INTEGRAÇÃO DE SISTEMAS LEGADOS

Prof. Rodrigo Cantú Polo

CONVERSA INICIAL

Olá, aluno. Nesta aula, no primeiro tema, será explicado o processo de manutenção de sistemas legados. No segundo, discutidos os custos de manutenção de sistema. O terceiro tema trata da manutenção preventiva de sistemas, considerando a reestruturação/refatoração de código. No quarto tema, é discutida a manutenção preventiva pela reestruturação da arquitetura de um sistema legado; no quinto tema, é descrito o conceito de débito técnico.

CONTEXTUALIZANDO

Nesta aula, você vai estudar conceitos sobre manutenção de sistemas legados, que, como visto nas aulas anteriores, costumam ser sistemas antigos, feitos com tecnologias, muitas vezes, obsoletas e com código degradado pelas mudanças feitas ao longo do tempo. Sua manutenção costuma ser um desafio, pois são sistemas críticos cujo código, em geral, tem baixa qualidade e, ainda assim, devem ser mantidos para correção de defeitos e suporte a novas funcionalidades.

Nas aulas anteriores, foi comentado como grandes sistemas evoluem e como é importante investir esforço na diminuição da queda de qualidade dos sistemas ao longo do tempo. Esse esforço se concentra na fase de manutenção de um sistema legado, que começa na implantação da sua primeira versão, terminando somente quando o sistema chega ao fim do seu funcionamento. Logo, o sucesso de um sistema depende muito do trabalho de manutenção feito para suportá-lo. Se o trabalho for benfeito, o sistema poderá durar muito tempo, atendendo a seus usuários de forma satisfatória e agregando um grande valor ao negócio da organização. Caso contrário, sua qualidade ficará degradada rapidamente, acelerando o fim do seu ciclo de vida. Boas práticas no desenvolvimento de novas funcionalidades e na correção de defeitos são fundamentais para garantir a boa qualidade do sistema.

Muitos dos conceitos a serem vistos nesta aula se aplicam a *softwares* de dispositivos móveis, principalmente temas relacionados à refatoração de código.

TEMA 1 – MANUTENÇÃO DE SISTEMAS LEGADOS

Começa quase que de forma imediata. O sistema é liberado para os usuários e, em questão de dias, relatórios de defeitos são enviados para a equipe

de desenvolvimento. Em semanas, um grupo de usuários indica que o *software* deve ser modificado para que seja possível utilizá-lo de acordo com as suas necessidades. Em meses, outro grupo que não estava interessado no sistema reconhece que o programa pode beneficiá-los de forma inesperada. Eles vão precisar de algumas melhorias para que possam utilizá-lo.

O desafio da manutenção de *software* começou. A equipe de desenvolvimento tem que lidar com uma fila crescente de solicitações de correções de defeitos, pedidos de adaptação e de novas funcionalidades que devem ser planejadas e implementadas. Em pouco tempo, a fila de pedidos de mudança pode crescer muito, e o trabalho envolvido para lidar com as mudanças ameaça sobrecarregar o pessoal disponível para o trabalho. Ao longo do tempo, a organização descobre que está despendendo mais tempo e dinheiro na manutenção de sistemas existentes que no desenvolvimento de novos sistemas.

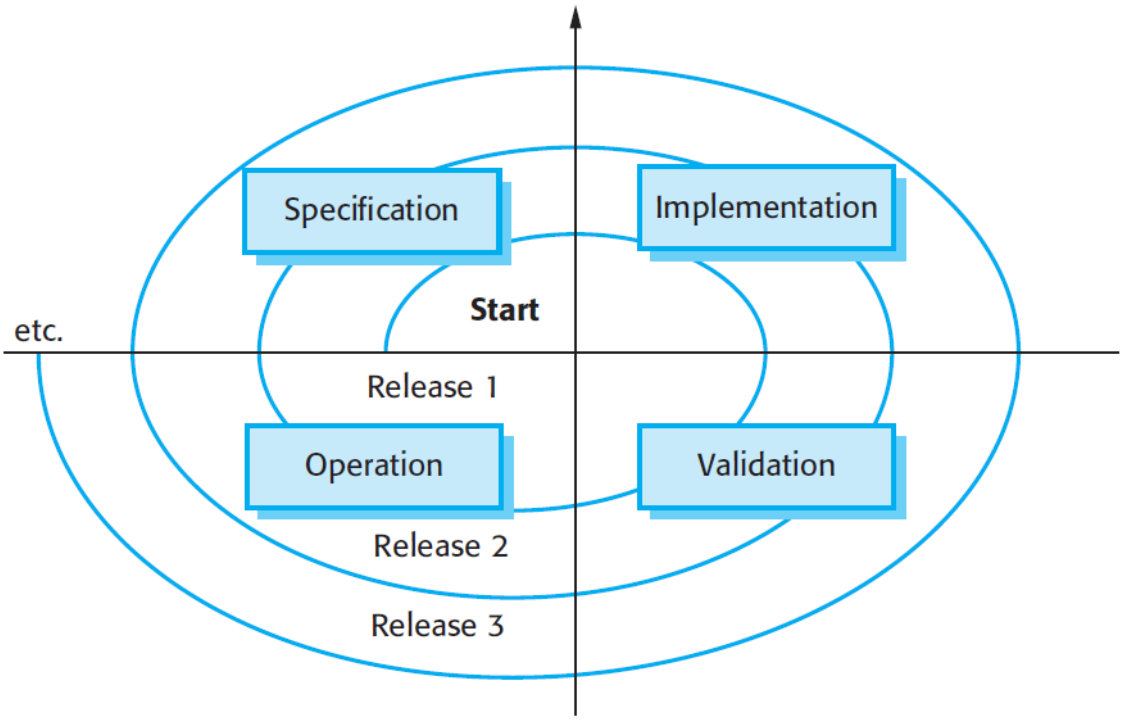
A evolução de sistemas é importante porque organizações investem grandes quantias de dinheiro nos seus sistemas e agora são completamente dependentes deles. Por serem recursos críticos de negócio, elas devem continuar investindo em mudanças para mantê-los relevantes. Consequentemente, grandes organizações gastam mais em sistemas existentes que no desenvolvimento de novos sistemas.

