Capítulo 4

Arquitetura baseada em componentes

Com o aumento da complexidade de um projeto e a necessidade de entregar soluções cada vez mais rápidas à área de negócio e acompanhar, assim, a necessidade do mercado, o reaproveitamento de funcionalidades tornou-se primordial. O primeiro passo para tratar o reaproveitamento de códigos foi a migração de programas escritos em linguagens estruturais para um novo paradigma de programação, a orientação a objetos (SOMMERVILLE, 2011).

Contudo, o reaproveitamento que se obteve ao utilizar esse paradigma não foi tão efetivo quanto o esperado. As classes, como unidades de códigos-fontes, não conseguiam ser comuns o suficiente para atender a diferentes contextos. Sendo assim, muitas vezes houve a necessidade de refazer determinada classe, encaixando-a melhor à solução entregue (ENGHOLM JUNIOR, 2017; SOMMERVILLE, 2011).

Essa foi a motivação para a criação de componentes de softwares, o que, por consequência, gerou a necessidade de se trabalhar em arquiteturas para que estes fossem facilmente conectados em uma sequência lógica, atendendo à necessidade do negócio.

1 Visão geral sobre componentes

Quando olhamos para o cenário da programação hoje, com tantas linguagens de alto nível e técnicas avançadas, não conseguimos imaginar como os sistemas eram escritos anteriormente. Entre as décadas de 1950 e 1960, programar era uma grande burocracia. Cada programa escrito informava em sua primeira linha o endereço de memória que deveria ocupar. A comunicação entre os desenvolvedores sempre foi crucial para o sucesso de um projeto, mas naquela época era mais ainda. Se dois desenvolvedores escolhessem o mesmo endereço de memória para uma concorrência de programas, isso era motivo de falha do sistema. Para reaproveitar códigos, era preciso criar bibliotecas de código-fonte. Então, para reaproveitar o código já escrito por alguém, o que se fazia era basicamente copiar e colar o trecho de código necessário (MARTIN, 2018).

Depois disso, foi possível criar pequenos projetos que eram colocados em memória e chamados por meio de outros programas. As bibliotecas de códigos tornaram-se digitais, efetivamente. Contudo, os endereços de memória ainda eram compartilhados, portanto intervalos de endereços tinham que ser reservados para cada parte do programa. Geralmente,

acordos eram feitos com o objetivo de determinar quais endereços cada parte do sistema iria ocupar. Com o passar do tempo, apareceram os programas responsáveis por criar os vínculos entre cada uma das bibliotecas e aplicações antes da compilação do programa. Porém, segundo Martin (2018), esse processo de compilação durava em média uma hora para conseguir gerar o executável do sistema.

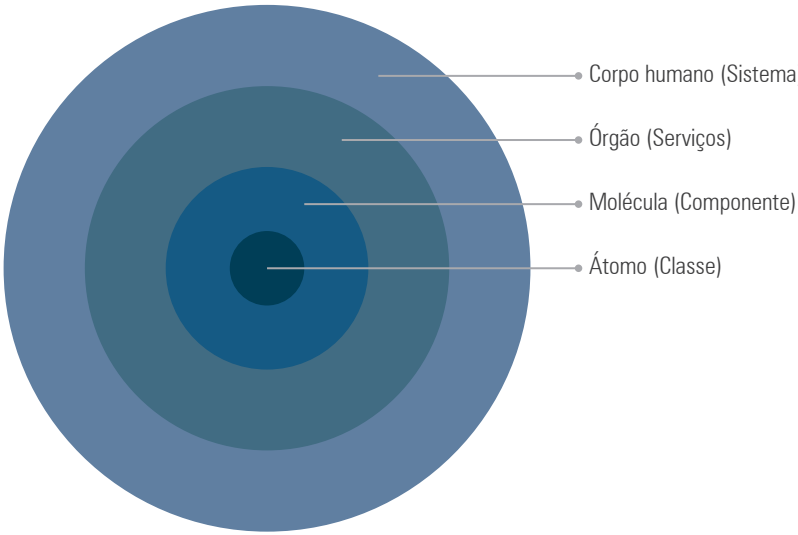
Em meados dos anos 1990, o poder computacional atingiu o nível em que esse processo de vínculo passou a poder ser feito no momento da execução do programa. Isso abriu portas para uma nova era na maneira de construir e organizar os programas. Hoje, as bibliotecas são enviadas para diversos sistemas conectarem-se via rede, como se fossem pequenos plugins em nosso sistema. Mas como isso foi possível?

