Instituto Federal de Educação, Ciência e Tecnologia de São Paulo - IFSP
Câmpus Jacareí

Tecnologia em Análise e Desenvolvimento de Sistemas - ADS

2º Semestre de 2023

Engenharia de Software – JCRESW1

Prof. Lineu Mialaret

Aula 14: Orientação a Objetos - OO

Introdução

- O objetivo da Engenharia de Software (ES) é produzir soluções de software robustas e de alta qualidade que agreguem valor aos usuários.
  - Alcançar este objetivo requer a precisão da engenharia combinada com a sutileza da arte.
- Os projetos de software também têm diversos *stakeholders* com agendas concorrentes, o que aumenta a complexidade do gerenciamento de pessoas.
  - Gerenciar esta complexidade também faz parte do escopo da Engenharia de Software.

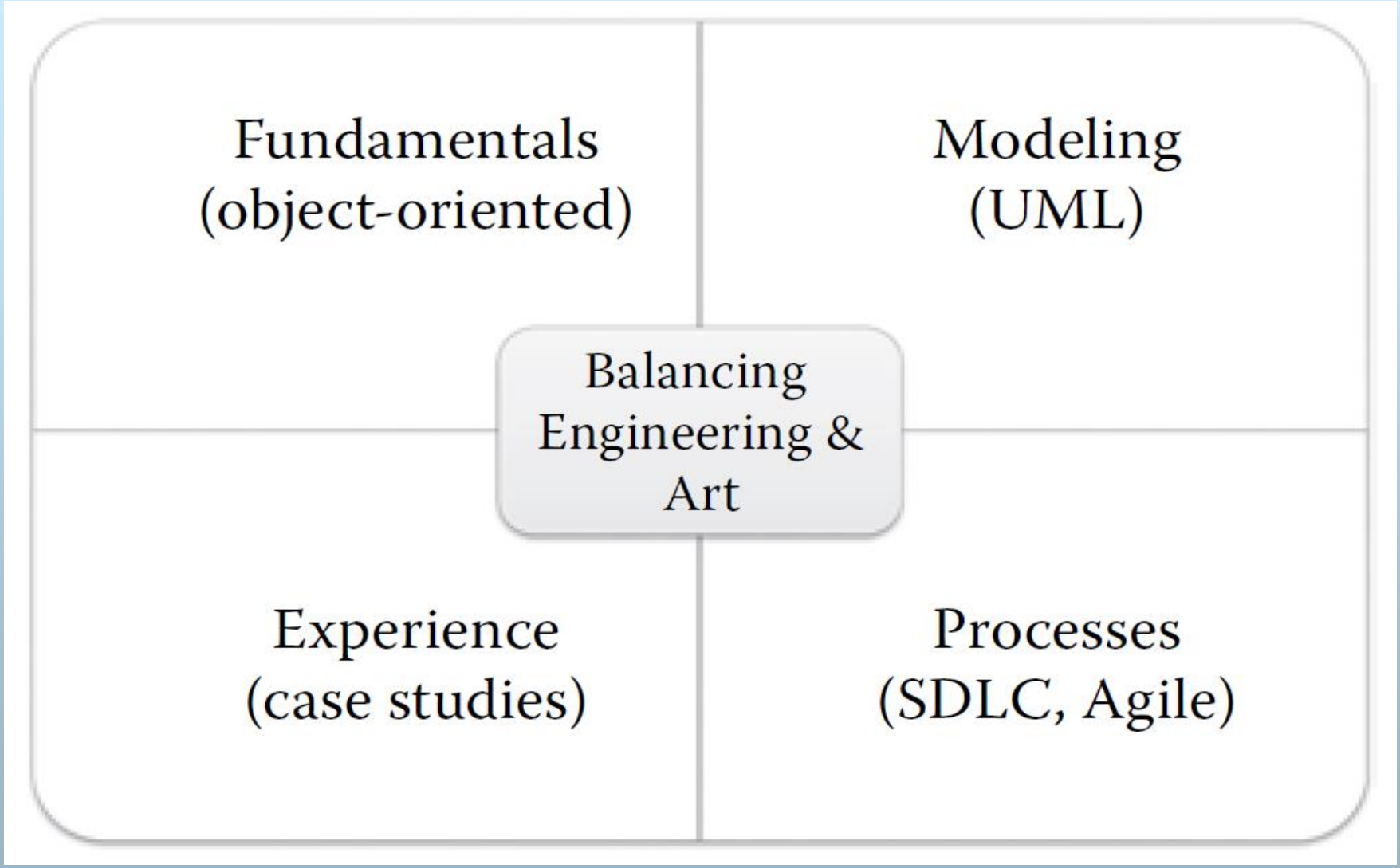
Introdução (cont.)

- A ES, nos seus primórdios, compreendia nada mais do que programação.
  - Porém, os resultados práticos obtidos nos ambientes de negócios demonstraram a necessidade de entender e analisar um problema e projetá-lo cuidadosamente antes da programação.
- Desta forma:
  - As práticas ágeis estendem ainda mais os requisitos de um sistema (no espaço do problema) de forma iterativa e incremental em um *design* no nível da solução (no espaço da solução).
  - As práticas de arquitetura de software (no espaço de arquitetura) garantem que o projeto detalhado da solução se ajuste ao ambiente corporativo antes de ser implementado.

ES Orientada a Objetos

- A ES engloba funções, atividades e tarefas, incluindo processos de desenvolvimento, gerenciamento de projetos, análise de negócio, modelagem de requisitos, *designs* de usabilidade, desempenho operacional, segurança, gerenciamento financeiro, gerenciamento regulatório e de conformidade, gerenciamento de riscos, garantia de qualidade, controle de qualidade, gerenciamento de versões e gerenciamento de serviços.
- Estudar e aprender ES é, portanto, um processo complexo em si que pode começar com o aprendizado dos fundamentos do desenvolvimento até a adoção da agilidade entre as equipes de projeto e a organização

ES Orientada a Objetos (cont.)



Fundamentos da OO

- Os conceitos fundamentais de Orientação a Objetos (OO), baseados nas primeiras linguagens de programação, como Simula e Smalltalk, fornecem uma base sólida para a Engenharia de Software Orientada a Objetos (ES OO).
  - Esses conceitos são ainda expressos em linguagens como C++ e Java.
- Todas essas tecnologias são mais bem utilizadas com uma forte compreensão conceitual dos fundamentos da OO.

Modelagem - UML

- Um padrão de modelagem que permite a criação de diagramas padronizados e especificações associadas contribui muito para melhorar a comunicação e aumentar a participação de todas as partes interessadas do projeto.
  - O aumento da participação das partes interessadas melhora a qualidade do software, reduz erros e incentiva a fácil aceitação da solução pelos usuários.
- A *Unified Modeling Language* (UML) fornece esse padrão necessário.
  - Com a UML, diagramas e especificações são criados, estudados, revisados e modificados pelas equipes de desenvolvimento, de forma compartilhada.

Modelagem - UML (cont.)

- Esses diagramas e modelos são fáceis de inserir em uma ferramenta de modelagem, também chamada de Ferramenta de Engenharia de Software Auxiliada por Computador (*Computer Aided Software Engineering* - CASE), para permitir que um grupo de usuários, analistas, *designers* e testadores trabalhem juntos.
  - A UML pode ser considerada um padrão de fato para modelagem de software.