A evolução de um sistema raramente pode ser considerada de forma isolada. Mudanças no ambiente levam a mudanças no sistema, que podem ter como consequências mais mudanças de ambiente. O fato de que os sistemas evoluem em um ambiente complexo geralmente aumenta as dificuldades e os custos de evolução. Além de entender e analisar o impacto de uma mudança proposta no sistema, deve-se verificar como ela pode afetar outros sistemas pertencentes ao mesmo ambiente.

Sistemas importantes costumam ter um longo tempo de vida. Como citado anteriormente, os militares ou de infraestrutura, como os de controle de tráfego aéreo, podem ter um tempo de vida de 30 anos ou mais. Já os corporativos costumam ter mais de 10 anos. E todos custam muito dinheiro. Logo, as organizações utilizam seus sistemas por muitos anos, a fim de terem o retorno de seus investimentos. Obviamente, os requisitos dos sistemas instalados mudam à medida que o negócio e seu ambiente mudam. Como consequência, novas versões dos sistemas, que incorporam mudanças e atualizações, são liberadas em intervalos regulares de tempo.

Deve-se pensar em engenharia de *software* como um processo em espiral, com especificação de requisitos, arquitetura, implementação e testes ao longo do ciclo de vida do sistema (figura 1). No início, é criada a versão 1. Logo que esta é liberada, mudanças são propostas e o desenvolvimento da versão 2 começa imediatamente. De fato, a necessidade de evolução se torna óbvia, mesmo antes de o sistema ser liberado, novas versões podem estar em desenvolvimento antes.

Figura 1 – Modelo espiral de desenvolvimento e evolução



Fonte: Sommerville (2011)

Esse modelo de evolução de *software* implica que uma organização é responsável pelo desenvolvimento inicial de um sistema, assim como sua evolução. A maior parte dos produtos de *software* é desenvolvida de acordo com essa abordagem. Para *software* personalizado, uma abordagem diferente costuma ser utilizada. Uma companhia de desenvolvimento de *software* desenvolve um sistema para um cliente, e a equipe de desenvolvimento do cliente assume o controle das mudanças no sistema. De forma alternativa, o cliente pode fazer um contrato separado para que uma companhia diferente continue o trabalho de evolução e suporte para o sistema.

Nesse caso, é provável que existam descontinuidades no processo em espiral. Requisitos e documentos de arquitetura podem não ter sido repassados de uma companhia para outra. Empresas podem se fundir ou se reorganizar e herdar sistemas de outras empresas, para depois descobrir que esses sistemas

devem ser modificados. Quando a transição do processo de desenvolvimento para o processo de evolução não acontece de forma natural, o processo de manutenção do sistema depois da entrega é chamado, normalmente, de manutenção de *software*.

