Unidade III

5 PROJETOS DE DADOS E CLASSES E PROJETO ARQUITETURAL

5.1 Projeto de dados e classes

Como vimos na Unidade I, a fase de projetos é organizada em quatro fases: projeto de dados, projeto arquitetural, projeto de interfaces e projeto de componentes.

Lembrete

A subdivisão das atividades na fase de projeto é delimitada pelos artefatos que produzem.

Na Unidade II, tratamos do processo da passagem da fase de análise para a fase de projetos, ou seja, entramos efetivamente na fase de projetos, na atividade de projeto de dados e classes, com a produção do modelo de classes de projeto, realizado a partir do refinamento do diagrama de classes do domínio.

A partir de agora, trataremos única e exclusivamente das tarefas técnicas de projeto, tendo como base o modelo de classes de projeto e dos aspectos dinâmicos do projeto de objetos mapeados até aqui.

Segundo Pressman (2006, p. 208), a fase de projetos de dados tem como objetivo a geração do modelo de dados e a transformação de classes e objetos conceituais em classes e objetos equivalentes em projeto.

Até este momento atingimos a parte de transformação de classes e objetos conceituais em classes e objetos equivalentes em projeto; entraremos, agora, na modelagem de dados.

5.1.1 Introdução ao projeto de dados

O projeto de dados, ou modelagem de dados, tem como objetivo definir uma estrutura de informações necessárias para implementar o sistema de *software* (PRESSMAN, 2006).

Em linhas gerais, devemos montar uma estrutura para armazenar, atualizar e recuperar as informações do sistema.

Nesta fase, como estamos tratando de aspectos de construção do projeto, devemos determinar a organização, métodos de acesso, associações e alternativas de processamento dessas informações.

Como vimos, boa parte dessas atividades é feita pelo SGBD, que é o responsável por prover os mecanismos de acesso, gerenciamento e processamento das informações em um repositório de dados.

Todavia, fica a cargo do projeto a adoção de um SGBD que satisfaça as necessidades do sistema de *software*, e faz parte das atribuições técnicas do arquiteto o conhecimento das tecnologias disponíveis a serem utilizadas.

Nesta fase daremos ênfase à organização das informações e em como representar um dicionário de dados que dê suporte à resolução do problema pelo sistema de *software*.

Projetar um banco de dados também não é uma tarefa que pode ser realizada sem organização.

O processo de projeto de banco de dados pode ser subdividido em três fases: projeto conceitual, projeto lógico e projeto físico, como mostra a figura a seguir (ELMASRI; NAVATHE, 2011).

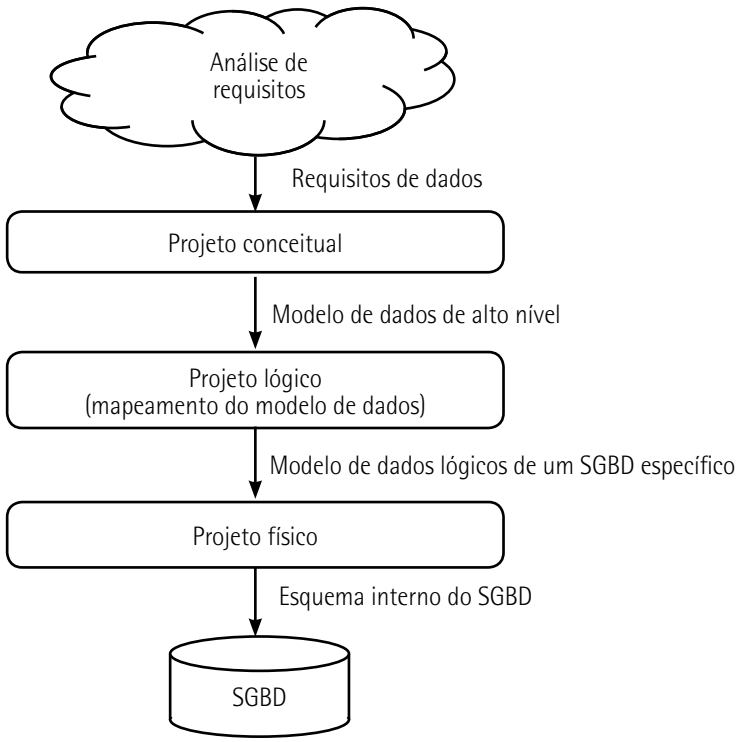


Figura 19 – Projeto de banco de dados

- Projeto conceitual: tem como objetivo produzir um modelo de alto nível, ou seja, sem detalhes específicos de um SGBD, que represente os requisitos de dados. Esses requisitos expressam tudo aquilo que deverá ser armazenado pelo sistema, e o modelo de alto nível pode ser utilizado como instrumento de validação com a própria área usuária.

- Projeto lógico: essa fase do projeto utiliza como base o modelo produzido no projeto conceitual e monta o modelo de dados de implementação, que contempla detalhes que possam tornar esse modelo implementável em um SGBD específico.

- Projeto físico: é a implementação do modelo lógico em uma linguagem de programação que possa fazer a transação do modelo lógico para o esquema interno do SGBD. É a concretização do projeto de dados.

Atualmente, como já mencionamos, o modelo de dados relacional é um dos modelos mais utilizados quando falamos em paradigma de projetos de banco de dados.

Saiba mais

Aprofunde seus conhecimentos sobre outros modelos de banco de dados, como o modelo orientado a objetos, em:

ELMASRI, R.; NAVATHE, S. B. *Sistemas de banco de dados*. 6. ed. São Paulo: Addison-Wesley, 2011.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. *Sistema de banco de dados*. 5. ed. Rio de Janeiro. Campus, 2006.

5.1.2 Introdução a banco de dados relacionais

O modelo entidade relacional enxerga os dados do mundo real como o conjunto: entidade, atributos e relacionamento. Cada entidade, ou um conjunto de entidades, gera uma tabela, seus atributos ou características são representados por colunas desta tabela e cada linha desta tabela representa uma instância dessa entidade.

A figura a seguir mostra um exemplo de um modelo conceitual, de alto nível, produzido na fase conceitual do projeto de banco de dados, e o quadro na sequência explica o significado de cada um desses elementos.

Suponhamos que o cenário de negócio a ser modelado seja a inscrição de alunos para cursar disciplinas. Nesse caso fictício, um aluno pode não estar inscrito em nenhuma disciplina ou pode estar inscrito em várias (sem limite preestabelecido), enquanto uma disciplina pode não ter nenhum aluno inscrito ou possuir no máximo quarenta inscrições.

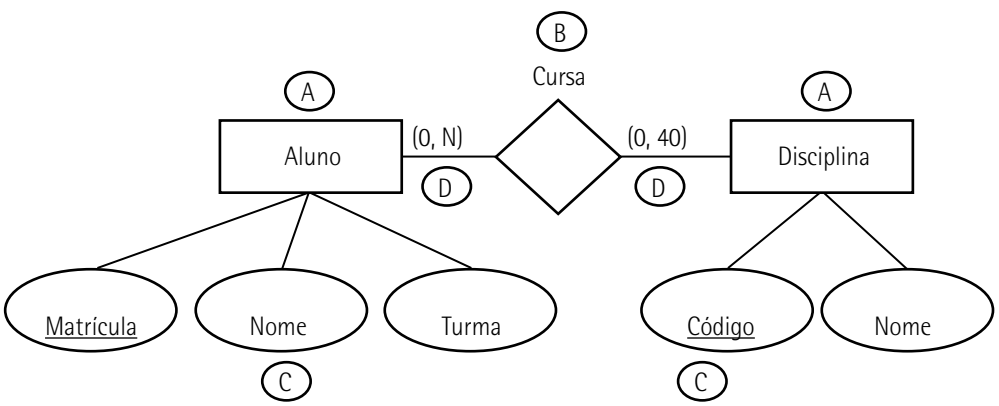


Figura 20 – Modelo Conceitual E-R

Quadro 7 – Elementos do modelo conceitual E-R

Letra	Descrição
A	Entidades: representação de algo do mundo real.
B	Relacionamento: indicação do relacionamento entre duas entidades. No exemplo, a relação entre "Aluno" e "Disciplina" é "Cursa". A leitura que podemos fazer deste modelo é: Aluno(s) cursa(m) Disciplina(s) e Disciplina(s) é(são) cursada(s) por Aluno(s).
C	Atributos: conjunto de características de uma entidade. No exemplo, podemos fazer as seguintes leituras: - Aluno possui: Matrícula, Nome e Turma; - Disciplina possui: Código e Nome; Obs: note que os atributos "Matricula" em "Aluno" e "Codigo" em "Disciplina" estão sublinhados. Esses atributos são "Chaves Primárias" de suas entidades; isso quer dizer que esses campos são identificadores únicos de uma entidade, ou seja, um aluno nunca poderá possuir uma matrícula igual ao outro e um código de disciplina nunca poderá ser repetido. Os conceitos de chave primária e identificadores únicos, em banco de dados, garantem a identidade e a integridade referencial.
D	Multiplicidade: denota a proporção, ou a quantificação, do relacionamento entre duas entidades. No caso do exemplo, fazemos as seguintes leituras: - Um aluno cursa no mínimo zero disciplina e no máximo N (sem número preciso) disciplina(s). - Uma disciplina é cursada por no mínimo zero aluno e no máximo quarenta. Existem as seguintes representações de multiplicidade: (0,1): lê-se zero para um; (1,1): lê-se um para um; (0,N): lê-se zero para N; (1,N): lê-se um para N.

Note que no exemplo da figura anterior os atributos das entidades não possuem tipos de dados; não sabemos, por exemplo, se o código da disciplina é um número ou um texto, e essas informações são importantes para a conversão dessas entidades em tabelas do SGBD.

Para isso, seguindo o processo de projeto de banco de dados, precisamos adentrar a fase de projeto lógico, para o refinamento do modelo conceitual de tal forma que geremos um modelo lógico mais próximo do que será implementado no SGBD.

No nosso exemplo de disciplina/aluno, suponhamos que esse modelo gere tabelas que, quando preenchidas, produzam algo como:

MATRÍCULA	NOME	TURMA
12345	Maria de Maria	ADS3
54321	João da Silva	ADS3
667788	Pedro de Pietro	ADS4

CÓDIGO	NOME
BD001	Banco de Dados
POO1	Programação Orientada a Objetos 1
WEB01	Desenvolvimento Web

Figura 21 – Exemplo de tabelas E-R

Nesse caso, a partir da figura anterior, podemos pensar, por exemplo, que o código da disciplina é um texto e que a matrícula do aluno é um número. Mas isso ainda não é suficiente, isso são apenas suposições. Além do mais, a partir desse modelo de tabela simulado, não conseguimos saber quais alunos estão matriculados em uma determinada disciplina, nem em quantas disciplinas um determinado aluno está matriculado, como indicado pelo modelo conceitual.