No início dessa nova era, acreditou-se que a programação orientada a objetos, com a construção de classes, seria o suficiente para construir os sistemas de maneira a atingir uma reutilização máxima em relação ao que já havia sido construído anteriormente (SOMMERVILLE, 2011; MARTIN, 2018). Contudo, o acoplamento das classes exigia um vasto conhecimento dos detalhes de implementações para que a ideia de reutilizar o código fosse possível. Além disso, existe uma necessidade, em alguns casos, de realizar uma grande implementação para adaptar determinada classe a um cenário específico. Isso significa que, ao analisarmos objetos como elementos independentes, torna-se impossível o reúso destes em diversos sistemas. Nesse momento, nasce o conceito de componentes de softwares (SILVEIRA *et al*., 2011; SOMMERVILLE, 2011).

Componentes são abstrações de mais alto nível quando comparados a objetos. Eles devem ser a menor unidade que pode ser acoplada ao sistema, e cada um deve possuir uma interface para facilitar seu acoplamento. Encontramos diversos componentes nas linguagens de programação: em Java, os arquivos *jar*; em Ruby, os arquivos *gem*.; o .Net trata seus componentes como DLLs. Podemos, ainda, ter scripts que são componentes, nos casos de linguagens interpretadas como JavaScript

e Python; e arquivos binários, para os casos de linguagens totalmente compiladas, como o C e o C++ (SOMMERVILLE, 2011; MARTIN, 2018). Para entendermos melhor como o componente se encaixa no desenvolvimento, acompanhe a analogia ilustrada na figura 1.

Figura 1 – Analogia das unidades de um sistema



Na figura 1, temos uma analogia entre as unidades encontradas no corpo humano e as unidades encontradas em sistemas. No círculo central, de menor tamanho, temos o átomo, sendo a menor unidade encontrada em um corpo humano. O átomo representa a classe criada para aquele problema. Em seguida, temos a molécula, um conjunto de átomos; do mesmo modo, um conjunto de classes cria um componente. O próximo círculo representa os órgãos, que em um sistema seria o conjunto de componentes, os quais, quando acoplados para trabalhar em conjunto, são chamados de serviços. O conjunto de serviços trazem como resultado o sistema, assim como o conjunto de órgãos nos guia para a totalidade do corpo humano.

A partir desse momento, podemos começar a pensar em uma arquitetura que se baseia em componentes. Para isso, é necessário considerar o processo de definir, implementar, integrar ou criar componentes de maneira independente. Assim, é possível manter um baixo acoplamento com sistemas específicos. Esse tipo de abordagem é uma boa saída para sistemas que se tornam complexos e maiores a partir de sua evolução temporal (SOMMERVILLE, 2011).

Se o componente criado é uma peça independente, podemos fazer pequenas peças para conectá-las de acordo com as implantações de diferentes sistemas. Dessa forma, podemos começar a gerar resultados mais rápidos para o cliente (SILVEIRA *et al*., 2011; SOMMERVILLE, 2011).

Utilizar uma arquitetura baseada em componentes pode ajudar a evitar falhas do sistema. Caso um componente dê problema durante a implantação, basta desativá-lo e o sistema continuará funcionando sem transtornos. Assim, é só realizar a manutenção no componente e acoplá-lo novamente. Voltando à analogia do corpo humano, é como se você pegasse um leve resfriado.

2 Princípios e uso de componentes

Ao projetar componentes, é importante ter em mente que eles devem ser softwares passíveis de manutenção e que tenham como princípio a fácil compreensão. Eles devem ser independentes, realizar comunicação por meio de interfaces bem definidas e possuir uma gama de funcionalidades padrão (SOMMERVILLE, 2011).

Toda essa ideia por trás do reúso de um componente como uma versão estendida de objetos começou com a distribuição dos sistemas. O conceito de sistemas distribuídos foi apresentado em alguns frameworks, sendo os protagonistas dessa implementação o CCM, da CORBA; o COM e o .Net, da Microsoft; e o Enterprise Java Beans (EJB),

do Java, que até então era controlado pela Sun Microsystems.[1] Esse início do uso de componentes como sistemas acoplados era um pouco trabalhoso para o desenvolvedor. A facilidade na criação de componentes só começou quando eles passaram a ser vistos como serviços (GONÇALVES, 2007; SOMMERVILLE, 2011).

Houve uma padronização entre os protocolos de comunicação, o que facilitou a conversa entre cada componente, sem a necessidade de uniformidade em uma única tecnologia (SOMMERVILLE, 2011).

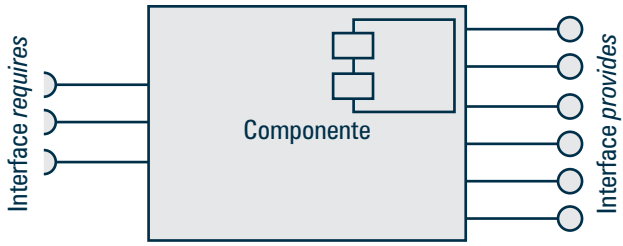
Para seu adequado funcionamento, o componente deve ser:

- **Padronizado:** precisa obedecer a um modelo padrão básico de componentes e definir as interfaces de entrada e saída, metadados, documentação, composição e implantação.
- **Independente:** não deve necessitar de outros componentes para ser funcional.
- **Passível de composição:** as interfaces para acesso externo devem estar bem claras e muito bem definidas.
- **Implantável:** deve ser autocontido, capaz de atuar como uma entidade autônoma.
- **Documentado:** para facilitar o uso de potenciais usuários.