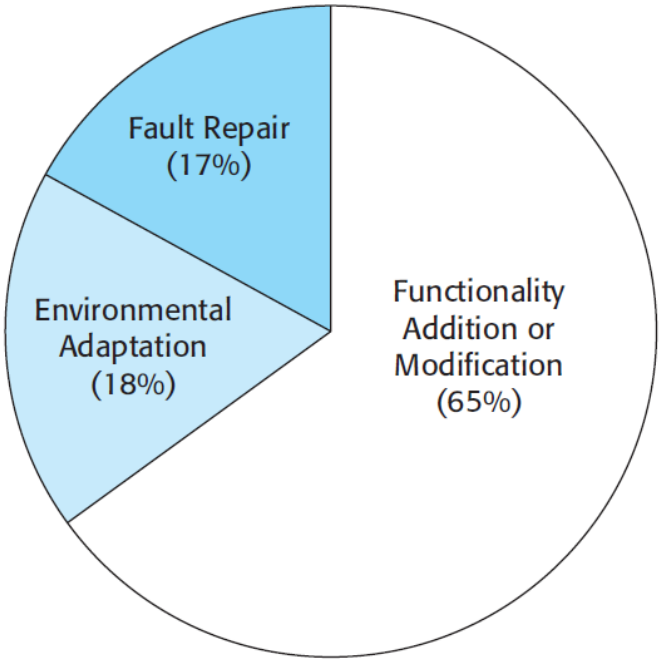
As mudanças são implementadas por modificações nos componentes do sistema e, quando necessário, por adição de novos componentes. Existem quatro tipos de manutenção de *software*, os quais são citados a seguir.

1. **Manutenção corretiva**: atividades de identificação e correção de falhas. Essas atividades compreendem correções de erros de código, de arquitetura e de requisitos. As correções de erros de código costumam ser as mais simples. Já as de arquitetura costumam exigir um esforço maior para correção, pois envolvem mudanças maiores no sistema. As correções relacionadas a erros de especificação de requisitos costumam ser ainda mais trabalhosas, por causa do maior esforço envolvido nas alterações.
2. **Manutenção adaptativa**: esse tipo de manutenção é necessário quando algum aspecto do ambiente do sistema, como o *hardware*, o sistema operacional ou algum outro *software* pertencente à plataforma é modificado. O sistema de aplicação deve ser modificado para se adaptar e suportar essas novas mudanças de ambiente.
3. **Manutenção perfectiva/adição de funcionalidades**: esse tipo de manutenção é necessário quando os requisitos do sistema mudam em resposta a mudanças do negócio ou da organização. A escala das mudanças requeridas costuma ser maior que a de qualquer outro tipo de manutenção.
4. **Manutenção preventiva**: compreende atividades de modificação do sistema para prevenir ocorrência de falhas e para permitir maiores facilidades para modificações posteriores do sistema. Resulta em maior confiabilidade e manutenibilidade.

Estudos feitos por vários pesquisadores do assunto chegaram à conclusão de que o esforço envolvido nas atividades de manutenção de *software* é muito maior que o envolvido nas atividades de desenvolvimento de novos sistemas de *software*. Para Pfleeger (2004), considerando o esforço total de desenvolvimento de *software*, o esforço relacionado à manutenção seria de, aproximadamente, 80%. A figura 2 mostra uma distribuição aproximada dos custos de manutenção. Universalmente, a correção de falhas de sistema não é a atividade que envolve o maior custo. A evolução do sistema para se adaptar a novos ambientes e a

mudanças de requisitos é o que consome a maior parte do esforço de manutenção.

Figura 2 – Distribuição de esforço de manutenção



Fonte: Sommerville (2011)

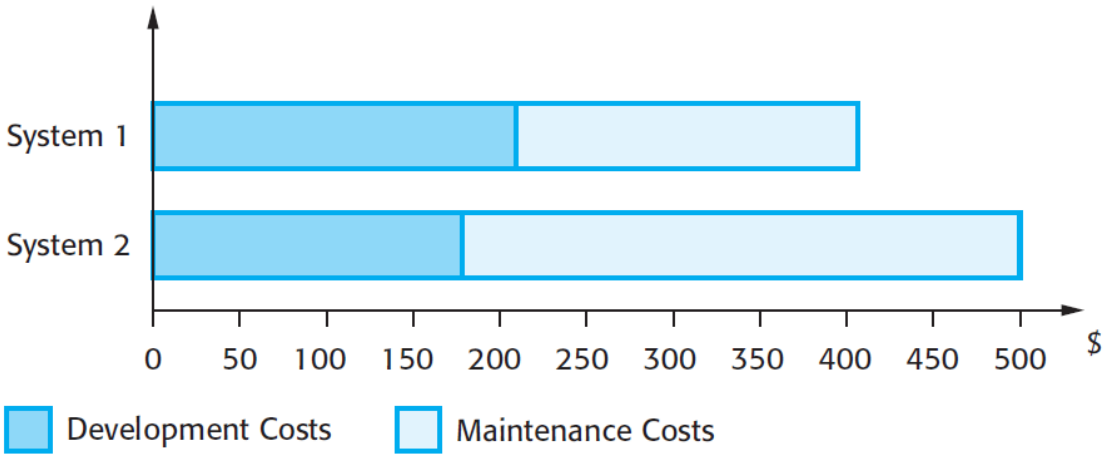
Os custos relacionados à manutenção e ao desenvolvimento de novos sistemas variam de acordo com o domínio de aplicação. Em sistemas corporativos, os custos de manutenção costumam ser comparáveis aos de novos desenvolvimentos. Já em sistemas de tempo real embutidos (desenvolvimento de *firmware*, por exemplo), os custos de manutenção medidos foram de até quatro vezes maiores que os de desenvolvimento. Os requisitos de missão crítica, alta confiabilidade e de desempenho desses sistemas significam que os módulos do sistema devem estar altamente acoplados, o que significa que a mudança nesses sistemas costuma ser mais difícil de ser feita. Apesar das medições referentes às pesquisas citadas anteriormente serem antigas (tem mais de 25 anos), é improvável que as distribuições de custo entre os diferentes tipos de sistema tenha mudado de forma significativa.