Faz-se necessário o desenvolvimento de um modelo lógico, como mostram o exemplo da figura seguinte e o quadro explicativo na sequência.

Figura 22 – Exemplo de modelo lógico (Diagrama E-R)

Neste caso, podemos observar a criação de uma terceira entidade, que implementa o relacionamento proposto pelo modelo conceitual. Ainda é possível notarmos, nessa entidade, o uso do conceito de chave estrangeira (ou *foreign key/FK*).

Chave estrangeira pode ser considerada como uma referência à chave primária de outra tabela em uma relação entre duas entidades. No caso do nosso exemplo, as chaves primárias "codigo" e "matricula" tornam-se referências na entidade "aluno_disciplina", uma vez que esses atributos são identificadores de suas respectivas entidades.

Podemos notar também a simbologia da representação da cardinalidade. Enquanto no modelo conceitual tínhamos a notação da cardinalidade explícita, no modelo lógico temos a representação em simbologia.

A próxima figura mostra a simbologia para representação de cardinalidade e seus respectivos significados.

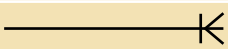
Cardinalidade	Representação
(0,1)	
(1,1)	
(0,N)	
(1,N)	

Figura 23 – Representação de cardinalidade ER

Assim, esse modelo lógico, quando simulado em uma estrutura de tabelas e com informações fictícias, produz o resultado exibido pela figura a seguir:

MATRÍCULA	NOME	TURMA
12345	Maria de Maria	ADS3
54321	João da Silva	ADS3
667788	Pedro de Pietro	ADS4

CÓDIGO	NOME
BD001	Banco de Dados
POO1	Programação Orientada a Objetos 1
WEB01	Desenvolvimento Web

MATRÍCULA	CÓDIGO
12345	BD001
54321	BD001
667788	BD001
12345	POO1
54321	POO1
667788	WEB01

Figura 24 – Exemplos de tabelas E-R

A partir do exemplo desta última figura, podemos visualizar quais alunos cursam uma determinada disciplina e quais disciplinas um aluno cursa. Todavia, ainda não conseguimos saber os tipos de dados dos atributos.

Precisamos de um modelo mais próximo do SGBD, que é o que o modelo físico se presta a fazer. Um exemplo de como seria o modelo físico, implementado a partir da linguagem SQL, utilizando o SGBD MySQL, segue na próxima figura:

```
CREATE TABLE ALUNO
(
    MATRICULA INT NOT NULL,
    NOME VARCHAR(20) NOT NULL,
    TURMA VARCHAR(10) NOT NULL,
    PRIMARY KEY(MATRICULA)
)

CREATE TABLE DISCIPLINA
(
    CODIGO VARCHAR(20) NOT NULL,
    NOME VARCHAR(100) NOT NULL,
    PRIMARY KEY(CODIGO)
)

CREATE TABLE ALUNO_DISCIPLINA
(
    MATRICULA_ALUNO,
    CODIGO_DISCIPLINA,
    FOREIGN KEY (MATRICULA_ALUNO)
      REFERENCES ALUNO(MATRICULA),
    FOREIGN KEY (CODIGO_DISCIPLINA)
      REFERENCES DISCIPLINA(CODIGO)
)
```

Figura 25 – Exemplo de modelo físico E-R

Se executarmos o *script* descrito pela figura anterior em um SGBD MySQL, serão criadas três tabelas, com seus respectivos atributos, tipos de dados, chaves primárias e chaves estrangeiras.

Retomando a orientação a objetos, os dados de uma classe têm de ser, muitas vezes, persistidos. Persistir um dado é colocá-lo em um meio de guarda permanente, em geral um banco de dados.

5.1.3 Bancos de dados relacionais *versus* orientação a objetos

Pode-se notar que o modelo relacional e o modelo orientado a objetos possuem certas diferenças, por exemplo, no modelo relacional, não temos o armazenamento dos métodos que poderiam agir sobre uma determinada classe.

Em contrapartida, esses modelos possuem semelhanças:

- Assim como na orientação a objetos, uma entidade possui atributos.
- Assim como na orientação a objetos, uma entidade possui uma identidade, que pode ser obtida a partir de um conjunto de valores de seus atributos. Podemos dizer então que cada linha de uma tabela é uma instância de uma entidade, ou um objeto.
- Assim como na orientação a objetos, as entidades se relacionam.

É exatamente aqui que está um dos principais problemas na relação entre paradigma orientado a objetos e modelo relacional: como representar algumas características da orientação a objetos, herança, por exemplo, já que não existe herança em modelos relacionais?

Uma possível solução para o problema está no banco de dados orientado a objetos. Todavia, como vimos, o modelo orientado a objetos ainda é muito incipiente.

A saída viável é fazer um "mapeamento" entre as classes do modelo orientado a objetos e as entidades do modelo relacional. Para fazer a persistência dos dados é necessário mapear as classes cujos objetos se deseja persistir em tabelas.

Quando definimos o modelo de classes de projetos e atribuímos as responsabilidades para cada classe dentro do sistema de *software*, modelamos classes transientes ou que não tenham necessidade de persistência, como aquelas que só contêm métodos, classes do tipo controle.

Lembrete

Temos três tipos de classes, quando analisamos pela questão responsabilidade, no paradigma orientado a objetos: entidade (*entity*), controle (*control*) e fronteira (*boundary*).

Olhando por esse aspecto, classes de fronteira (*boundary*) geralmente são transientes, bem como as classes de controle (*control*), logo não precisam ser persistidas, pois são compostas, em sua maioria, de métodos, e os poucos atributos que possuem não necessitam ser persistidos.

Outro ponto importante: neste mapeamento, é necessário mapear se todos os atributos serão persistidos ou apenas uma parte, pois podem existir atributos em uma classe que não necessariamente precisem ser persistidos, que são usados somente no contexto da aplicação.

Existem algumas formas de se persistir uma classe em uma ou mais tabelas.

A situação mais simples é quando uma classe é persistida diretamente em uma tabela; neste caso, o único trabalho é verificar quais atributos serão persistidos e eventualmente fazer uma adaptação de chave primária, criando ou alterando uma existente.

A segunda situação é quando temos uma composição de classes que, no caso do modelo E-R, denote uma associação do tipo "um para muitos". Nesse caso, é necessário adicionar uma chave estrangeira na extremidade "muitos" para referenciar a chave primária da tabela da outra extremidade.

A terceira situação é quando temos uma composição de classes que, no caso do modelo E-R, denote uma associação do tipo "muitos para muitos". Nesse caso, é necessário criar uma tabela intermediária contendo duas chaves estrangeiras, cada uma relacionada à chave primária de cada classe do relacionamento.

A quarta e última situação é o grande problema do mapeamento E-R/orientado a objetos: como mapear herança? Como mapear uma relação?

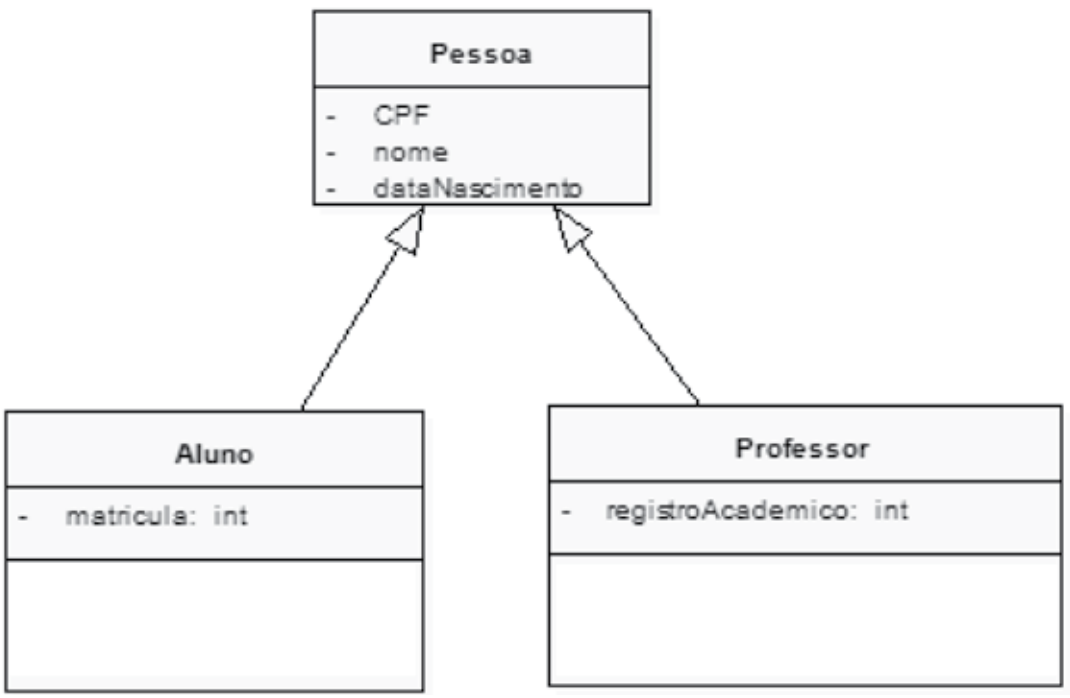


Figura 26 – Exemplo de relação de herança a ser mapeada no modelo E-R

Temos três possíveis soluções.

Solução A: criar uma entidade única contendo todas as informações, conforme mostra a figura a seguir. Neste caso, teríamos uma desvantagem, pois alguns campos ficariam vazios quando fôssemos persistir um professor ou um aluno, e isso é péssimo para o desempenho do banco de dados e para o

algoritmo tratamento no sistema de *software*. Como resultado dessa solução, com os dados preenchidos na tabela, teríamos o que segue na figura:

NOME	CPF	Data Nascimento	Matrícula	Registro Acadêmico
Maria de Maria	123.456.789-10	01/01/1980	123456	
João da Silva	111.222.333-44	01/02/1983	456789	
Pedro de Pietro	555.666.777-55	01/03/1984	987654	
Paulo de Paolo	000.111.222-34	10/02/1964		RA58972
Andrea di Andria	999.888.777-55	05/08/1979		RA987654-0

Figura 27 – Resultado da Solução A

Solução B: criar duas entidades, uma para professor e outra para aluno, como mostra a figura a seguir. Eliminaríamos o problema dos campos vazios, todavia teríamos os mesmos atributos mapeados em entidades diferentes, o que também não é produtivo. Como resultado dessa solução, com os dados preenchidos nas tabelas, teríamos o que segue nas figuras:

NOME	CPF	Data Nascimento	Matrícula
Maria de Maria	123.456.789-10	01/01/1980	123456
João da Silva	111.222.333-44	01/02/1983	456789
Pedro de Pietro	555.666.777-55	01/03/1984	987654

NOME	CPF	Data Nascimento	Registro Acadêmico
Paulo de Paolo	000.111.222-34	10/02/1964	RA58972
Andrea di Andria	999.888.777-55	05/08/1979	RA987654-0

Figura 28 – Resultado da Solução B

Solução C: criar três entidades, "aluno", "professor" e uma terceira entidade conceitual "pessoa". Eliminaríamos o problema dos campos vazios, não teríamos os mesmos atributos mapeados em entidades diferentes, todavia teríamos a necessidade da criação de um uma terceira entidade, apenas conceitual, e teríamos um custo para manter esse modelo. Como resultado dessa solução, com os dados preenchidos nas tabelas, teríamos o que segue nas figuras:

NOME	CPF	Data Nascimento
Maria de Maria	123.456.789-10	01/01/1980
João da Silva	111.222.333-44	01/02/1983
Pedro de Pietro	555.666.777-55	01/03/1984
Paulo de Paolo	000.111.222-34	10/02/1964
Andrea di Andria	999.888.777-55	05/08/1979

CPF	Matrícula
123.456.789-10	123456
111.222.333-44	456789
555.666.777-55	987654

CPF	Registro Acadêmico
000.111.222-34	RA58972
999.888.777-55	RA987654-0

Figura 29 – Resultado da Solução C

Dentre as três situações, a que fere menos os princípios da normalização de banco de dados é a situação C; de qualquer forma, temos de ter em mente que não existe uma adaptação natural do modelo E-R para o modelo OO.