Processos

- Um processo define atividades e fases e fornece direção para o desenvolvimento de software.
  - Esse ciclo de vida de desenvolvimento de software fornece orientação significativa no esforço de modelagem realizado por uma equipe de designers e desenvolvedores, pois eles são capazes de entender as atividades, tarefas, funções e entregas de toda a equipe do projeto no desenvolvimento, integração e liberação de uma solução.
- A agilidade é uma parte dos processos de desenvolvimento de software que se concentra intensamente em iterações e incrementos, colaboração, confiança e visibilidade.

Experiência

- Os fundamentos da ES e sua expressão por meio da UML são mais bem compreendidos por meio do aprendizado experiencial.
  - A experiência na criação de modelos UML, especialmente em um ambiente de equipe, é essencial para aprender ES.

Importância da Modelagem

- Os sistemas de software atendem a finalidades e funções comerciais específicas.
  - Esses propósitos comerciais são articulados pelos usuários.
  - Os usuários ou seus representantes são, portanto, parte integrante do processo de desenvolvimento.
  - Os usuários declaram suas necessidades, articulam os cenários de negócios e geralmente fornecem a visão do produto.
  - O desafio é que essas necessidades e requisitos sejam capturados, modelados e traduzidos em uma solução aceitável.
  - Lidar com esse desafio de desenvolver um software utilizável que seja aceitável para os usuários é o cerne do sucesso de um projeto de software.

Importância da Modelagem (cont.)

- Idealmente, todos os projetos de ES começam com uma compreensão dos principais objetivos de negócios derivados da solução.
  - Os projetos de ES, então, realizam análise (especificação de requisitos) e modelagem.
  - Essas atividades ajudam a esclarecer e modelar as necessidades e desejos dos usuários.
  - A análise é seguida pelo *design* da solução de software e eventual codificação baseada no ambiente técnico disponível e nos recursos da organização.
  - As atividades de análise e *design* são realizadas de maneira altamente iterativa e incremental.

Importância da Modelagem (cont.)

- O *design* fornece a ponte entre a análise e a codificação.
  - Um bom *design* é um canal suave que transforma requisitos em implementação.
  - Bons projetistas de software estão cientes das atividades de análise e do modelo de requisitos resultante e familiarizados com tecnologias (como programação e bancos de dados) e limitações impostas pela arquitetura da organização.
- Os modelos produzidos melhoram significativamente as comunicações dentro e entre as equipes de desenvolvimento.
  - Isso ocorre porque esses modelos representam requisitos no espaço do problema, projetos no espaço da solução e restrições no espaço arquitetônico.

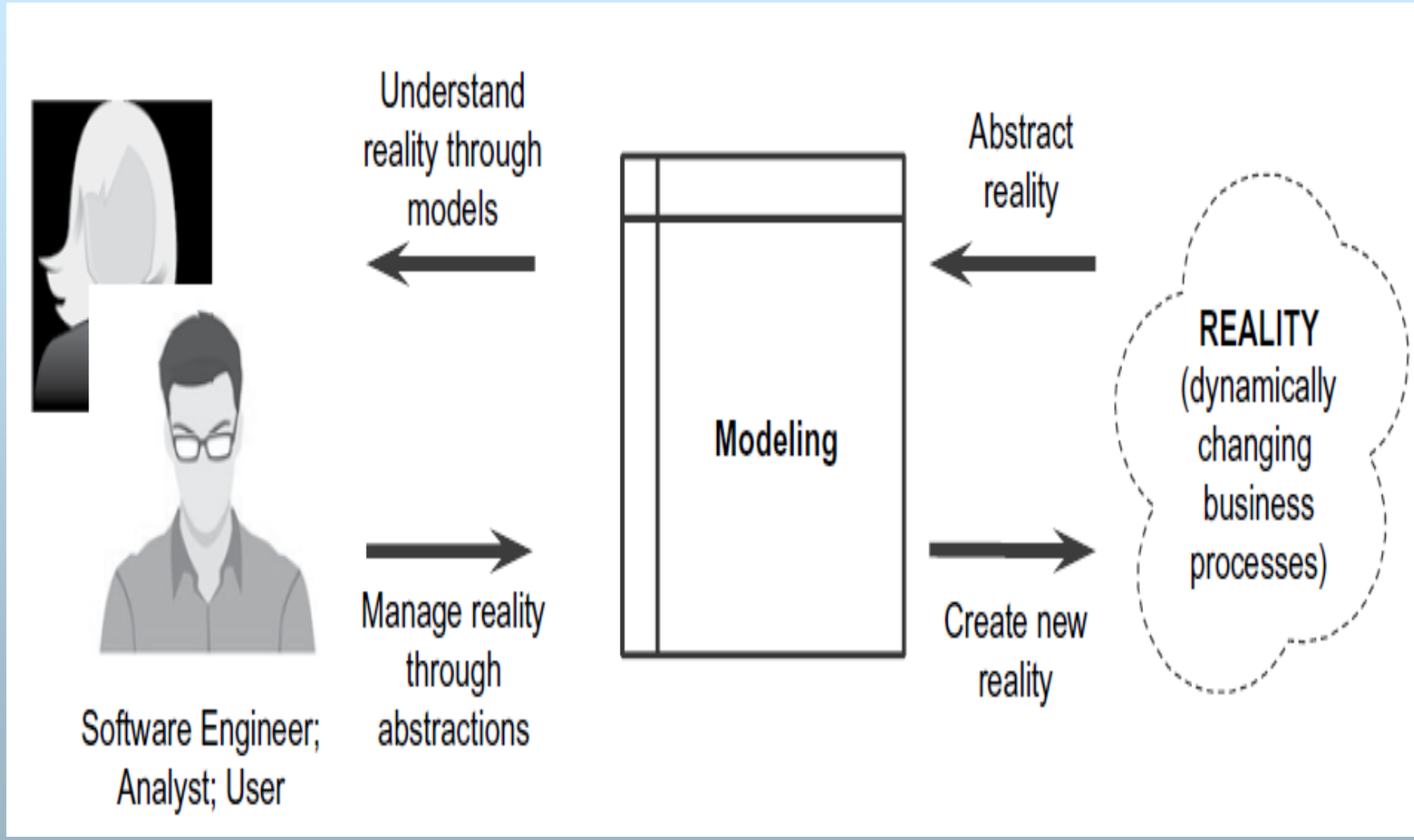
Importância da Modelagem (cont.)

- Em geral, os modelos fornecem à equipe do projeto grandes oportunidades para identificar lacunas, erros de compreensão, incompatibilidade de tecnologia e mudanças nas expectativas do usuário.
  - Os modelos permitem que as equipes façam tudo o que precisam antes de começar a codificação.
  - A codificação, ou programação, torna-se assim quase a última e talvez a menos cansativa atividade de todas em um projeto de engenharia de software bem organizado e bem modelado.
- Portanto, a modelagem e as comunicações decorrentes são cada vez mais vistas como uma das atividades mais importantes na produção de qualidade e valor em soluções de software.

Importância da Modelagem (cont.)