O investimento em esforço para planejar e implementar um sistema para reduzir os custos de manutenções futuras costuma ter um bom retorno para as organizações. A inclusão de novas funcionalidades em um sistema, depois que este é liberado para uso, é cara, pois o esforço para aprender como o sistema funciona para analisar o impacto das mudanças propostas é alto. Por esse motivo,

o trabalho feito durante o desenvolvimento para fazer com que o *software* seja fácil de entender e de modificar costuma reduzir o custo da sua evolução. Boas práticas de engenharia de *software*, como a utilização adequada da especificação do sistema, o desenvolvimento orientado a objetos e a gerência de configuração, contribuem para a redução dos custos de manutenção.

A figura 3 mostra como os custos totais de manutenção podem ser diminuídos se um esforço maior for despendido para facilitar a manutenção do sistema. Por causa da redução em custos de entendimento, análise e testes, existe um impacto significativo quando o sistema é desenvolvido para que seja de fácil manutenção. Para o sistema 1, custos de desenvolvimento extra de R$ 25.000,00 são investidos para fazer com que tenha melhor manutenção. O resultado é uma economia de R$ 100.000,00 em custos de manutenção durante todo o tempo de vida do sistema. Esse exemplo mostra que o esforço para fazer com que o sistema tenha uma manutenção mais fácil pode ser comparável à redução de custos totais de manutenção.

Figura 3 – Comparação de custos de desenvolvimento e manutenção



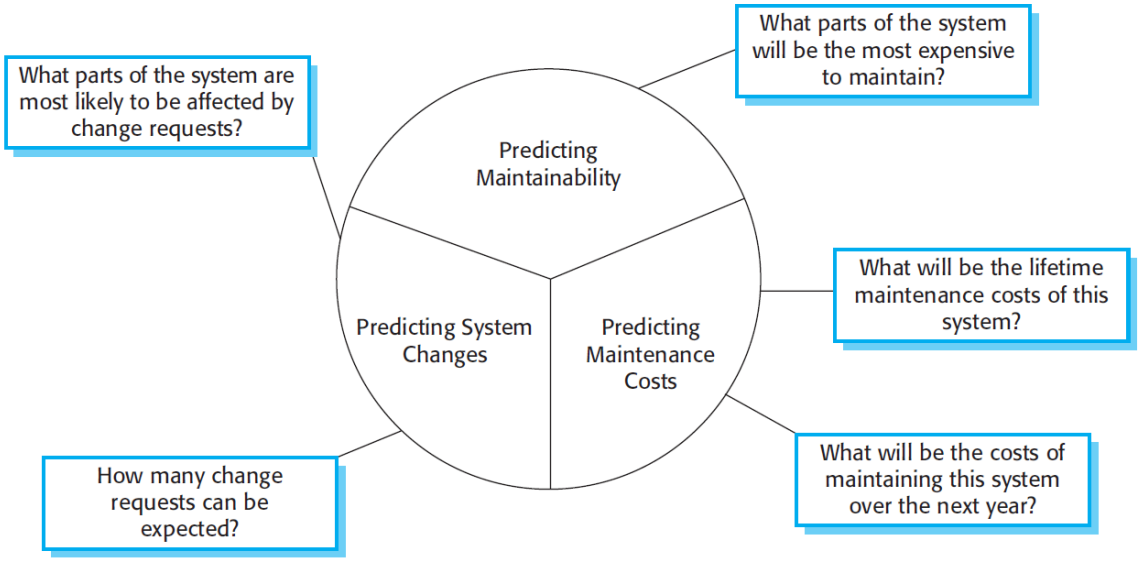
Fonte: Sommerville (2011)

O exemplo da figura 3 é fictício, mas não existem dúvidas de que o desenvolvimento de *software* feito para tornar a manutenção do sistema mais fácil implica em redução de custos de manutenção, quando todo o ciclo de vida do *software* é considerado.