Observação

Normalização é o nome que se dá ao processo de organizar e dividir as tabelas da forma mais eficiente possível, diminuindo a redundância e permitindo a evolução do BD com o mínimo de efeitos colaterais.

Atualmente existem alguns *frameworks*, ou bibliotecas prontas, chamados *frameworks*, de persistência, por exemplo, o NHibernate, para aplicações baseadas no Microsoft .Net, e o Hibernate, similar, utilizado para soluções desenvolvidas utilizando a plataforma Java.

O objetivo desses componentes é justamente fazer o mapeamento, de forma automática, do modelo orientado a objetos para o E-R e vice-versa, além de abstrair para o desenvolvedor todo o

gerenciamento dessas informações no SGDB, ou seja, o próprio *framework* disponibiliza funções prontas para criação, atualização e recuperação das informações do BD para o modelo orientado a objetos, sem que o desenvolvedor necessite desenvolver essas funcionalidades.

Observação

Atualmente, a interface entre uma aplicação e as funções para criação, atualização e recuperação das informações do BD se dá a partir do desenvolvimento de consultas baseadas na linguagem SQL, e essas consultas são desenvolvidas pela equipe de construção.

A única e principal ressalva a ser feita na adoção desses mecanismos é que, muito embora sejam facilitadores na produtividade do desenvolvimento, eles passam a não ter um bom desempenho, uma vez que uma das partes do modelo não está benfeita.

Normalmente, quando estamos tratando de sistemas legados, ou que já existam há algum tempo, é comum que nos deparemos com modelos E-R, ou até mesmo modelos de classes, desenvolvidos sem utilizar as melhores práticas.

Nesses casos, não é recomendável a utilização destes *frameworks*. Em muitos casos, na fase de projeto, tecnologia alguma substitui a capacidade analítica da equipe de projeto.

5.2 Projeto arquitetural

Neste subtópico avançaremos para a fase de projeto arquitetural, que é a fase seguinte à de projeto de dados e classes que debatemos no subtópico anterior.

5.2.1 Qual o papel da arquitetura?

Na engenharia civil, arquitetura é a arte de projetar e construir prédios, edifícios ou outras estruturas, a constituição de um edifício ou a contextura de um todo (MICHAELIS, 1998).

Analogamente à arquitetura na engenharia civil, a arquitetura desempenha papel importante também na engenharia de *software*.

Na engenharia de *software*, o papel da arquitetura é bem semelhante ao da arquitetura na engenharia civil, ou seja, projetar estruturas, a constituição de um *software* ou a contextura de um todo, assim como na arquitetura da engenharia civil.

A figura a seguir mostra exatamente onde a arquitetura se encontra dentro do contexto do ciclo de vida de um projeto de *software*.

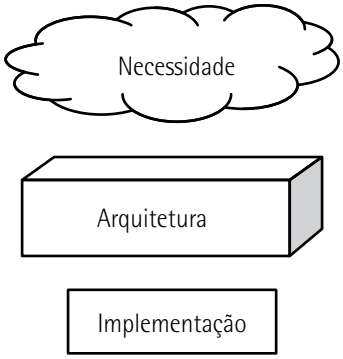


Figura 30 – Disposição da arquitetura dentro de um projeto de *software*

Arquitetura de *software* é o nome dado à disciplina fundamental da fase de projeto arquitetural.

5.2.2 O que é arquitetura de *software*?

Algumas das definições clássicas de arquitetura de *software* são:

- Arquitetura de *software* é uma representação do sistema que auxilia na compreensão de como ele irá se comportar (SEI, 2015a).
- Arquitetura de *software* é uma descrição de como um sistema de *software* é organizado (SOMMERVILLE, 2010).
- Arquitetura é a organização fundamental de um sistema incorporada em seus componentes, seus relacionamentos com o ambiente, e nos princípios que conduzem seu projeto, construção e evolução (ISO, 2011b).
- Arquitetura de *software* pode ser definida como uma representação, em alto nível de abstração, dos componentes de um sistema de *software*, suas propriedades externamente visíveis, e como estes interagem com o objetivo de resolver as necessidades, ou problema de um cenário de negócio (BASS; CLEMENTS; KAZMAN, 2010).

Observação

A norma ISO/IEC/IEEE 42010 (2011b) tem como objetivo padronizar a prática da arquitetura definindo termo, apresentando uma base conceitual para expressar, comunicar e rever arquiteturas e especificações de requisitos que se aplicam às descrições de arquitetura, *frameworks* arquiteturais e linguagens de descrição de arquitetura. Esta norma substitui a norma ISO 1471 de 2000.

Todas as definições de arquitetura de *software* corroboram um conceito único: o de projeto de *software*.

Está claro que a arquitetura de um *software* não é o *software* em si, mas sim a representação, o modelo, a organização do *software* que será efetivamente construído.

Mais importante do que pensar em componentes de um sistema de *software* e como estes se comunicam é pensar na natureza desses componentes, quais são suas responsabilidades, qual é o significado de suas conexões e qual o significado da forma como estes componentes estão distribuídos no modelo (BASS; CLEMENTS; KAZMAN, 2010).

O tema Arquitetura de *Software* é bem amplo. O livro de Bass, Clements e Kazman (2010), que é uma das principais referências no tema, entende que pensar na natureza dos componentes é uma forma diferente de se pensar em arquitetura de *software*.

Os autores ainda defendem que essa abordagem deve ter ênfase nos requisitos não funcionais, ou atributos de qualidade, do problema a ser resolvido.

Esta ênfase nos atributos de qualidade dá-se pela maior dependência destes em relação à estrutura do *software*, uma vez que as funcionalidades, ou requisitos funcionais, podem ser implementadas em um número maior de alternativas, sendo assim menos dependentes das decisões arquiteturais (BASS; CLEMENTS; KAZMAN, 2010).

5.2.3 A importância da arquitetura de *software*

A importância fundamental da arquitetura de um *software* é a mesma importância que a arquitetura tem na construção de uma casa: a arquitetura, ou o projeto arquitetural, representado por uma planta, por exemplo, é o principal guia para a construção.

Assim como a arquitetura na engenharia civil, a arquitetura de *software* produz modelos e organiza a estrutura de um *software* de tal forma que sirva como o principal guia para a construção.

Segundo o SEI (2015a), a arquitetura de um sistema de *software* serve como modelo tanto para o projeto quanto para a construção, facilitando inclusive a divisão das tarefas a serem realizadas pela equipe de projeto e de construção.

Além do mais, a arquitetura é o principal meio para o atingimento de qualidade de um sistema como: desempenho, manutenibilidade e segurança. Nenhum índice de qualidade pode ser alcançado sem uma visão arquitetural unificada (SEI, 2015a).

Observação

Note que, assim como Bass, Clements e Kazman (2010), o SEI (2015a) bate na tecla de que a arquitetura de *software* está intimamente ligada aos atributos de qualidade, ou requisitos não funcionais, que podem ser vistos, por exemplo, na norma ISO 25010 (ISO, 2011a).

Bass, Clements e Kazman (2010) pontuam a importância da arquitetura de *software:*

- Facilita a comunicação entre os envolvidos no projeto.
- Representa, em escala menor e compreensível, a forma pela qual os componentes de um sistema são estruturados e interagem.
- Modela e coloca em evidência as decisões de projeto como fator fundamental para o sucesso do sistema de *software*.

Alguns outros benefícios de uma arquitetura bem-definida para um sistema de *software*:

- Facilita a evolução do *software*.
- Facilita o reúso de componentes, logo promove maior produtividade na construção e na manutenção, pois também facilita a manutenção.
- Promove a diminuição do espaço entre as fases de análise e construção e, por consequência, a diminuição do retrabalho na construção.
- Auxilia a gestão do projeto quanto à estimativa de tempo, ao custo e à complexidade do projeto.
- Promove mitigação e diminuição de riscos.

5.2.4 Processo de arquitetura de *software*

Segundo Albin (2003), a participação de arquitetura de *software* pode ser dividida em cinco fases, e deve ser iniciada antes mesmo da fase de projetos do ciclo de vida da engenharia de *software*:

Fases do ciclo de vida da engenharia de *software*		Fases da arquiteura
Análise de requisitos	→	Pré-projeto
Projeto	→	Análise de domínio
		Projeto esquemático
Implementação e testes	→	Desenvolvimento de projeto
		Construção
Implantação	→	(sem correspondência)

Figura 31 – Participação da arquitetura *versus* fases da engenharia de *software*

Participação da arquitetura dentro do ciclo de vida do projeto:

- Pré-projeto: é comum observarmos que a arquitetura de *software* participa do ciclo de vida de desenvolvimento apenas a partir da fase de projetos. No entanto, é aconselhável

que o arquiteto, ou a equipe de arquitetura, participe como observador na fase de análise de requisitos. Essa participação é importante, pois possibilita a extração de informações técnicas importantes por exemplo, padrões de arquitetura corporativa para integração de sistemas, que podem ser importantes para mitigação de custo, tempo de desenvolvimento e perfil da equipe.

- Análise de domínio: o entendimento dos requisitos funcionais fundamentais é ponto crucial para definição do modelo de domínio que será utilizado como entrada para a fase de projeto, como vimos nas unidades anteriores.

- Projeto esquemático: criação ou seleção de um modelo de arquitetura. Nessa fase o arquiteto deve criar ou escolher um modelo arquitetural que irá direcionar o processo de modelagem arquitetural e o desenvolvimento.