- A modelagem na ES serve a dois propósitos principais:
  - Ajudar a esclarecer a realidade de negócios existente; e
  - Criar uma nova realidade de negócios.
- A importância da modelagem em ES é multifuncional.
  - Entender os sistemas, aplicativos e processos existentes, seguido pela criação de novos processos, fornecendo uma base para testes e permitindo comunicações eficazes com todas as partes interessadas.
- As fases de manutenção e operacional de uma solução computacional também se beneficiam como modelos que fornecem uma abstração confiável do sistema que são mais fáceis de entender e alterar, em vez de alterar diretamente o sistema.

Importância da Modelagem (cont.)

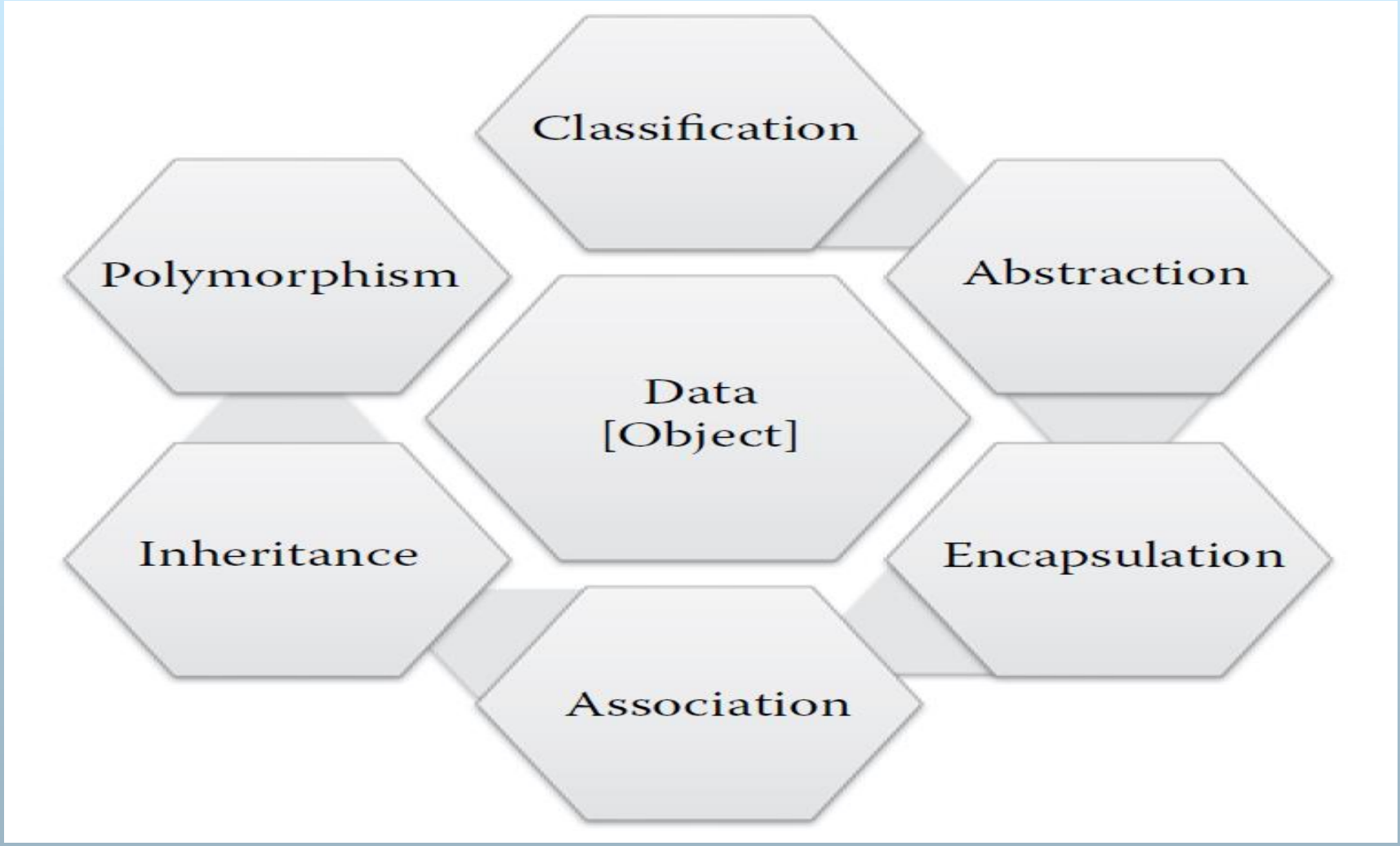


Processo de Modelagem

Conceitos Básicos da OO na ES

- Alguns fundamentos compõem o Paradigma da Orientação a Objetos (OO).
  - Esses fundamentos ajudam na criação de classes e programas que processam e manipulam dados e objetos.
- Conforme será mostrado a seguir, no centro da ES estão os dados.
  - Os fundamentos da OO giram em torno dos dados e extraem valor deles de várias maneiras.
  - Portanto, entender esses fundamentos da OO está no cerne de se tornar um bom engenheiro de software.

Conceitos Básicos da OO na ES (cont.)