A figura 2 representa graficamente um componente.

1 Empresa norte-americana, fabricante de computadores, adquirida em 2010 pela Oracle.

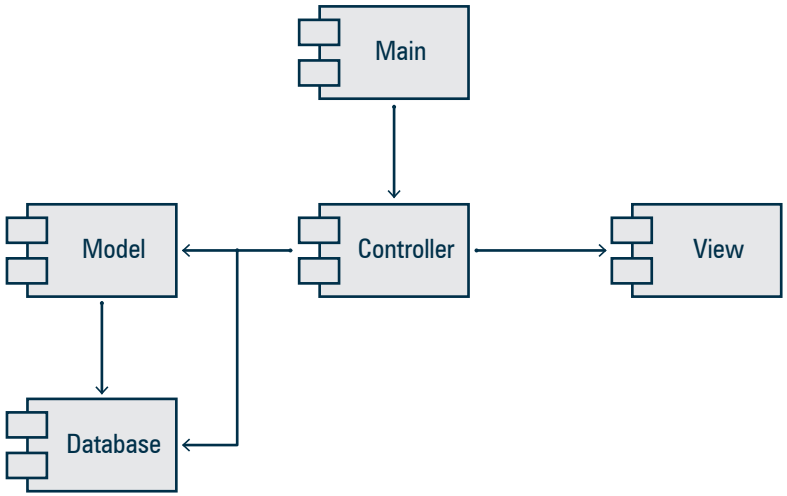
Figura 2 – Interfaces de um componente



Basicamente, o componente possui uma interface com um conjunto de métodos *requires*, que é a parte onde ele recebe a informação de que necessita para trabalhar. Cada método é representado por uma meia-lua em aberto, como se esperasse algum mecanismo para encaixar o outro componente. A interface *provides* é o resultado do processamento de um componente e se encaixa nos métodos *requires* do componente de interação.

Apesar de nosso foco serem componentes de sistemas back-end, hoje em dia os frameworks focados em front-end têm ganhado muita força no mercado. Tanto os frameworks focados no mercado móvel quanto os focados na web possuem uma arquitetura baseada em componentes. No caso dos frameworks front-end, como Angular, React, Vue, React Native e Flutter, é mais fácil conseguir identificar um componente, pois é mais simples conseguirmos visualizá-lo de forma concreta na interface gráfica do sistema. Mas e como podemos visualizar os componentes em uma aplicação back-end, uma vez que estamos falando basicamente de um código? Confira uma arquitetura MVC em formato de componentes na figura 3.

Figura 3 – Modelo de arquitetura MVC baseada em componentes



Perceba, na figura 3, que possuímos cinco componentes: main, model, controller, view e database. Cada um deles representa um pacote que é desenvolvido no formato de componente. Essa é uma abordagem para trabalhar com componentes. A outra é cada entidade ser um componente, de modo a conter todos os métodos das camadas MVC. Assim, o relacionamento continua de forma independente em cada parte do sistema. A melhor maneira de adotar o componente vai depender do cenário e da aplicação que está sendo construída. O importante é pensar em pedaços de sistema que podem funcionar de maneira independente e ao mesmo tempo ser integrados com qualquer outro sistema ou componente.

Considerações finais

Neste capítulo, apresentamos o conceito de arquitetura baseada em componentes. Com esse tipo de abordagem, podemos deixar o sistema dividido em diversas partes independentes e, a partir desse ponto,

construir cada vez mais funcionalidades sem nos preocuparmos com a falha geral do sistema. Pense sempre no seu código como pequenas partes, que, por mais independentes que sejam, juntas podem construir um sistema mais robusto.

Referências

ENGHOLM JUNIOR, Hélio. **Análise e design orientados a objetos**. São Paulo: Novatec, 2017.

GONÇALVES, Edson. **Desenvolvendo aplicações web com JSP, Servlets, JavaServer Faces, Hibernate, EJB 3 Persistence e Ajax**. Rio de Janeiro: Ciência Moderna, 2007.

MARTIN, Robert C. **Clean architecture**: a craftsman's guide to software structure and design. [*S. l.*]: Pearson Education, 2018.

SILVEIRA, Paulo *et al*. **Introdução à arquitetura de design de software**: uma introdução à plataforma Java. Rio de Janeiro: Elsevier Brasil, 2011.

SOMMERVILLE, Ian. **Engenharia de software**. 9. ed. São Paulo: Pearson; Boston: Addison-Wesley, 2011.