TEMA 2 – PREVISÃO DE CUSTOS DE MANUTENÇÃO

Para que se possa prever os custos de manutenção relacionados a mudanças a serem feitas em um sistema, deve-se prever quais mudanças seriam feitas e quais partes do sistema seriam mais difíceis de manter. Deve-se estimar também os custos totais de manutenção para um sistema em um determinado período. A figura 4 mostra essas questões e os questionamentos relacionados.

Figura 4 – Previsão de manutenção



Fonte: Sommerville (2011)

A previsão do número de pedidos de mudança em um sistema requer um entendimento da relação entre o sistema e seu ambiente externo. Alguns têm um forte relacionamento com seu ambiente externo e é inevitável que mudanças nesse ambiente externo causem mudanças no sistema. Para avaliar o relacionamento entre o sistema e seu ambiente, deve-se avaliar:

- **A quantidade e a complexidade das interfaces do sistema:** quanto maior o número de interfaces e quanto mais complexas elas forem, maior a probabilidade de que mudanças nessas interfaces serão requeridas a medida que novos requisitos forem propostos.
- **A quantidade de requisitos voláteis:** requisitos que refletem políticas e procedimentos organizacionais são mais voláteis que aqueles baseados em características estáveis de domínio da aplicação.
- **Os processos de negócio onde o sistema é utilizado:** à medida que os processos de negócio evoluem, mudanças no sistema são necessárias para acomodar novos pedidos de mudança. Quanto mais processos de

negócio utilizam o sistema, maior será a quantidade de pedidos de mudança.

Por muitos anos, vários pesquisadores analisaram os relacionamentos entre a complexidade de programas, medidas por métricas como complexidade ciclomática e de manutenibilidade. A conclusão desses estudos foi de que, quanto mais complexo for um sistema ou um componente, maiores serão os custos da sua manutenção. Logo, para reduzir os custos de manutenção, devem-se substituir componentes complexos do sistema por alternativas mais simples.

Depois que um sistema é implantado, os dados relacionados a sua execução devem ser utilizados para que se possa prever o grau de complexidade de manutenção. Exemplos de métricas que podem ser utilizadas para esta análise são as seguintes:

- **Quantidade de pedidos de manutenção corretiva:** um aumento no número de falhas pode indicar que mais erros foram introduzidos que corrigidos durante o processo de manutenção no sistema. Isso pode indicar um declínio na qualidade do processo de manutenção.
- **Tempo médio requerido para análises de impacto:** indica a quantidade de componentes de *software* que são afetados por pedidos de mudança. Se esse tempo aumenta, significa que mais componentes estão sendo afetados, diminuindo a qualidade da manutenção.
- **Tempo médio requerido para implementar um pedido de mudança:** esse tempo é diferente do de análise de impacto, apesar de ser possível correlacionar as duas medidas. Esse é o tempo necessário para modificar o sistema e sua documentação, depois que os componentes relacionados à mudança foram identificados. Um aumento no tempo necessário para implementar a mudança pode indicar um declínio na sua manutenibilidade.
- **Quantidade de pedidos de mudança abertos:** um aumento nesse número durante um período de tempo pode indicar uma queda na qualidade da sua manutenção.

As informações derivadas dos pedidos de mudança e as previsões sobre a manutenibilidade do sistema podem ser utilizadas para realizar uma previsão sobre os custos de manutenção.

TEMA 3 – MANUTENÇÃO PREVENTIVA POR REESTRUTURAÇÃO DE CÓDIGO

O processo de reestruturação de código, ou refatoração, consiste em desenvolver melhorias em um programa para que sua degradação com o tempo diminua. Significa modificar um programa para melhorar sua estrutura, reduzir sua complexidade, ou para melhorar seu entendimento. O processo de refatoração é, algumas vezes, interpretado como sendo limitado a sistemas orientados a objetos, mas seus princípios podem ser aplicados a qualquer paradigma de desenvolvimento. Quando um programa é refatorado, não se deve adicionar novas funcionalidades, e sim concentrar apenas em melhorias. A refatoração pode ser interpretada como um tipo de manutenção preventiva.

Reengenharia e refatoração são processos feitos para fazer com que um *software* seja mais fácil de manter e de compreender, sendo diferentes entre si. A reengenharia pode ser feita depois que um sistema foi mantido por um certo tempo e seus custos de manutenção estão aumentando. A refatoração é um processo contínuo de melhorias durante o desenvolvimento e o processo de evolução do *software*. Serve para evitar a degradação deste, o que aumenta os custos e as dificuldades de manutenção.