Fundamentos Básicos da OO

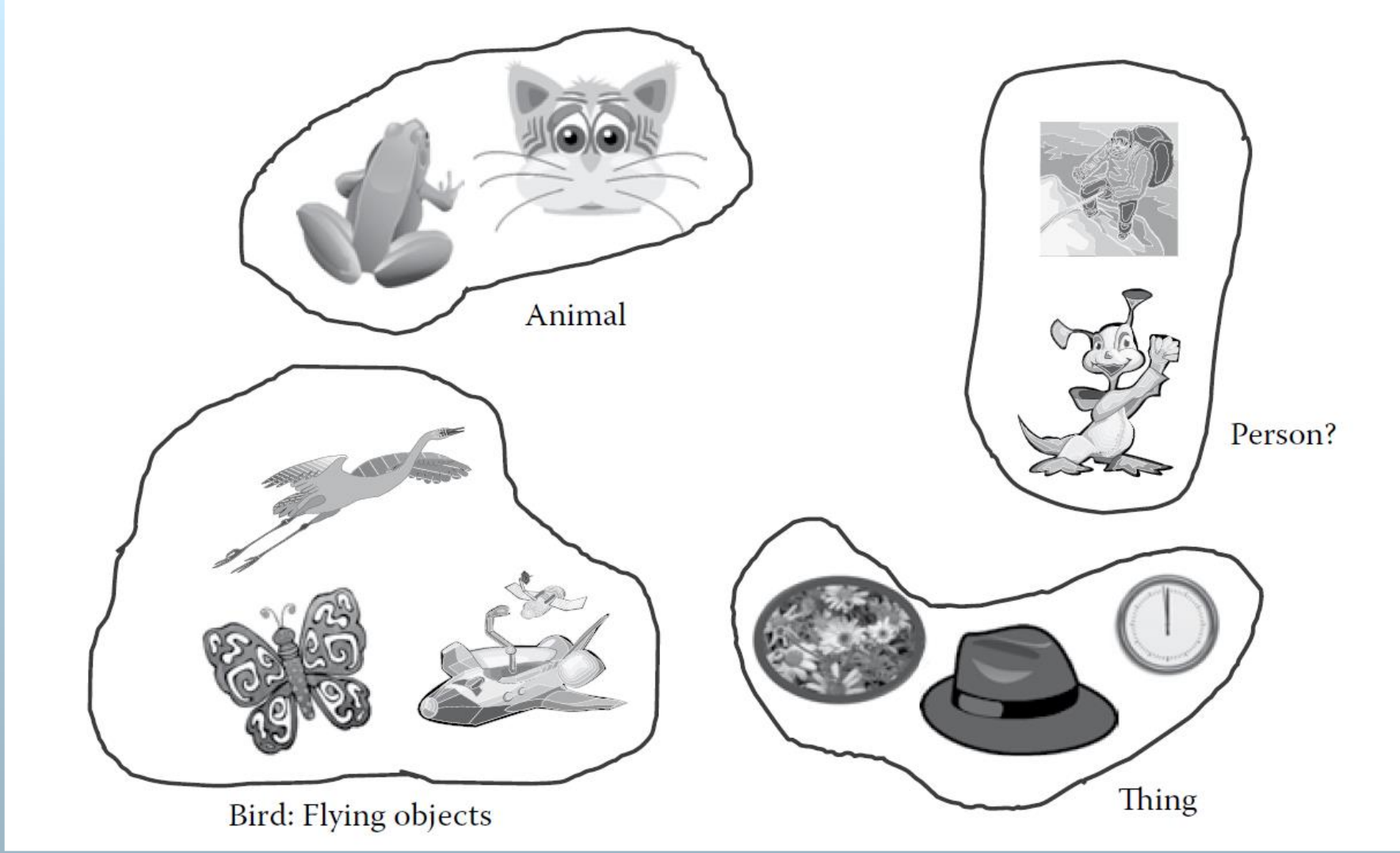
Classificação

- A classificação é o ponto de partida da OO.
  - Bons engenheiros de software entendem os requisitos, identificando primeiro as entidades (objetos) no espaço de negócios.
  - Uma vez que essas entidades ou objetos potenciais são identificados, eles são agrupados ou classificados.
  - A classificação é baseada nos requisitos que aparecem no espaço do problema, e esses requisitos, por sua vez, são modificados iterativamente com base na classificação.

Classificação - Exemplo

- Para um determinado conjunto de objetos como mostrado na figura a seguir, a borboleta e a garça são agrupadas sob o rótulo “pássaro”, o gato e o sapo sob o rótulo “animal”, o chapéu e o relógio sob o rótulo “coisa”. e assim por diante.
  - Essa classificação é baseada em um grupo de objetos que aparecem no espaço do problema.
  - Mudanças nos requisitos do sistema no espaço do problema alteram a base correspondente para classificação.
  - Por exemplo, se houver requisitos adicionais para objetos voadores, então, além da borboleta e do guindaste, o avião também será incluído nessa coleção de objetos, resultando em uma classificação diferente.

Classificação – Exemplo (cont.)



Abstração

- Objetos, que são entidades do mundo real, precisam ser representados por um modelo que também defina suas características (atributos) e comportamento (métodos).
  - Coleções de objetos classificados são abstraídas para classes.
  - Uma classe fornece uma definição detalhada de todos os objetos que podem ser instanciados a partir dela.
  - Este é o nível básico de abstração.

Abstração - Exemplo

The ones below are real objects with multiple instances. Each object has a unique identifier.

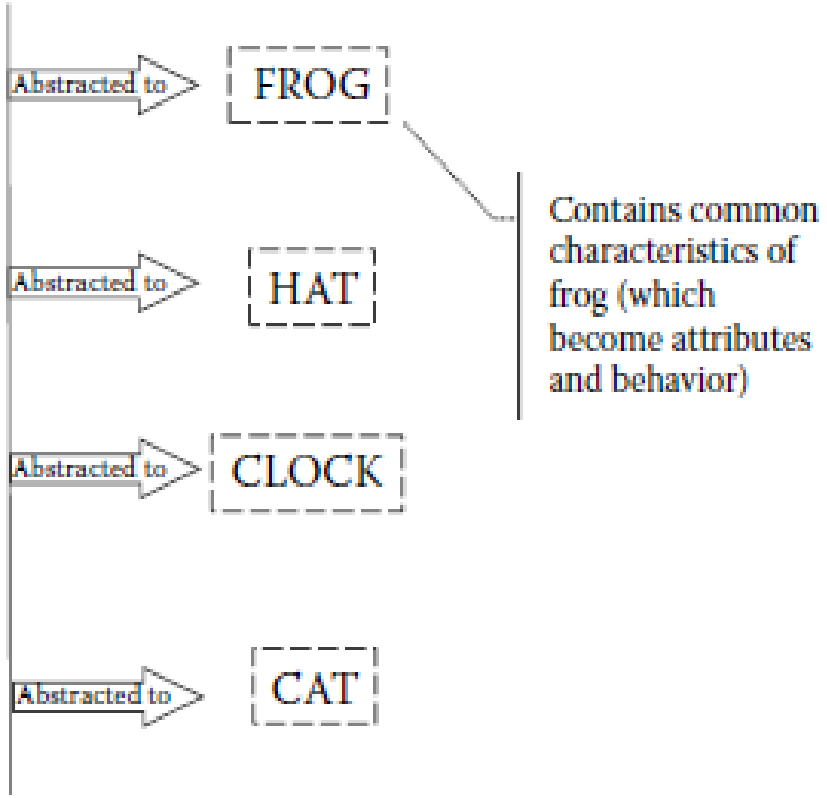
These names with boxes around them are *ABSTRACTIONS*. They form the basis for classes.

Frog Object-1, Frog Object-2, and so on...

Hat Object-1, Hat Object-2, and so on...

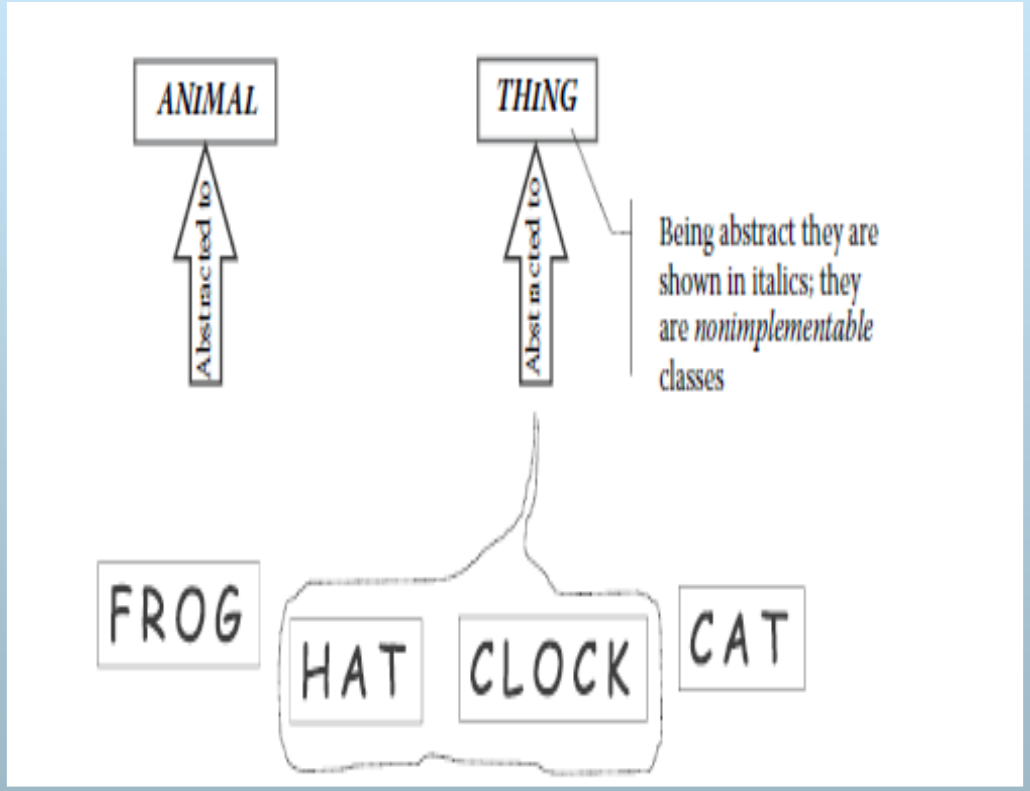
Clock Object-1, Clock Object-2, and so on...