- Desenvolvimento do projeto: nessa fase ocorre a escolha de elementos tecnológicos, como *frameworks* ou componentes de desenvolvimento, que deem suporte à implementação.

- Construção: a implementação ou construção é orientada pela arquitetura. Ela irá orientar tecnologias e todo o processo de desenvolvimento no qual os desenvolvedores estarão envolvidos.

O processo de definição da arquitetura também pode ser subdividido em algumas atividades, que são extensões das fases que citamos anteriormente. A figura a seguir mostra a sequência de atividades do processo de definição de arquitetura sugerido por Albin (2003), e o quadro na sequência contextualiza o objetivo de cada fase.

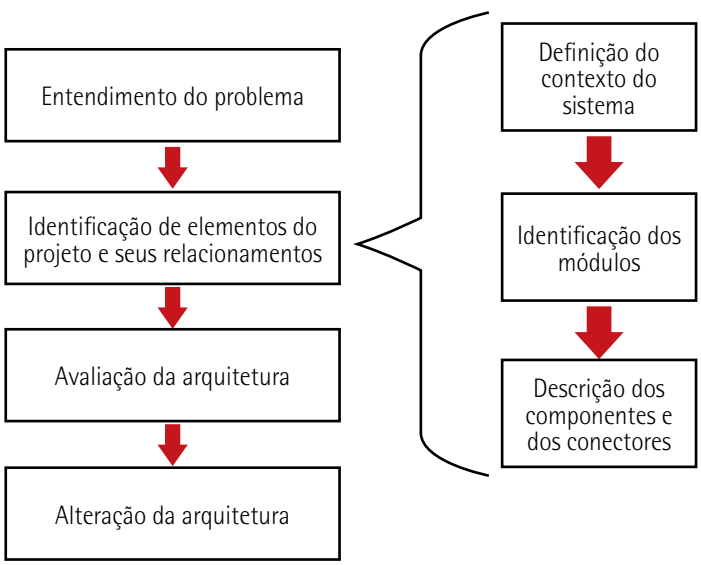


Figura 32 – Fases do processo de definição de arquitetura

Quadro 8 – Fases do processo de arquitetura

Fase	Descrição
Entendimento do problema	A fase de entendimento do problema deve ser iniciada, idealmente, na fase de análise de requisitos, como dissemos anteriormente.
Identificação de elementos do projeto e seus relacionamentos	Nesta etapa, estabelecemos uma decomposição do sistema em uma linha de base que, inicialmente, divide o sistema baseado nos requisitos funcionais. A arquitetura de um aplicativo de *software* é, muitas vezes, representada como um conjunto de módulos ou subsistemas interligados, muitas vezes chamados de componentes. Estes módulos são construções organizacionais que o código-fonte estrutura, mas muitas vezes não são diretamente visíveis no código-fonte. O código-fonte é uma abstração de instruções de máquina e padrões de dados estruturados em termos de gramática de uma linguagem de programação. Esta fase também pode ser subdividida em três etapas: • Definição do contexto do sistema: o contexto do sistema ajuda a descrever a aplicação de uma perspectiva externa ao sistema, ou seja, aos usuários do sistema. O contexto do sistema é útil para descrever a finalidade do sistema, bem como para identificar as interfaces externas deste sistema; o insumo para definir o contexto do sistema é a lista de requisitos inicial. • Identificação dos módulos: os módulos são unidades de *software* (binário e fonte). Módulos binários são instanciados em tempo de execução, e essas instâncias são chamadas de componentes. Um determinado módulo interage com os demais a partir de conectores e ele pode atender a uma determinada quantidade de situações e possuir mais de um conector. • Descrição dos componentes e dos conectores: é importante a descrição dos componentes, seu comportamento nas mais diversas situações, bem como as condições de seu acionamento, que vêm a ser a descrição de seus conectores. A partir dessa descrição podemos avaliar se um componente possui alto ou baixo acoplamento e alta ou baixa coesão.
Avaliação da arquitetura	A atividade de avaliação da arquitetura tem como objetivo verificar possíveis pontos frágeis da arquitetura, para melhoria.
Alteração da arquitetura	A alteração da arquitetura vem em decorrência dos possíveis pontos frágeis encontrados na fase de avaliação.

Saiba mais

A avaliação de arquitetura de um sistema de *software* deve, por boa prática, seguir um método ou modelo de referência. Um exemplo de modelo de referência para avaliação de arquitetura é o ATAM; mais informações podem ser encontradas em:

SOFTWARE ENGINEERING INSTITUTE (SEI). *Architecture tradeoff analysis method*, 2015b. Disponível em: <www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm>. Acesso em: 4 maio 2015.

Nos próximos capítulos, daremos ênfase ao detalhamento da fase de identificação de elementos do projeto e aos seus relacionamentos.

5.2.5 Processo de arquitetura de *software* em aspectos humanos

O processo de arquitetura de *software*, bem como as demais atividades desenvolvidas ao longo de todo o ciclo de desenvolvimento do projeto, possuem alta dependência do fator humano.

Segundo o OpenUP ([s.d.]a), existem alguns interessados no processo de definição de arquitetura de um sistema de *software*: analistas de requisitos, desenvolvedores, gerentes de projetos e arquitetos de *software*.

Existem algumas divergências quanto à participação do gerente de projetos nesse processo, uma vez que se trata de uma atividade de cunho estritamente técnico.

Na verdade, se levarmos em consideração o papel da arquitetura de *software* dentro do projeto de *software*, veremos que o gerente de projeto é um dos principais envolvidos no processo de definição de arquitetura.

Lembrete

O fato de um papel envolver interesse em um processo não quer dizer que ele seja responsável direto por desempenhar uma determinada função. No caso da arquitetura, lembramos que um de seus principais benefícios é auxiliar a gestão do projeto quanto à estimativa de tempo, ao custo e à complexidade do projeto; logo, o gerente de projeto é um interessado no processo de arquitetura.

Obviamente, o papel principal durante o projeto arquitetural é do arquiteto de *software*. Mas, afinal de contas, quem é o arquiteto? Quais são suas responsabilidades? Quais são as habilidades necessárias para ser um arquiteto de *software*?

Para definirmos quem é o arquiteto de *software*, vamos dividir a visão da seguinte forma: primeiramente sob a ótica do projeto em que atua, e depois sob a ótica de sua área de atuação.

Sob o ponto de vista do projeto em que atua, o arquiteto tem a obrigação de:

- Conhecer os aspectos culturais do seu cliente.
- Conhecer as estratégias de mercado do cliente e o que ele pretende com *software*.
- Conhecer as limitações de tecnologia e infraestrutura do cliente.
- Conhecer o domínio do negócio e do problema sobre o qual pretende promover uma solução.

Já sob o ponto de vista da área de atuação, o arquiteto tem a obrigação de:

- Manter-se constantemente atualizado a respeito das metodologias, tecnologias, soluções e plataformas de desenvolvimento.
- Conhecimento nível sênior em engenharia de *software*, que compreende conhecimento em pessoas, ferramentas e processos inerentes à engenharia de *software*.

As responsabilidades do arquiteto no projeto arquitetural, além de converter os requisitos do cliente em projeto de *software*, são:

- Promover um estudo sobre a viabilidade das soluções técnicas adotadas, a chamada prova de conceitos.
- Avaliar as tendências tecnológicas, em vista das soluções adotadas no projeto, sob o ponto de vista de competitividade do produto a ser desenvolvido.
- Buscar o aumento do tempo de vida útil do *software*.

6 VISÕES DA ARQUITETURA DE *SOFTWARE*

Como vimos, a arquitetura de um sistema de *software* define os componentes que formarão este sistema de *software*. Estes componentes podem ter o menor nível de composição, dependendo do nível de abstração utilizado nessa composição.

Esses componentes podem ser divididos em dois grupos: estáticos e dinâmicos.

- A visão estática da arquitetura promove a visão da organização dos componentes do *software* e de suas relações com elementos de dados (banco de dados, arquivos-texto etc.) com sistemas de *hardware* e com outros sistemas de *software*. Além, claro, de promover a visão de como estes componentes se relacionam entre si.
- A visão dinâmica da arquitetura promove a visão comportamental do sistema e de seus componentes durante a execução de uma determinada ação. Nessa visão, definimos como os componentes do sistema reagem a eventos internos e externos e ainda a forma como eles se comunicam entre si, ou seja, o protocolo de comunicação interno, e até mesmo como eles se comunicam com componentes externos ao domínio do *software*, chamado de protocolo de comunicação externa.

6.1 Visão estática

A definição da estrutura arquitetural estática de uma solução não deve ser uma atividade executada de forma aleatória, sem um estudo prévio.

Projetar uma estrutura de *software* significa escolher as alternativas de soluções adequadas e inerentes ao domínio do problema, e esse domínio possui inúmeras variantes, por exemplo, os aspectos

culturais, a limitação tecnológica e de infraestrutura do cliente e a expectativa de competitividade do cliente em relação ao *software* a ser desenvolvido.

Essas alternativas de soluções devem validadas antes que sejam utilizadas através de provas de conceitos (poc) ou mesmo a partir de soluções que tenham sido previamente validadas.

Soluções previamente validadas podem ser: *frameworks*, linguagens, estilos e principalmente padrões. Definir a arquitetura de *software* passa principalmente por saber utilizar padrões.

6.1.1 Utilização de padrões na arquitetura de *software*

Utilização de padrões e padronização é um conceito amplamente utilizado por outras engenharias. A engenharia civil, por exemplo, possui uma gama de padrões para resolver o problema de distribuição elétrica de uma residência, assim como a engenharia elétrica possui um conjunto de soluções-padrão para comunicação entre componentes eletrônicos em uma placa de circuito.

Padrões são determinadas soluções que comprovadamente funcionam para resolver um determinado problema. Ressaltamos, aqui, que se trata de determinadas soluções para determinados problemas.

No entanto, pode surgir uma dúvida: uma determinada solução que comprovadamente resolve um determinado problema é garantia de solução para outros problemas?

Esse é o principal dilema para o uso de padrões na arquitetura de *software*, pois existe a tendência em se usar soluções que garantem a resolução de um determinado problema para resolver outros problemas.

Utilizar dessa maneira um padrão na arquitetura de *software* é como utilizar um medicamento errado ou utilizar o medicamento certo em dose errada para o tratamento de uma determinada enfermidade.

Gamma *et al.* (1995), uma das principais referências em padrões na arquitetura de *software*, definem padrão como uma solução reutilizável descrita com base em três partes: um contexto, um problema e uma solução.

- Contexto: estende o problema a ser solucionado, apresentando situações de ocorrência desses problemas.

- Problema: determinado por um sistema de forças, em que estas forças estabelecem os aspectos do problema que devem ser considerados.

- Solução: mostra como resolver o problema recorrente e como balancear as forças associadas a ele.

