de design constituem uma forma efetiva e eficiente de verificação arquitetural que podem ser facilmente integrados ao processo de desenvolvimento. Os problemas encontrados no processo de análise das regras arquiteturais apenas refletem o fato de que, na prática, as decisões arquiteturais não são concretas e objetivamente documentadas. Com isso, os problemas encontrados não indicam que a adoção na prática será difícil.

Como trabalho futuro, pretendemos entender melhor como verificação de conformidade arquitetural com base em testes de design pode ser incorporada a um projeto em andamento e com débito arquitetural já identificado.

## Referências


BASS, L., CLEMENTS, P., and KAZMAN, R. (2003). Software architecture in practice. *Addison-Wesley Professional*.

BROOKS, F. (1975). The mythical man-month. *Addison-Wesley Reading, Mass*, 79.

BRUNET, J., GUERRERO, D., and FIGUEIREDO, J. (May 2009). Design Tests: An Approach to Programmatically Check your Code Against Design Rules. *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), New Ideas and Emerging Results*.

BRUNET, J., GUERRERO, D., and FIGUEIREDO, J. (September 2011). Structural Conformance Checking with Design Tests: An Evaluation of Usability and Scalability. *Proceedings of the 27th International Conference on Software Maintainance (ICSM 2011)*.

BUDGEN, D. (2003). Software design. *Addison Wesley*.

CLEMENTS, P., GARLAN, D., LITTLE, R., NORD, R., and STAFFORD, J. (2003). Documenting software architectures: views and beyond. *Proceedings of the 25th International Conference on Software Engineering - IEEE*, page 740–741.

GARLAN, D. and PERRY, D. (1995). Introduction to the special issue on software architecture. *IEEE Trans. Softw. Eng.*

GURP, J. and BOSCH, J. (2002). Design erosion: problems and causes. *Journal of systems and software*.

JANSEN, A. and BOSCH, J. (2005). Software architecture as a set of architectural design decisions. *Proceedings of the 5th Working Conference on Software Architecture - IEEE*, page 109–120.

MURPHY, G. and NOTKIN, D. (Aug 1997). Reengineering with reflexion models: a case study. *Computer*, 30(8):29,36.

MURPHY, G., NOTKIN, D., and SULLIVAN, K. (Apr 2001). Software reflexion models: bridging the gap between design and implementation. *Software Engineering, IEEE Transactions*, 27(4):364,380.

PERRY, D. and WOLF, A. (1992). Foundations for the study of software architecture. *ACM SIG-SOFT Software Engineering Notes*, 17:40–52.

UNPHON, H. and DITTRICH, Y. (November 2010). Software architecture awareness in long-term software product evolution. *J. Syst. Softw. 83, 11*.