Cat Object-1, Cat Object-2, and so on...



Good classification leads to creation of good abstractions.

Abstração – Exemplo (cont.)



- Embora uma classe seja uma abstração que representa uma coleção de objetos, as próprias classes estão sujeitas a outras abstrações.
  - Esse segundo nível de abstração é mostrado na figura ao lado, em que as classes *Frog* e *Cat* são abstraídas para uma classe de nível superior chamada *Animal*.
  - Da mesma forma, a classe *Clock* e a classe *Hat* são abstraídas para a classe *Thing*.

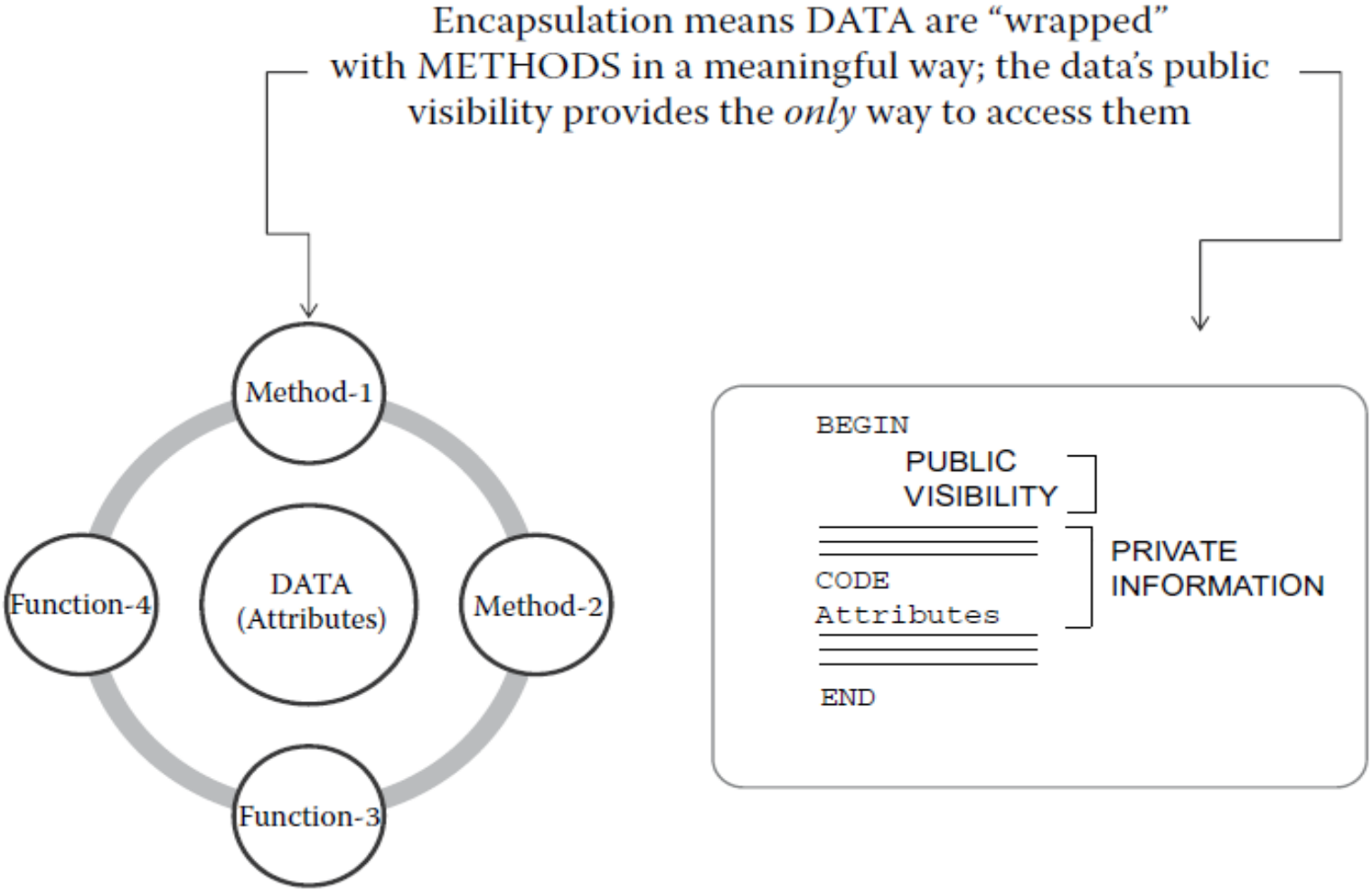
Abstração – Exemplo (cont.)

- Classificação e Abstração não são atividades isoladas.
  - Na verdade, eles estão intimamente relacionados, com uma alimentando a outra.
  - Classificação e abstração são aplicadas iterativamente para desenvolver modelos de software na prática.

Encapsulamento

- Encapsulamento é fundamental para envolver trechos de dados coesos com código significativo.
- O encapsulamento localiza os dados e evita que eles sejam expostos diretamente ao restante do sistema.
- O encapsulamento aumenta a qualidade e a reutilização porque os dados só são acessíveis por meio de chamadas para as operações (métodos ou funções) de uma classe.
- A próxima figura mostra um conjunto específico de “dados e códigos” que podem ser tratados como informações “privadas” pertencentes a uma classe.

Encapsulamento – Exemplo



Encapsulamento

Encapsulamento – Exemplo (cont.)

- Com o encapsulamento, um programador que reutiliza uma classe não precisa saber como um método é implementado para usá-lo (invocá-lo).
  - Tudo o que é necessário é o conhecimento da interface da classe (título e argumentos) e como chamá-la.
  - Isso é semelhante a chamar um serviço.
  - Os componentes orientados a serviços possuem documentação detalhada que ajuda a incorporar esses serviços em novos programas que estão sendo escritos.
  - A parte pública (interface), ou visível, da classe geralmente é um subconjunto de seus métodos ou funções.