Refatoração é uma parte importante das metodologias de desenvolvimento ágil, pois são feitas para acomodar mudanças. A qualidade de programas tende a se degradar rapidamente. Por esse motivo, desenvolvedores que seguem essas metodologias frequentemente refatoram seus programas a fim de evitar essa degradação. A ênfase em testes de regressão em metodologias ágeis diminui o risco de introduzir novos defeitos por meio da refatoração. Qualquer erro introduzido deve ser detectável, pois testes que antes funcionavam poderão falhar em virtude do erro inserido. Todavia, a refatoração não é uma atividade exclusiva de metodologias ágeis, podendo ser utilizada por qualquer outra metodologia de desenvolvimento.

Fowler (1999) sugere que existem situações, chamadas por ele de “*bad smells*” (cheiros ruins), em que o código deve ser melhorado. Exemplos desses casos são citados a seguir.

- **Código duplicado:** códigos iguais, ou muito similares, podem ter sido incluídos em partes diferentes do programa. Esses códigos podem ser removidos e implementados em um único método ou função, que pode ser chamada no lugar desses códigos.
- **Métodos longos:** se um método for muito longo, este deverá ser refeito em um conjunto de métodos menores.

- **Aglomerados de dados:** agrupamentos de dados ocorrem quando o mesmo grupo de dados (campos em classes, parâmetros em métodos) ocorre em várias partes diferentes de um programa. Esses dados podem ser substituídos por objetos, que os encapsulam.
- **Condições *switch-case*:** podem estar espalhadas pelo programa avaliando uma mesma condição repetidas vezes. Em linguagens orientadas a objetos, polimorfismo pode ser utilizado no lugar desse tipo de avaliação (Sommerville, 20011).

Existem muitos outros "*smells*" que podem ser vistos em sourcemaking.

Atividades de refatoração, feitas durante o desenvolvimento de um sistema, são um meio efetivo de se reduzir os custos em longo prazo de manutenção. Porém, se uma atividade de manutenção é feita em um sistema cuja estrutura tenha sido degradada significativamente, pode ser praticamente impossível de se efetuar a refatoração.

A refatoração de arquitetura também deve ser considerada, que pode ser um problema mais difícil de ser resolvido, já que envolve a substituição do código existente no sistema por padrões de projeto relevantes, melhorando a qualidade do *software*.

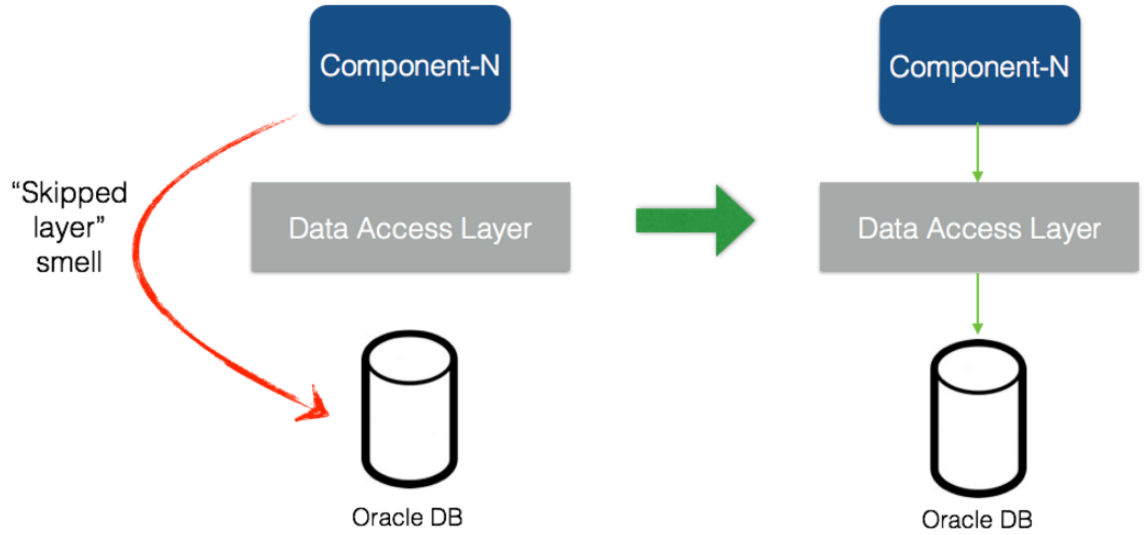
TEMA 4 – MANUTENÇÃO PREVENTIVA POR REESTRUTURAÇÃO DE ARQUITETURA

Reestruturação, ou refatoração de arquitetura, consiste em resolver problemas relacionados a falhas da própria arquitetura, ou da falta do seu seguimento pelos desenvolvedores. São problemas que têm um impacto negativo por todo o sistema.