Observação

Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, autores do livro *Design Patterns: Elements of Reusable Object-Oriented Software*, são considerados as maiores autoridades e precursores na utilização de projetos em arquitetura de *software*. Este grupo foi apelidado de *gang of four* (GoF) ou grupo dos quatro e é tratado assim na literatura específica da área.

A utilização de padrões na arquitetura de *software* tem como objetivos principais resolver problemas macros comuns a todos os sistemas de *software*, como: produtividade, reúso, redução de complexidade e geração de um protocolo de comunicação entre todos os envolvidos em um projeto de *software*. A opção pela utilização de um determinado padrão ou de um conjunto de padrões é chamada de decisão arquitetural.

O uso de um conjunto de padrões que resolve um determinado problema é chamado de estilo arquitetural ou modelo arquitetural.

6.1.2 Estilo arquitetural

Estilo arquitetural, modelo arquitetural ou ainda padrão arquitetural é a organização, em um alto nível de abstração, de um sistema de *software* em um conjunto finito de subsistemas. Essa organização especifica as responsabilidades, regras de organização e o relacionamento entre estes subsistemas (BUSCHMANN *et al.*, 1996).

Buschmann *et al.* (1996) aponta que um padrão arquitetural, além de auxiliar no desenvolvimento da estrutura fundamental de um sistema de *software*, auxilia no atendimento de um atributo de qualidade deste sistema, por exemplo, manutenibilidade.

O benefício da aplicação de um estilo arquitetural dá-se pela capacidade de reúso; logo, pela produtividade que ele proporciona.

Um estilo arquitetural visa à solução de um problema em um determinado contexto, todavia tem sua ênfase no nível estrutural da solução, não adentrando o nível de implementação, razão que lhe confere maior possibilidade de reúso no caso de este problema também existir em um contexto de negócio diferente do original (BUSCHMANN *et al.*, 1996).

Os principais estilos arquiteturais, extraídos de Buschmann *et al.* (1996), são classificados em quatro categorias. Essas categorias representam contextos de problemas com características similares, por exemplo, a necessidade de separação do sistema em componentes ou a necessidade de apoiar melhor a interação humano-computador, e um estilo pode se encaixar em mais de uma classificação. O seguinte mostra o agrupamento desses estilos e uma breve descrição de cada estilo arquitetural.

Quadro 9 – Estilos arquiteturais

Categoria	Estilo arquitetural
From mud to structure (Da "lama" à estrutura)	Camadas ou *Layers*
	Pipes and Filters
	Blackboard
Sistemas distribuídos	*Broker*
	Microkernel
	Pipes and Filters
Sistemas interativos	*Model-View-Controller (MVC)*
	Presentation-Abstraction-Control (PAC)
Sistemas adaptáveis	*Reflection*
	Microkernel

- Camadas ou *Layers*: padrão que promove a divisão em níveis de abstração, em que o critério para essa divisão é a responsabilidade de cada nível, ou camada (BUSCHMANN *et al.*, 2007).

- *Pipes and Filters*: este padrão promove a divisão de cada atividade da aplicação em um passo a passo de processamento de informações. Cada passo é uma unidade de processamento que recebe, processa e devolve uma informação que serve de entrada para o próximo. Esse processo gera uma cadeia denominada *pipeline* (BUSCHMANN *et al.*, 2007).

- *Blackboard*: neste padrão são montadas soluções para uma tarefa através do uso da heurística computacional. Essas soluções são chamadas hipóteses e são alimentadas gradualmente através de algoritmos de pequenos componentes do sistema e armazenadas em uma área de memória chamada *blackboard*. A solução da tarefa é colaborativa, dada pelo conjunto dessas hipóteses, originadas de um conjunto de componentes da aplicação (BUSCHMANN *et al.*, 2007).

- *Broker*: promove o encapsulamento de detalhes da infraestrutura de comunicação e a separação desta das funcionalidades da aplicação (BUSCHMANN *et al.*, 2007).

- *Microkernel*: promove a criação de um componente fechado com todos os serviços fundamentais para a aplicação, esse componente é chamado *microkernel*. Outras funcionalidades específicas são distribuídas em outros componentes da aplicação (BUSCHMANN *et al.*, 2007).

- *Model-View-Controller* (MVC): propõe a divisão lógica da aplicação em três camadas: *model, view e controller*. A camada *Model* ou modelo é responsável por objetos que representem o domínio da aplicação, por exemplo, as regras de negócio; a camada *View* ou visão representa a interface com o usuário; e a camada *Controller* tem como objetivo o gerenciamento do fluxo da aplicação (BUSCHMANN *et al.*, 2007).

- *Presentation-Abstraction-Control* (PAC)*:* propõe a divisão da aplicação em componentes, em que cada qual tem três divisões lógicas: *presentation, abstraction e control*. A camada *Presentation* ou

apresentação é responsável pela interface com o usuário, a camada *Abstraction* ou abstração é responsável por objetos que representam o domínio da aplicação, por exemplo, as regras de negócio, e a camada *Control* ou controle é responsável pelo fluxo da solução e comunicação entre os componentes da solução. O padrão PAC difere do MVC em um ponto: permite que cada componente seja uma célula de processamento, ou *thread* independente; pode-se dizer que cada componente é uma implementação do padrão MVC (COUTAZ, 1987).

- *Reflection*: propõe a separação da aplicação em duas camadas, denominadas metanível e nível-base. A camada metanível tem os chamados meta-objetos, que são objetos cujas responsabilidades estão ligadas a características estruturais, comportamentais e de estados da aplicação. A camada-base contém as funcionalidades da aplicação (BUSCHMANN *et al.*, 2007).

Observação

Esta disciplina não tem como objetivo principal a formação de arquiteto de *software*; para tal, é preciso maior especialização em algumas áreas e disciplinas específicas. Nosso objetivo principal é fazer você ter o primeiro contato com o tema e na sua vida profissional saiba entender e seguir um estilo adotado em um sistema de *software*.

De todos esses estilos, nos aprofundaremos em um dos estilos arquiteturais mais utilizados atualmente: arquitetura em camadas, que tem sua importância por ser uma arquitetura-base para outros estilos, como MVC e PAC.

6.1.3 Estruturação de sistemas em subsistemas e camadas

Antes de explorarmos a solução, vamos debater um pouco do problema que teremos de resolver.

Imagine a seguinte situação: você é contratado para desenvolver um simples sistema de cadastro de funcionários. Nesse sistema, terá apenas de inserir informações básicas de um funcionário, como nome, CPF, RG, número do PIS e salário; no requisito salário, o usuário do sistema entra com o salário bruto e o sistema automaticamente calcula o salário líquido a partir de algumas regras como desconto de imposto de renda e INSS. Suponhamos ainda que esse sistema seja um sistema *web*.

Você então desenvolve uma página *web* e dentro dessa página você desenvolve as regras de negócio para cálculo, validações de campos obrigatórios, de consistência de CPF, efetua conexão com o banco de dados e insere um registro no repositório de informações.

Até o momento, aparentemente, tudo está normal, afinal de contas, todos os requisitos do usuário foram atendidos e o sistema se dispõe a resolver o problema proposto.

Imaginemos agora que nesse sistema seja acrescido o cadastro de plano de previdência privada, que está associado ao salário líquido do funcionário. Aparentemente, não há problema algum, exceto

pelo fato de que você não terá como reutilizar o código para implementar o cálculo do salário líquido do funcionário, o que não aparenta ser um grande problema, pois você pode copiar e colar o código de uma página para outra.

Agora o sistema está funcionando normalmente, a operação dia a dia ocorre sem mais problemas, até que um determinado dia ocorre uma mudança.

É comum a ocorrência de mudanças na legislação, que acabam por acarretar modificações nos sistemas de informação que porventura utilizem ou trabalhem com informações governamentais.

Pois bem, eis que ocorre uma mudança hipotética no cálculo do imposto de renda e do INSS, o que interfere diretamente no cálculo do salário líquido dos funcionários.

Aproveitando esse momento, seu cliente decide, já que o sistema será alterado, solicitar algumas mudanças no *layout* das páginas *web*, como mudança no padrão de cores, imagens e melhoria na usabilidade. Já que estamos em um momento de mudança, o diretor de TI do seu cliente decide resolver um antigo problema: a mudança do fornecedor do SGDB e a padronização dos nomes das tabelas do banco de dados.

As alterações não parecem complexas e na verdade elas realmente não são, todavia você esbarra, agora, em alguns problemas:

- Você não se lembra em quantos pontos do código a regra de cálculo de salário líquido é executada. O sistema cresceu demais, outras páginas foram criadas, outras regras, implementadas, enfim, você terá de procurar todos os pontos, alterá-los e testá-los.

- Você tem um alto acoplamento de fatores que não têm nenhuma relação, como o banco de dados e suas páginas *web*. Como a parte de banco de dados está na página *web*, você acabou criando uma dependência indireta de componentes que não necessitam um do outro para funcionar. Na prática, a mudança no banco de dados não deveria influenciar a sua página *web*, mas, da forma como está construído, você necessariamente terá de testar tudo novamente, uma vez que haverá alteração no código-fonte da página.

- Você monta um cronograma com as atividades sequenciais, aumentando o custo da manutenção. Por quê? Porque é impossível dividir e paralelizar as tarefas de alteração da regra de negócio do cálculo de impostos, a alteração das páginas e a alteração do banco de dados, uma vez que todas elas serão feitas nos mesmos componentes do *software*, ou seja, nas páginas *web*.

O problema por que você está passando, infelizmente, é muito comum de encontrar na nossa vida profissional. O seu sistema possui:

- Alto acoplamento e baixa coesão: quando existem dependências demais de tecnologias e componentes que, em tese, não deveriam depender, pois possuem independência funcional umas das outras.

- Baixa manutenibilidade: ocorre quando a sua produtividade na manutenção é baixa, você tem de procurar todos os pontos do código que devem ser alterados e é impossível que se altere um componente sem que o todo seja afetado.

- Baixo reúso: é quando seu sistema basicamente não possui reúso, lembrando que copiar e colar um trecho de código não pode ser considerado reúso, mas, meramente, uma cópia, uma facilidade para que você não precise digitar todo o código novamente.

Imaginemos então que pudéssemos separar em uma "caixa" todas as classes que fossem relacionadas a páginas *web*, em uma segunda "caixa" todas as classes que executassem algum tipo de regra de negócio e em uma terceira todas as classes que fizessem interface com o banco de dados, e que essas "caixas" tivessem um protocolo de comunicação entre elas, como mostra a figura a seguir:

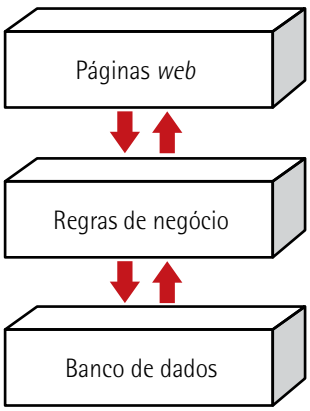


Figura 33 – Organização em "caixas"

Neste caso, teríamos resolvido alguns problemas:

- Divisão de responsabilidades: cada "caixa" teria uma responsabilidade bem-definida.