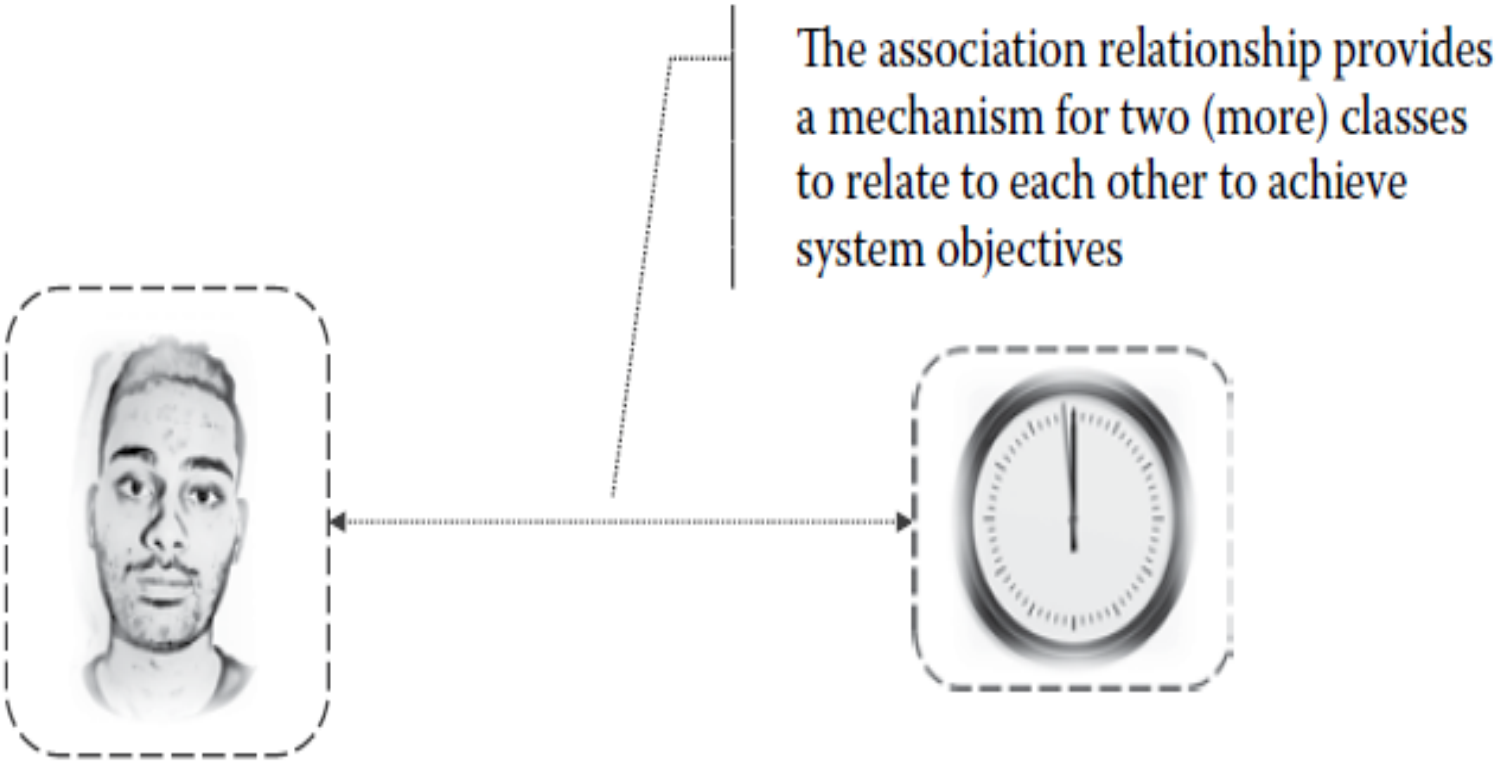
Encapsulamento – Exemplo (cont.)

- O encapsulamento também facilita a depuração de código.
  - Isso ocorre porque o código encapsulado é localizado.
  - Os erros podem, portanto, ser relativamente fáceis de restringir.

Associação

- Os objetos são classificados e abstraídos em classes.
- As classes não existem isoladamente.
  - Eles se relacionam com outras classes de várias maneiras.
  - O relacionamento de associação é um mecanismo para duas (mais) classes se relacionarem.
  - A figura a seguir, por exemplo, mostra a classe Pessoa associando-se à classe Relógio para atingir o objetivo de, diga-se, “obter o horário atual”.
  - Pode haver muitas associações adicionais entre Pessoa e Relógio, como “trocar as pilhas de um relógio”, “comprar um relógio” ou “acertar a hora”.

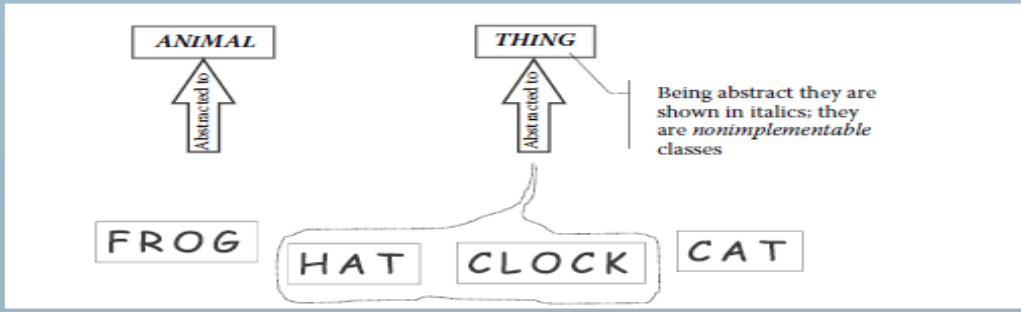
Associação - Exemplo



Associação

Herança

- As classes na OO também se relacionam por meio de herança.
  - A herança resulta de classes sendo generalizadas em classes de nível superior ou abstratas.
  - A abstração de segundo nível, mostrada na figura abaixo é o ponto de partida de boas hierarquias.
  - Uma classe pode herdar os atributos, comportamento e relacionamentos de outra classe.
  - A herança permite a extensibilidade de *design* e reutilização de código.