Um exemplo de problema relacionado à arquitetura seria o seguinte: considere que um dos componentes arquiteturais de um sistema é uma camada de acesso a dados (DAL – *Data Access Layer*). A decisão de arquitetura feita anteriormente é a de que todos os acessos ao banco de dados devem ser feitos pela DAL. No plano de projeto, consta que deve haver suporte para mais de um banco de dados, como o MySQL e o SQL Server. Porém, os desenvolvedores introduziram várias consultas SQL que acessam o banco de dados diretamente, não utilizando a camada DAL por negligência ou por problemas de transferência de conhecimento de decisões de arquitetura. Esse problema arquitetural afeta a flexibilidade da arquitetura em suportar novos bancos de dados, afetando

negativamente a qualidade do sistema. O time do projeto deve, em algum momento, refatorar as consultas SQL e direcionar as chamadas à DAL quando o suporte aos diferentes bancos de dados for realmente necessário. A figura 5 ilustra esse problema de arquitetura.

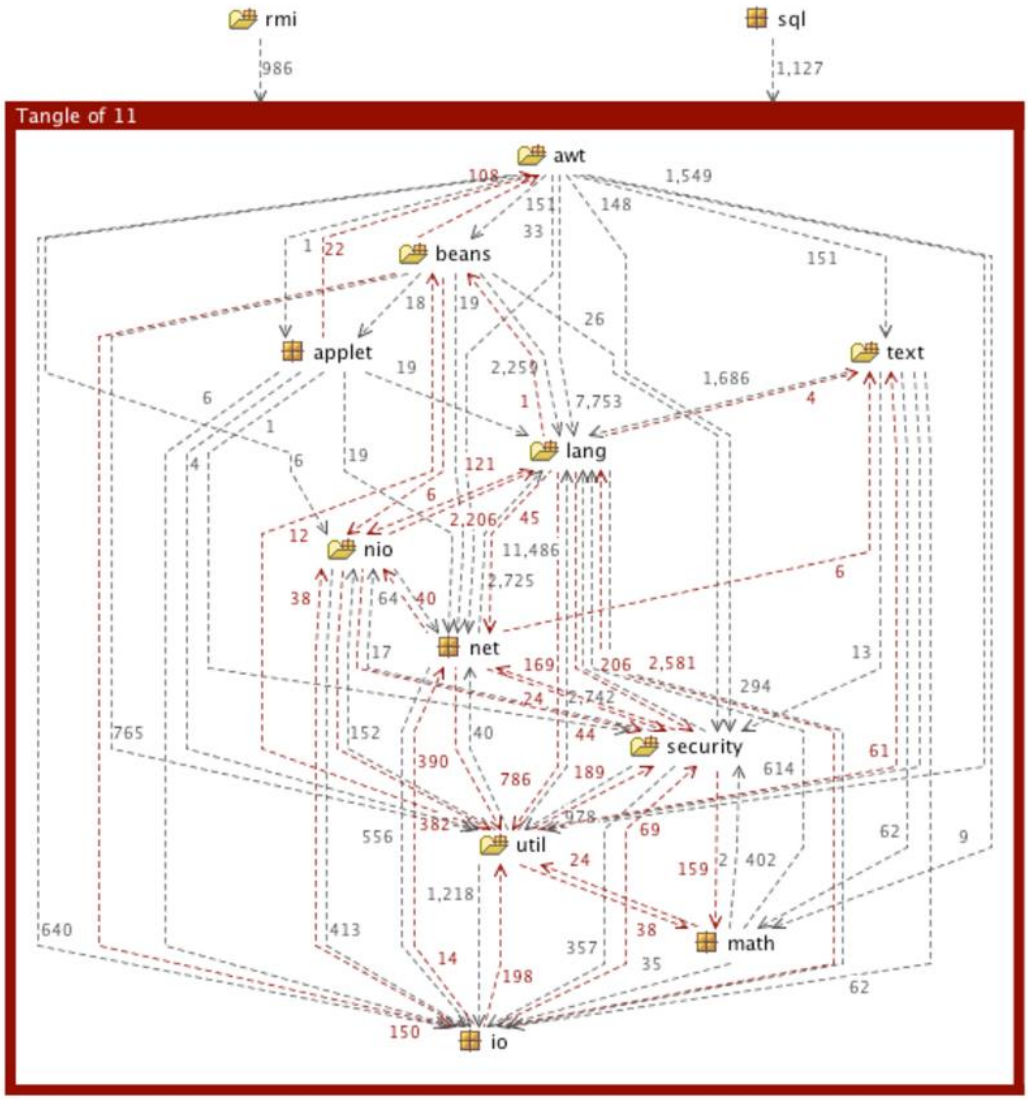
Figura 5 – Refatoração de problema de não utilização de camada



Fonte: Designsmells (2016)

Outro problema de arquitetura muito comum é o de dependência cíclica entre pacotes, como no pacote java.lang da linguagem Java. Esse é um problema que afeta a modularidade do sistema, dificultando a reutilização de componentes. Um componente de nível mais baixo, que poderia ser utilizado em outro sistema, fica impossibilitado de ser reutilizado, pois depende de classes de nível mais alto, gerando uma dependência mútua (cíclica) entre classes. As dependências cíclicas podem ser observadas nos relacionamentos em vermelho na figura 6.