- Baixo acoplamento: as "caixas" teriam apenas as dependências necessárias, por exemplo, as páginas *web* não precisariam depender do banco de dados e vice-versa.

- Boa manutenibilidade: agora é possível saber exatamente em que "caixa" se encontram os componentes responsáveis por fazer algum tipo de cálculo ou executar alguma regra de negócio, bem como é possível paralelizar as atividades, cada equipe cuida da manutenção de uma "caixa": uma cuida do banco de dados, outra das páginas *web* e uma terceira das mudanças nas regras de negócio. O trabalho é mais produtivo.

- Reúso: os componentes podem ser reutilizáveis, ou seja, você pode simplesmente usar um componente da "caixa" regra de negócio de quantas páginas *web* você quiser, com a certeza de que o código implementado é o mesmo, sem ter de utilizar o artifício de copiar o código.

Notamos que a simples decomposição das classes e a organização delas em estruturas conceituais de acordo com as suas responsabilidades já resolve boa parte dos problemas.

É como se dividíssemos o nosso sistema em camadas, em que cada "caixa" corresponde a uma camada. Esse é o estilo arquitetural de camadas, ou arquitetura três camadas, que possui a estrutura final, representada na figura a seguir:

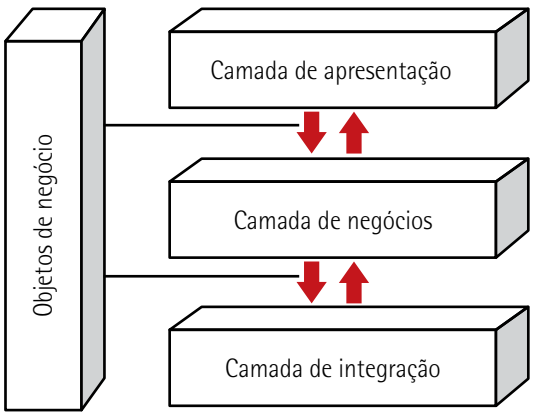


Figura 34 – Arquitetura em camadas

Nesta arquitetura temos três camadas principais:

- Camada de apresentação: que contém classes responsáveis pela interação com o usuário, como páginas *web*, formulários etc.

- Camada de negócios: que contém classes responsáveis por execução de regras de negócio, como cálculos, validações etc.

- Camada de integração: que contém classes responsáveis por fazer integração com tecnologias externas ao sistema, como banco de dados, serviços *web*, além de outros sistemas ou dispositivos de *hardware*.

Definimos que as camadas se comunicarão através de um protocolo e que esse protocolo é definido pela camada auxiliar de objetos de negócio. Essa camada é composta por objetos cuja finalidade é apenas transportar informação, sem executar nenhuma regra, sem fazer nenhuma interface externa. São as nossas classes do tipo entidade, ou *entity*.

No nosso exemplo, poderíamos ter nessa camada uma classe "Funcionário" apenas com os atributos: nome, CPF, RG, número do PIS, salário bruto e salário líquido:

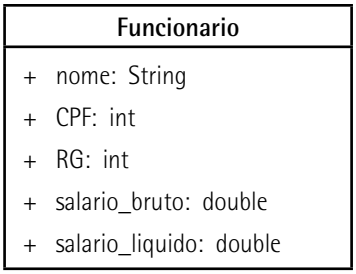


Figura 35 – Classe "Funcionario"

Lembrete

Podemos atribuir responsabilidades às classes em um modelo de projeto, que são: entidade (*entity*), controle (*control*) e fronteira (*boundary*).

Ainda na linha das responsabilidades das classes, seria correto afirmar que, nas camadas de integração e de apresentação, teremos única e exclusivamente objetos do tipo fronteira (*boundary*), e, na camada de negócios, apenas objetos do tipo controle (*control*).

A arquitetura em camadas é muito importante; por ora, direcione seus esforços a entender o conceito, que é simples.

Saiba mais

Aprofunde seus conhecimentos sobre arquitetura em camadas utilizando a plataforma Microsoft .Net em:

<http://layerguidance.codeplex.com/>.

6.1.4 Introdução a padrões de projeto (*design patterns*)

Padrão de projeto, ou *design pattern*, é a menor estrutura arquitetural proveniente dos subsistemas de um sistema de *software* e do relacionamento entre eles (BUSCHMANN *et al.*, 1996).

É semelhante aos estilos arquiteturais em todos os pontos, todavia difere pelo nível de abstração. Ao passo que um estilo arquitetural tem ênfase no nível estrutural da solução, o padrão de projeto tem ênfase na estrutura do subsistema da solução (BASS; CLEMENTS; KAZMAN, 2010; BUSCHMANN *et al.*, 1996).

A exemplo do estilo arquitetural, o padrão de projeto não entra no nível de implementação, sendo independente de qualquer tipo de tecnologia ou linguagem, razão pela qual possibilita seu reúso em

diversos contextos de problemas, além de promover a redução da complexidade da solução através da decomposição de subsistemas de maior complexidade em componentes de menor complexidade (BUSCHMANN *et al.*, 1996).

Gamma *et al.* (1995) propõem um catálogo com 23 padrões de projetos, organizados em quatro categorias, que representam contextos de uso com características similares. O quadro a seguir mostra o agrupamento desses padrões:

Quadro 10 – Padrões de projeto

Categoria	Padrão de projeto
Padrões para criação	*Abstract factory*
	Builder
	Factory method
	Prototype
	Singleton
Padrões estruturais	*Adapter*
	Bridge
	Composite
	Decorator
	Facade
	Flyweight
	Proxy
Padrões comportamentais	*Chain of responsibility*
	Command
	Interpreter
	Iterator
	Mediator
	Memento
	Observer
	State
	Strategy
	Template method
	Visitor

Segue uma breve descrição de cada padrão de projeto apresentado no quadro anterior:

- *Abstract factory*: padrão que provê uma interface para criação de famílias de objetos relacionados ou dependentes sem que haja necessidade de discriminação de suas classes concretas (GAMMA *et al.*, 1995).

- *Builder*: propõe a separação da construção de um objeto complexo de sua representação, assim, o mesmo processo de criação pode criar diferentes representações (GAMMA *et al.*, 1995).

- *Factory method*: padrão que provê uma interface para criação de um objeto, esta atividade fica a cargo de subclasses, que decidem quais classes devem ser instanciadas (GAMMA *et al.*, 1995).

- *Prototype*: propõe a criação de objetos através da instância de um protótipo. Instâncias de novos objetos são criadas pela cópia deste protótipo (GAMMA *et al.*, 1995).

- *Singleton*: padrão que promove a garantia de que uma classe terá uma única instância dentro da aplicação e propõe um ponto de acesso único de todos os pontos da aplicação à instância dessa classe (GAMMA *et al.*, 1995).

- *Adapter*: propõe a conversão da interface de uma classe para outra interface esperada pelo cliente (GAMMA *et al.*, 1995).

- *Bridge*: promove a separação da abstração de uma classe de sua implementação (GAMMA *et al.*, 1995).

- *Composite*: padrão que propõe a composição de um objeto em uma árvore hierárquica que representa o conceito todo/parte desse objeto (GAMMA *et al.*, 1995).

- *Decorator*: propõe que comportamentos sejam adicionados a um objeto de forma dinâmica (GAMMA *et al.*, 1995).

- *Facade*: padrão que provê um ponto único de interface para um conjunto de interfaces de subsistemas (GAMMA *et al.*, 1995).

- *Flyweight*: este padrão propõe a criação de um objeto compartilhado que pode ser usado em múltiplos contextos simultaneamente, com o objetivo de apoiar o uso de um grande número de objetos de menor granularidade de forma eficiente (GAMMA *et al.*, 1995).

- *Proxy*: este padrão propõe que um objeto crie um mecanismo para controlar o acesso de outro objeto a seus recursos (GAMMA *et al.*, 1995).

- *Chain of responsability*: propõe a separação de um objeto remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a capacidade de receber essa solicitação. Esses objetos são colocados em uma cadeia, e a solicitação é passada de um a outro até que o destinatário a receba (GAMMA *et al.*, 1995).

- *Command*: propõe a abstração de uma requisição como um objeto. Possibilita parametrização de clientes com diferentes requisições, filas ou armazenamento dessas informações em formato de *log* e ainda apoia operações que podem ser desfeitas (GAMMA *et al.*, 1995).

- *Interpreter*: esse padrão descreve como definir uma representação para uma determinada linguagem, representar sentenças nesta linguagem e interpretar essas sentenças (GAMMA *et al.*, 1995).

- *Iterator*: propõe um ponto de acesso a elementos de um objeto agregado ou uma lista de objetos de forma sequencial sem que haja necessidade de exposição de detalhes da implementação desses elementos (GAMMA *et al.*, 1995).

- *Mediator*: propõe a criação de um objeto que encapsule a forma de interação de um conjunto de outros objetos (GAMMA *et al.*, 1995).

- *Memento*: propõe o armazenamento de um objeto interno sem que haja violação de encapsulamento, para que esse objeto possa ser recuperado a este estado futuramente (GAMMA *et al.*, 1995).

- *Observer*: padrão que propõe a composição de um objeto em uma árvore hierárquica de objetos dependentes. Quando este objeto muda de estado, todos os objetos desta árvore são notificados, e seu estado é atualizado automaticamente (GAMMA *et al.*, 1995).

- *State*: este padrão propõe que um objeto altere seu comportamento assim que seu estado muda, o efeito é como se a classe do objeto mudasse (GAMMA *et al.*, 1995).

- *Strategy*: propõe a criação de uma família de algoritmos encapsulados e intercambiáveis que são acessados independentemente do cliente que os utiliza por uma interface-padrão denominada *strategy* (GAMMA *et al.*, 1995).

- *Template method*: este padrão propõe que uma subclasse redefina certos passos de um algoritmo sem que haja mudança na estrutura deste algoritmo (GAMMA *et al.*, 1995).

- *Visitor*: este padrão promove a criação de novas operações através da adição de subclasses na hierarquia da classe na qual essa nova operação irá atuar (GAMMA *et al.*, 1995).

Saiba mais

Para aprofundar seus conhecimentos em padrões de projeto, consulte:

BUSCHMANN, F.; HENNEY, K.; SCHIMIDT, D. *Pattern-oriented software architecture*: a pattern language for distributed computing. New York: Wiley, 2007. v. 4.

GAMMA, E. *et al. Design patterns*: elements of reusable object-oriented software. Boston: Addison-Wesley, 1995.