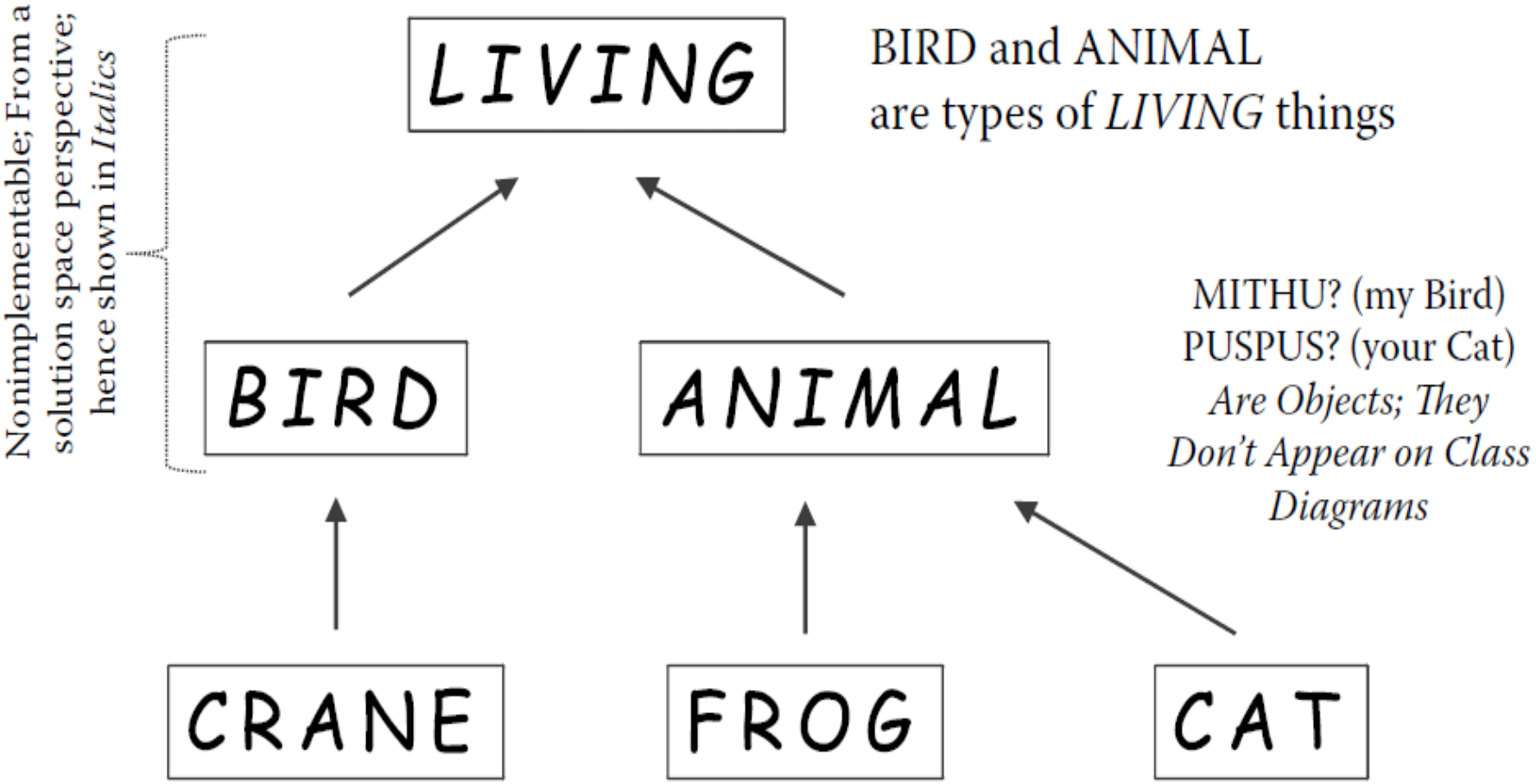
Herança - Exemplo

- Quando um conjunto de classes é abstraído para um nível superior, isso é chamado de Generalização.
- Quando uma classe é derivada de uma classe existente, ela é chamada de Especialização.

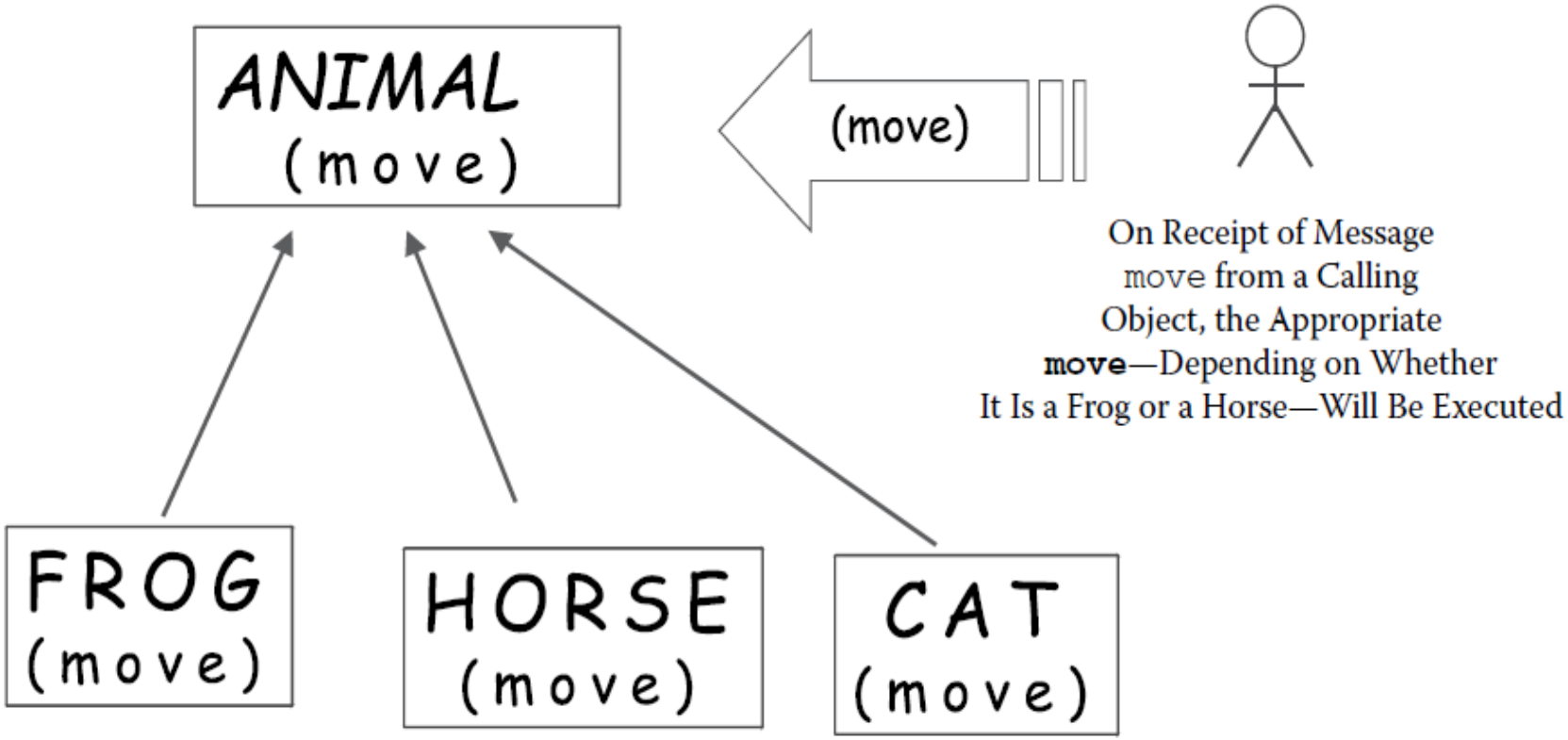
Herança - Exemplo

- A figura a seguir baseia-se no exemplo de classificação e abstração já mencionado anteriormente.
  - A classe *Animal* é herdada por *Frog* e *Cat*, resultando na especialização da classe *Animal*.
  - A classe *Bird* também é herdada por *Crane* com todos os seus atributos e relacionamentos.
  - Além disso, as semelhanças entre *Bird* e *Animal* são generalizadas em uma classe de nível superior chamada *Living*.