Figura 6 – Dependências cíclicas entre as classes do pacote java.lang



Fonte: Designsmells (2016)

Em resumo, problemas de arquitetura impactam negativamente na qualidade geral do sistema. Refatoração de arquitetura serve para preservar a utilização correta da arquitetura do sistema, melhorando a facilidade de manutenção e mantendo o débito técnico de arquitetura sobre controle.

TEMA 5 – DÍVIDA TÉCNICA

A expressão “débito técnico”, ou “dívida técnica”, descreve a dívida que uma equipe de desenvolvimento assume quando escolhe uma abordagem fácil de implementar um projeto de *software* no curto prazo, mas com grande impacto negativo de qualidade no longo prazo.

Fowler (1999) sugeriu a seguinte definição para a dívida técnica:

> A dívida técnica é similar à dívida financeira. Assim como a dívida financeira, a dívida técnica exige o pagamento de juros. Estes vêm na forma de esforço extra, que devem ser pagos em desenvolvimentos

> futuros por conta da escolha de um design mais rápido e de baixa qualidade. Nós podemos optar por continuar pagando estes juros ou quitar de uma vez a dívida fazendo uma refatoração, transformando um design de baixa qualidade em um design melhor. Apesar dos custos para saldar a dívida, ganhamos reduzindo os juros no futuro.

A dívida técnica pode ser classificada em dois tipos:

1. **Sem querer:** desenvolvedores sem experiência escrevem código de baixa qualidade por causa de sua inexperiência técnica;
2. **Intencional:** a equipe faz uma decisão consciente para otimizar o desempenho de desenvolvimento de *software* para o momento atual e não para o futuro, fazendo algumas escolhas que podem ser uma maneira rápida e de baixa qualidade para resolver a situação.

Existe também a definição de bagunça técnica, onde desenvolvedores escrevem um código de baixa qualidade por preguiça e falta de profissionalismo. Esse tipo de débito técnico sempre consiste em perdas, pois é o pior e mais difícil tipo de dívida técnica a ser revertido.

De acordo com outra classificação, a bagunça técnica também pode ser interpretada como uma dívida irresponsável, em que uma dívida prudente seria baseada em uma situação pensada. Veja, abaixo, essa classificação mais detalhada.

- **Irresponsável e proposital:** o time não tem tempo para desenvolver o *software* com qualidade, utilizando uma solução rápida e com pouca preocupação com a qualidade.
- **Prudente e proposital:** o time precisa entregar o produto o mais rápido possível, com todas as limitações conhecidas e assume, de maneira proativa, as consequências.
- **Irresponsável e sem querer:** o time não tem consciência dos princípios básicos de código com boa qualidade nem conhecimento da bagunça que está sendo feita.
- **Prudente e sem querer:** aplica-se a times com excelentes arquitetos/desenvolvedores. Eles fornecem uma solução que agrega valor ao negócio, mas, depois de completar a solução, entendem que a abordagem implementada poderia ter sido melhor (HAZRATI, 2009].

Dessa forma, ter uma dívida técnica em um projeto é normalmente inevitável, devendo ser considerada como uma expectativa. A chave é ter certeza de que o time não está introduzindo dívidas irresponsáveis, que contribuem para bagunçar o código e que são muito difíceis, ou impossíveis, de lidar (Hazrati, 2009).

FINALIZANDO

Nesta aula, foram abordadas a manutenção de sistemas legados, os tipos de manutenção e a distribuição do esforço de manutenção em uma organização. Também foram abordados os indicadores de custos de manutenção, a manutenção preventiva por reestruturação de código e de arquitetura, além dos conceitos de débito técnico.

REFERÊNCIAS

DESIGNSMELLS. **Refactoring for design smells**: an introduction. 25 abr. 2016. Disponível em: <http://www.designsmells.com/articles/refactoring-for-architecture-smells-an-introduction/>. Acesso em: 8 out. 2017.

FOWLER, M. et al. **Refactoring**: improving the design of existing code. Boston: Addison-Wesley, 1999.

HAZRATI, V. **Analisando a dívida técnica**. 26 out. 2009. Disponível em: <https://www.infoq.com/br/news/2009/10/dissecting-technical-debt>. Acesso em: 8 out. 2017.

PFLEEGER, S. L. **Engenharia de software**: teoria e prática. 2. ed. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. **Engenharia de software**: uma abordagem profissional. 7. ed. Porto Alegre: AMGH, 2011.

SOMMERVILLE, I. **Engenharia de software**. 9. ed. São Paulo: Pearson Education, 2011.

SOURCEMAKING. Disponível em: <https://sourcemaking.com/>. Acesso em: 8 out. 2017.