A maneira mais simples de entender e assimilar *design pattern* é buscar um padrão dentro deste catálogo, entender qual é seu objetivo, ler o seu modelo de classes e implementar, na forma de prova de conceito, na linguagem orientada a objetos de nossa preferência.

Lembrete

A implementação de padrões de projeto, ou *design pattern*, é independente da linguagem de programação. Essa é uma das grandes vantagens de utilizarmos padrões.

O padrão *Singleton*, por exemplo, tem por objetivo utilizar uma única instância de uma classe em todo o sistema. A figura seguinte representa a documentação do padrão *Singleton*.

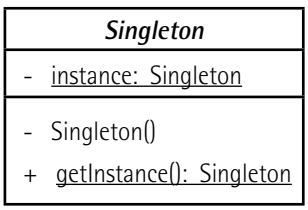


Figura 36 – Padrão *Singleton*

Agora vamos entender cada ponto que o padrão sugere:

- Repare que o construtor da classe é privado. Por que privado? Porque nenhum outro objeto terá acesso a esse construtor, logo não poderá instanciar essa classe.

- Temos um método público (e estático), de nome *getInstance*, que retorna um objeto do tipo da própria classe em que está criado. Por que público e por que estático? Público para que outros objetos possam acessar e estático para que ele possa ser chamado sem que haja necessidade de instanciar a classe em que ele está declarado.

- Temos um atributo privado, de nome *instance*, do tipo da própria classe em que está criado. Por que do tipo da própria classe? Para que possamos armazenar uma instância dessa classe.

Quando estamos entendendo padrões de projeto, é importante que não deixemos passar despercebido nenhum detalhe, por exemplo, a visibilidade dos atributos, métodos e construtores.

A figura seguinte mostra a implementação desta classe utilizando *Microsoft .Net C#*.

```
public class Singleton
{
    private static Singleton instance;

    private Singleton() { }

    public static Singleton getInstance
    {
        if(intance == null){
            instance = new Singleton();
        }
        return instance;
    }
}
```

Figura 37 – Implementação *Singleton* em C#

Na figura apresentada, podemos notar que o detalhe está na implementação do método *getInstance*, no qual se verifica se a variável responsável por armazenar a instância da classe é nula; em caso positivo, cria-se a instância (para primeira utilização), e, caso contrário, retorna-se a instância, garantindo assim que essa classe seja instanciada apenas na primeira vez.

Saiba mais

Aprofunde seus estudos sobre *design pattern* com o livro:

FREEMAN, E *et al. Head first design patterns*. Sebastopol: O'Reilly, 2004.

Saiba como implementar padrões de projeto utilizando o Microsoft .Net Framework em:

DOFACTORY. *.NET design pattern framework 4.5*, [s.d.]. Disponível em: <http://www.dofactory.com/products/net-design-pattern-framework>. Acesso em: 4 maio 2015.

6.2 Visão dinâmica

O objetivo principal da visão dinâmica da arquitetura é representar a realização dos casos de uso, que fundamentalmente se dá pela troca de mensagens entre os objetos no decorrer do tempo.

O primeiro passo para se determinar como uma tarefa será executada é definir as responsabilidades de cada objeto.

6.2.1 Definindo responsabilidades

Como vimos, o estilo arquitetural em camadas tem, entre outros objetivos, o de subdividir e agrupar as classes em grupos de acordo com as suas responsabilidades.

Vimos que, ao fazer isso, classificamos as classes como: entidade (*entity*), controle (*control*) e fronteira (*boundary*) e cada uma delas possui uma atribuição bem-definida no contexto do sistema.

Voltando ao exemplo do cadastro de funcionários, após a reorganização (também chamada de *refactory*) do seu sistema, suponhamos que você tenha chegado ao seguinte modelo de classes de projeto:

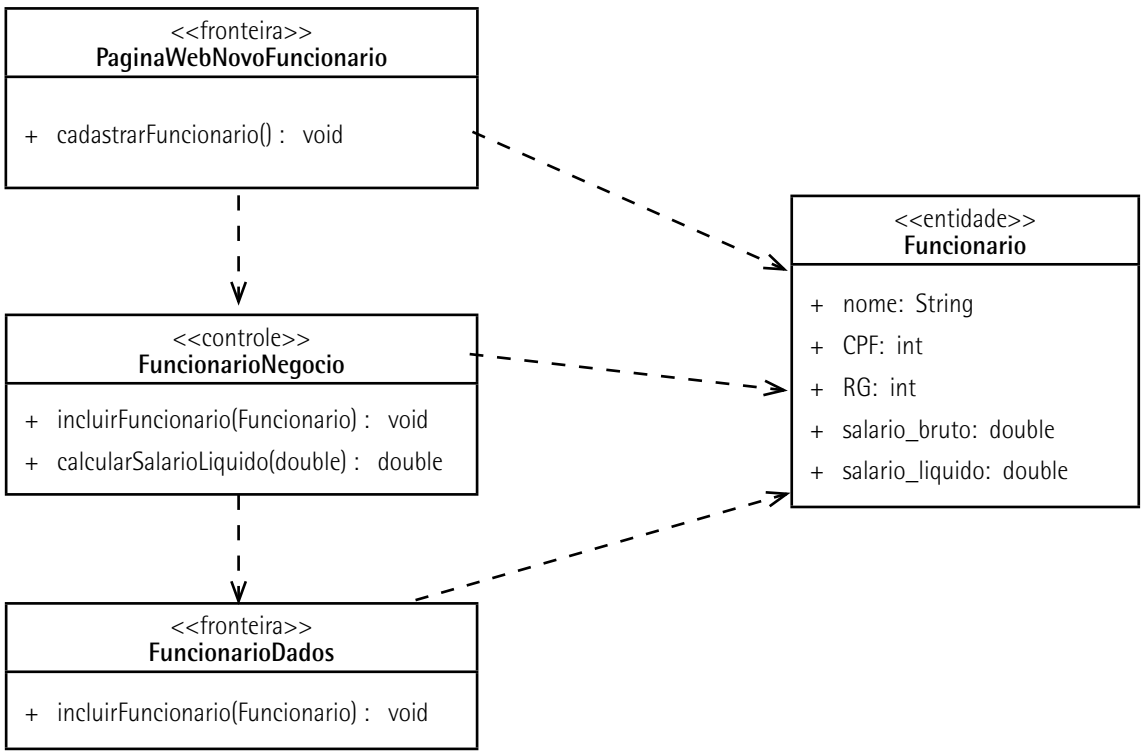


Figura 38 – Modelo de classes de projeto com responsabilidades

Note que agora temos todas as classes referentes ao nosso problema: a página *web* ("PaginaWebNovoFuncionario"), a classe responsável pelas regras de negócio ("FuncionarioNegocio"), a classe responsável pelo armazenamento dos dados no banco de dados ("FuncionarioDados") e a nossa entidade "Funcionario".

No entanto, ainda precisamos representar o fluxo de mensagens dessas entidades. É necessário representar quais objetos estão envolvidos em uma troca de mensagens, quais são essas mensagens trocadas e o significado dessas mensagens. Precisamos, por exemplo, saber qual evento da tela é o gatilho para todo o processo de cadastrar funcionário.

Para isso, precisamos pegar os conceitos e os diagramas de sequência que fizemos até então e adaptá-los, refiná-los adicionando as novas classes e as suas respectivas responsabilidades.

6.2.2 Refinando o diagrama de sequência

Antes de refinar o diagrama de sequência, precisamos saber como representamos as classes, ou objetos, de acordo com suas respectivas responsabilidades no diagrama de sequência. A figura a seguir mostra os estereótipos utilizados nessa representação.

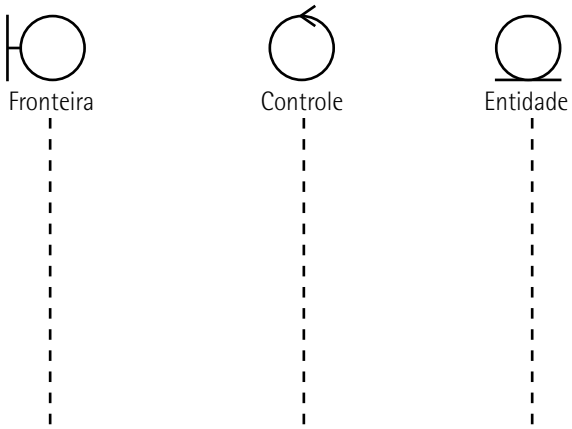


Figura 39 – Estereótipos de responsabilidades no diagrama de sequência

Agora podemos refinar ou desenvolver o diagrama de sequência representando corretamente as classes, suas respectivas responsabilidades e os significados de suas mensagens, como mostra o exemplo da figura a seguir que representa a visão dinâmica do processo de cadastrar funcionário:

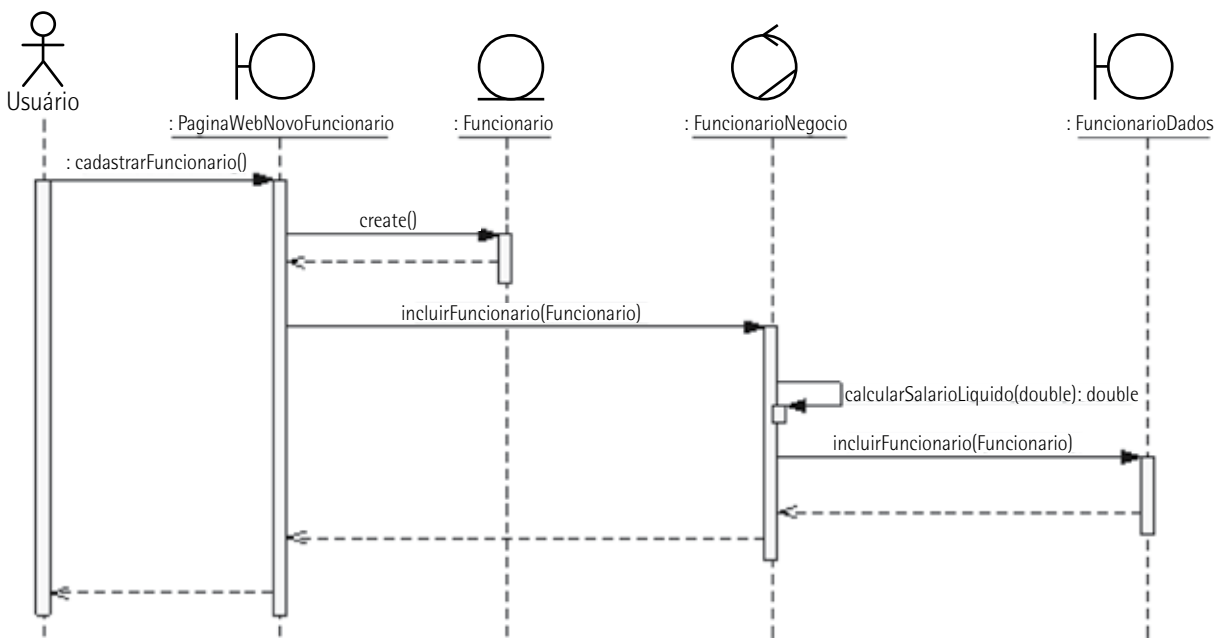


Figura 40 – Diagrama de sequência refinado

6.3 Documentação de arquitetura

Como vimos, um sistema de *software* é um conjunto de componentes dispostos em uma estrutura, que diz como eles estão arranjados, suas composições, dependências e ainda como eles se relacionam para resolver um determinado problema, além de definir como estes componentes estão distribuídos em uma infraestrutura (BASS; CLEMENTS; KAZMAN, 2010).

