O artigo aborda um dos maiores desafios da engenharia de software: identificar problemas arquiteturais que aumentam significativamente os custos de manutenção e a propensão a erros em sistemas complexos. Os autores destacam que, embora existam diversas pesquisas sobre métricas de código e previsão de defeitos, essas abordagens costumam analisar arquivos de forma isolada e acabam ignorando a estrutura arquitetural como um todo. Isso é um problema porque muitos dos defeitos e dos altos custos de manutenção decorrem de falhas na arquitetura, e não apenas de “code smells” em trechos específicos do código.

Para resolver essa lacuna, os pesquisadores propõem o conceito de Design Rule Spaces (DRSpaces), que representa a arquitetura de um sistema como um conjunto de regras de projeto (design rules) e módulos independentes. As design rules funcionam como interfaces ou classes abstratas que devem se manter estáveis ao longo do tempo para evitar impactos em todo o sistema. Com base nesse modelo, os autores definiram e formalizaram cinco padrões recorrentes de problemas arquiteturais, chamados de hotspot patterns, que podem ser detectados de forma automática a partir da análise das dependências estruturais entre arquivos e do histórico de mudanças do sistema.

Os cinco hotspots identificados foram: Unstable Interface, que ocorre quando arquivos muito influentes mudam com frequência e causam efeitos em cascata; Implicit Cross-module Dependency, quando módulos que deveriam ser independentes acabam mudando juntos com frequência, revelando dependências ocultas; Unhealthy Inheritance Hierarchy, que envolve hierarquias de herança mal projetadas e que violam princípios como o da substituição de Liskov; Cross-Module Cycle, quando há ciclos de dependência entre módulos diferentes; e Cross-Package Cycle, que ocorre quando há ciclos entre pacotes que deveriam estar organizados de forma hierárquica. Segundo os autores, esses padrões não são capturados por ferramentas tradicionais como SonarQube, pois exigem uma visão integrada da estrutura e da evolução do sistema.

A principal contribuição do artigo é a criação de uma ferramenta chamada Hotspot Detector, capaz de identificar automaticamente esses padrões. A ferramenta foi aplicada em nove projetos open source da Apache e em um projeto comercial, e os resultados mostraram que os arquivos que participavam de hotspots apresentavam taxas de bugs e de mudanças significativamente maiores do que os demais. Além disso, ficou evidente que quanto mais hotspots um arquivo apresentava, maior era sua propensão a erros e o esforço necessário para sua manutenção. Isso demonstra que os problemas arquiteturais têm impacto direto na qualidade e nos custos do software, e que detectá-los de forma precoce pode reduzir a dívida técnica acumulada.

Outro ponto importante do trabalho foi a validação prática. Ao apresentar os hotspots identificados para arquitetos de um projeto real, os autores verificaram que a maioria dos problemas apontados correspondia a falhas arquiteturais críticas que não haviam sido detectadas por outras ferramentas. Esse reconhecimento reforça a utilidade e a aplicabilidade do método, mostrando que ele pode apoiar decisões de refatoração e orientar os esforços de melhoria da arquitetura. Além disso, o estudo mostra que, ao contrário dos code smells, que se restringem a trechos isolados de código, os hotspots representam problemas estruturais amplos, com potencial para afetar profundamente a manutenção do sistema.

Por fim, os autores reconhecem algumas limitações de seu trabalho, como a dependência de histórico de mudanças para detectar certos padrões, a sensibilidade dos resultados aos limiares configurados e a necessidade de aplicar a técnica em mais projetos para comprovar sua generalização. Mesmo assim, o artigo representa um avanço significativo ao oferecer uma abordagem sistemática e automatizada para revelar as raízes arquiteturais da dívida técnica. Em síntese, a pesquisa demonstra que a qualidade do software depende não apenas do código individual, mas principalmente da estrutura arquitetural que o sustenta, e que compreender e corrigir os hotspots é essencial para reduzir riscos, custos e retrabalho no desenvolvimento de sistemas complexos.

Um exemplo de aplicação prática desse método pode ser visto em uma empresa que desenvolve um sistema bancário de grande porte, com milhares de arquivos e módulos interligados. Após anos de evolução, o sistema tornou-se difícil e caro de manter, com cada nova mudança causando falhas inesperadas e um grande acúmulo de bugs. Ao aplicar a ferramenta Hotspot Detector, a equipe coleta dados do histórico de commits e das dependências entre os módulos e descobre diversos problemas: uma Unstable Interface na camada de autenticação central, que mudou dezenas de vezes junto com vários outros arquivos; múltiplas Implicit Cross-module Dependencies entre os módulos de transações e relatórios, que deveriam ser independentes; e um grande Cross-Module Cycle envolvendo os módulos de contas, pagamentos e relatórios.

Com essas informações, os arquitetos conseguem priorizar as refatorações mais urgentes, começando pela divisão da interface central em interfaces menores e mais específicas, pela eliminação das dependências ocultas entre os módulos e pela quebra dos ciclos entre módulos. Como resultado, o sistema passa a aceitar mudanças com menor risco e custo, e os índices de bugs caem de forma contínua ao longo do tempo. Esse exemplo evidencia como a detecção de hotspots pode transformar diretamente a manutenção e evolução de um software, reduzindo a complexidade e aumentando sua confiabilidade.