Visões arquiteturais são representações de cada uma dessas estruturas, e cada uma dessas visões terá uma forma específica de documentação. Neste livro-texto utilizaremos a documentação das visões arquiteturais a partir da UML.

Saiba mais

Existem outras abordagens de documentação de arquitetura, por exemplo, o modelo RMODP. Para aprofundar seus conhecimentos sobre esse modelo, leia:

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (ISO). *10746-3*: open distributed processing – reference model. Geneve: ISO, 2009.

Segundo Bass, Clements e Kazman (2010), são três as visões arquiteturais:

- Visão modular: representa a visão do sistema em termos de unidade de implementação; essas unidades podem ser classes, componentes ou módulos.
- Visão componente e conector: representa a forma pela qual os componentes interagem, ou seja, seus protocolos de comunicação.
- Visão de alocação: representa a forma pela qual esses componentes estão distribuídos em uma infraestrutura.

Para cada uma dessas visões, temos um diagrama, ou um conjunto bem-definido de diagramas da UML que nos auxiliam na modelagem e na documentação.

Por ora, atente-se em saber quais diagramas são utilizados para cada uma das visões; adiante, detalharemos cada um deles. O quadro seguinte mostra a relação entre os diagramas da UML e as visões arquiteturais.

Quadro 11 – Diagramas UML e visões arquiteturais

Visão arquitetural	Diagrama UML
Visão modular	Diagrama de pacotes Diagrama de classes
Visão componente e conector	Diagrama de componentes
Visão de alocação	Diagrama de distribuição

Além dos diagramas referentes às visões arquiteturais, que têm como função principal representar a visão estática da arquitetura, temos de documentar a visão dinâmica da arquitetura.

Para isso, é preciso um conjunto de diagramas da UML que servem como complemento ao diagrama de sequência, que vem a ser o principal diagrama para documentarmos a visão dinâmica de uma arquitetura.

Os diagramas a seguir são utilizados para documentar a visão dinâmica de uma arquitetura:

- Diagrama de sequência.
- Diagrama de colaboração.
- Diagrama de máquinas de estado.

Observação

A documentação da arquitetura não se resume apenas no desenvolvimento de diagramas UML. É importante que tenhamos uma organização e uma gestão de conteúdo que faça o relacionamento entre os diagramas produzidos e seus significados para o sistema, casos de uso que se deseja resolver e justificativas de solução.

Resumo

Nesta unidade, abordamos o ciclo de vida das atividades referentes ao projeto.

Lembramos que o ciclo de vida das atividades de projeto é dividido em quatro atividades: projeto de classes e dados, projeto arquitetural, projeto de interfaces e projeto de componentes.

A fase de projeto de dados tem o objetivo de definir um dicionário de dados, uma estrutura de informação necessária para a implementação de um sistema de *software*. Nesta fase, a ênfase dá-se em projetar um banco de dados.

Projetar um banco de dados é algo que não deve ser feito a esmo, e, como vimos, o projeto é dividido em três fases: projeto conceitual, projeto lógico e projeto físico.

A fase de projeto conceitual tem como objetivo produzir um modelo de alto nível de abstração que represente a estrutura do banco de dados; o projeto lógico tem como objetivo refinar este modelo de tal forma que acrescente detalhes inerentes ao SGBD; e o projeto físico tem como objetivo pegar este modelo e convertê-lo em um modelo no nível do SGBD; esta é a fase mais associada à tecnologia de todo o processo.

Na atualidade o modelo de dados relacional é um dos modelos mais utilizados em projetos de banco de dados, e, por isso, a ênfase foi dada à produção do modelo entidade relacionamento ou MER.

O MER é baseado em três pilares: as entidades, que são representações de entidades do mundo real, seus atributos ou características, e o relacionamento entre essas entidades.

Vimos que a adaptação do MER ao paradigma da orientação a objetos requer alguns cuidados, principalmente, porque o MER não possui características fundamentais à orientação a objetos, como armazenamento de métodos e herança.

Em contrapartida, esses paradigmas possuem semelhanças: os conceitos de entidades e atributos, que são análogos aos de classes e atributos, e o conceito de identidade de entidade, que pode ser considerado análogo ao conceito de identidade da orientação a objetos.

De qualquer forma, uma alternativa viável para armazenarmos os objetos em um modelo relacional é fazer um mapeamento de quais classes devem ser persistidas na base de dados; boas classes candidatas a isso são as do tipo entidade (*entity*).

Após a fase de projeto de dados e classes, passamos a debater a fase de projeto arquitetural.

Vimos que, analogamente à engenharia civil, a arquitetura, na engenharia de *software*, tem papel fundamental, apesar de nem sempre ser valorizada pelas organizações.

É na arquitetura de *software* que se projetam as estruturas, a constituição de um *software* que servirá como base fundamental para a fase de construção.

Existem muitas definições de arquitetura de *software*, na visão de Bass, Clements e Kazman (2010), que são uma das melhores referências no tema. Ela pode ser definida como uma representação, em alto nível de abstração, dos componentes de um sistema de *software*, as propriedades externamente visíveis destes componentes e como eles interagem com o objetivo de resolver as necessidades, ou o problema, de um determinado cenário de negócio (BASS; CLEMENTS; KAZMAN, 2010).

A arquitetura é muito importante, pois facilita a comunicação entre os envolvidos no projeto, representa em escala menor a estrutura do sistema, evidencia as decisões de projeto, além de facilitar a evolução, o reúso e a manutenção do *software*.

Existe uma lacuna entre a arquitetura e o gerenciamento de projeto que, como debatemos, não deveria existir, uma vez que a arquitetura auxilia a gestão do projeto quanto à estimativa de tempo, ao custo e à complexidade do projeto, além de auxiliá-la na mitigação e na diminuição de riscos.

Vimos que o processo de definição da arquitetura de um *software*, idealmente, deve iniciar-se ainda na fase de análise de requisitos e modelagem do domínio, na qual o arquiteto obtém informações importantes tanto a respeito do negócio quanto a respeito do cliente que influenciarão as decisões arquiteturais futuras. Esse processo de definição também pode ser subdividido em quatro fases: entendimento do problema, identificação dos elementos do projeto e de seus relacionamentos, avaliação da arquitetura e atualização da arquitetura.

Demos ênfase, nesta unidade, à fase de identificação dos elementos do projeto e de seus relacionamentos, que tem no arquiteto a figura humana central do processo. Esse papel, necessariamente, deve ser exercido por um profissional que possua uma série de habilidades e um perfil próprio.

A identificação dos elementos do projeto e de seus relacionamentos passa pela representação das duas visões da arquitetura de *software*: a visão estática e a visão dinâmica.

Na visão estática estruturamos o sistema em subsistemas e organizamos seus componentes internos; para isso, utilizamos uma ferramenta muito importante: os padrões.

Estilos arquiteturais e padrões de projetos são importantes soluções para um problema específico em um contexto específico, e essas duas variáveis são importantes nessa hora. Não devemos usar esses padrões simplesmente por usar.

Na visão dinâmica, modelamos como os componentes e os objetos se comunicam para resolver um determinado problema; na verdade, nesse ponto, apenas refinamos nosso diagrama de sequência com os conceitos de responsabilidade e organização arquitetural que vimos na visão estática.

Ambas as visões são registradas na documentação da arquitetura a partir de um conjunto específico de diagramas da UML.

Exercícios

Questão 1. Arquitetura de *software* é o nome dado à disciplina fundamental da fase de projeto arquitetural. Dentre os benefícios de uma arquitetura de *software* bem definida encontram-se:

A) Facilita o reuso de componentes, logo promove maior produtividade na construção e na manutenção, pois também facilita a manutenção.

B) Permite uma melhor análise dos requisitos funcionais durante a especificação do *software*.

C) Propõe um padrão de documentação, principalmente em linguagens orientadas a objetos.

D) Permite a evolução dos sistemas operacionais, SGBDs e *software* básico existentes.

E) Nenhuma das alternativas anteriores é correta.

Resposta correta: alternativa A.

Análise das alternativas

A) Alternativa correta.

Justificativa: dentre tantos benefícios de uma arquitetura de *software* bem definida, o reuso de componentes promove a produtividade na construção, pois permite reusar componentes já construídos e também na manutenção, já que os ajustes são realizados em poucos componentes ao invés de grandes blocos de código.

B) Alternativa incorreta.

Justificativa: arquitetura de *software* não se concentra em requisitos funcionais.

C) Alternativa incorreta.

Justificativa: padrões de documentação são tratados por outras disciplinas e não por arquitetura de *software*.

D) Alternativa incorreta.

Justificativa: a arquitetura de *software* objetiva o *software* a ser desenvolvido. O *software* básico de apoio como Sistema Operacional, SGBD e outros não têm suas estruturas avaliadas.

Questão 2. A arquitetura de *software* participa das fases do ciclo de vida do projeto de *software*. Com relação à fase de Pré-Projeto de *software*, analise as duas afirmativas apresentadas a seguir:

É comum observarmos que a arquitetura de *software* participa do ciclo de vida de desenvolvimento apenas a partir da fase de projetos. No entanto, é aconselhável que o arquiteto, ou a equipe de arquitetura, participe como observador na fase de análise de requisitos.

Porque

Essa participação possibilita a extração de informações técnicas importantes, por exemplo, padrões de arquitetura corporativa para integração de sistemas, que podem ser importantes para mitigação de custo, tempo de desenvolvimento e perfil da equipe.

Acerca dessas afirmativas, assinale a alternativa correta:

A) As duas afirmativas são proposições verdadeiras e a segunda é uma justificativa correta da primeira.

B) As duas afirmativas são proposições verdadeiras e a segunda não é uma justificativa correta da primeira.

C) A primeira afirmativa é uma proposição verdadeira e a segunda é uma proposição falsa.

D) A primeira afirmativa é uma proposição falsa e a segunda é uma proposição verdadeira.

E) As duas afirmativas são proposições falsas.

Resolução desta questão na plataforma.