Herança - Exemplo



Polimorfismo



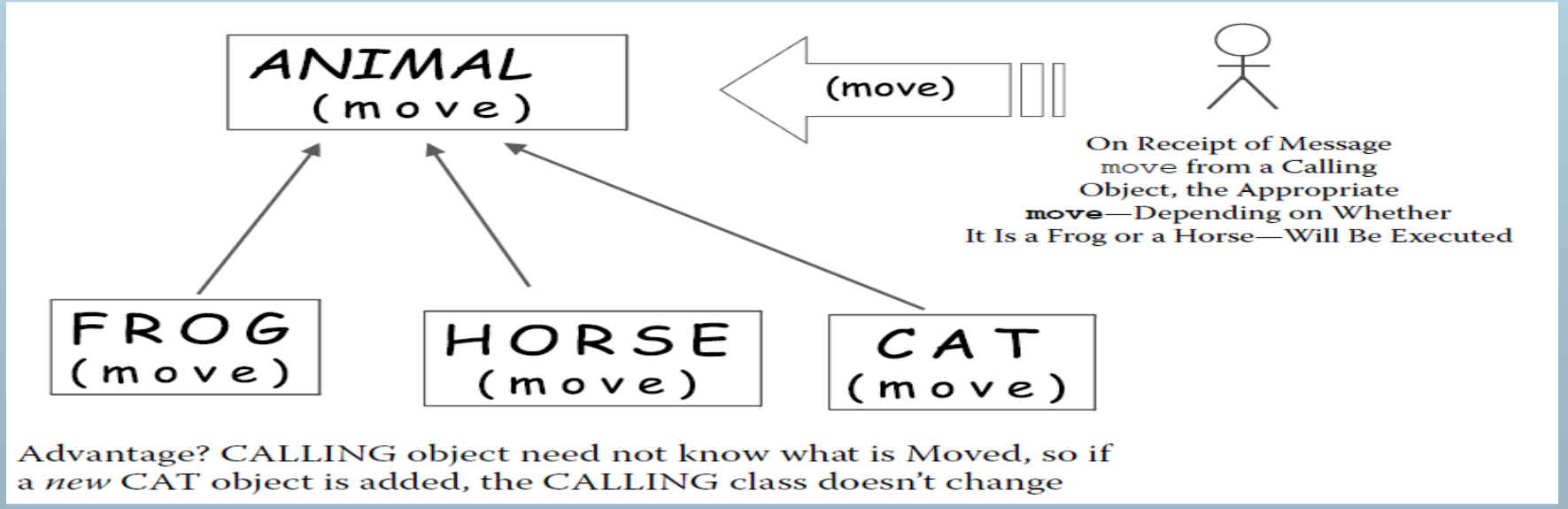
Advantage? CALLING object need not know what is Moved, so if a *new* CAT object is added, the CALLING class doesn't change

Polimorfismo - Exemplo

- Considere a figura anterior que mostra uma hierarquia de herança em que *Frog* e *Cat* são herdados da classe *Animal*.
  - Se um mensagem *move* for enviada para a classe Animal, do qual o objeto *Frog* é instanciado, então o objeto *Frog* executa seu “movimento” de maneira especializada (como um “salto”, por exemplo).
  - Este salto é implementado por baixo ou dentro da função de movimento para *Frog*.

Polimorfismo – Exemplo (cont.)

- Alternativamente, se um objeto *Horse* foi instanciado da classe *Animal*, então o código que implementa o movimento de um *Horse* é executado (um “galope”, por exemplo).
  - Os dois objetos, *Frog* e *Horse*, têm formas diferentes de realizar a mesma operação *move* quando chamados.



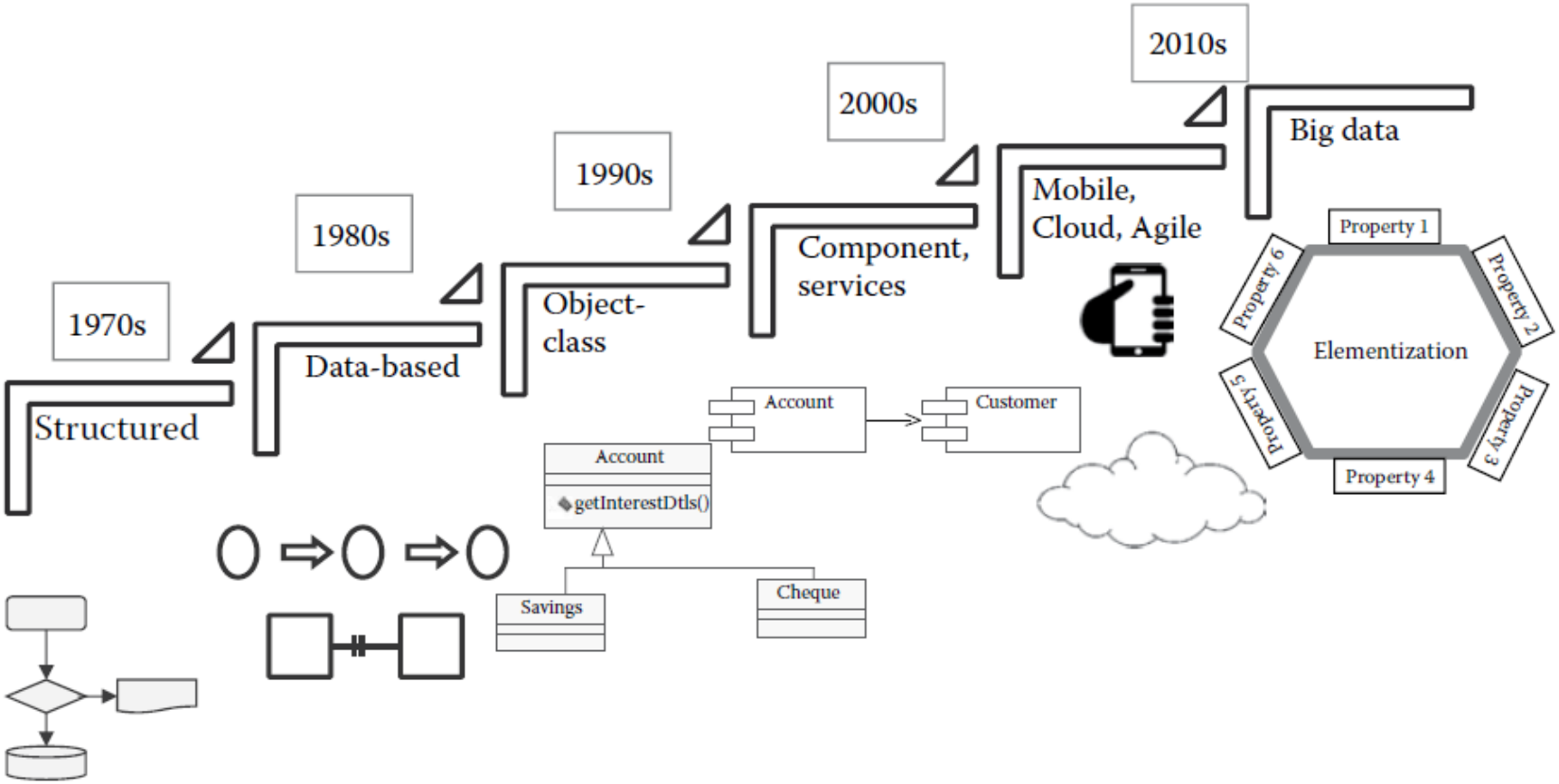
Polimorfismo – Exemplo (cont.)

- Os nomes dos métodos nas classes superiores (generalizadas) e derivadas (especializadas) precisam ser os mesmos para facilitar o polimorfismo.
- Deve-se observar também que o polimorfismo é uma característica de tempo de execução, em comparação com a herança, que é uma característica estrutural.
  - A herança no *design* torna possível o polimorfismo em tempo de execução.

Polimorfismo – Exemplo (cont.)

- Uma vantagem do polimorfismo é que o objeto chamador (ou seja, o objeto que está enviando a mensagem *move* para a classe *Animal*) não precisa saber qual objeto específico na extremidade receptora está executando o método *move*.
- Outra vantagem do polimorfismo é que quando surge um novo requisito para um objeto *Cat* no modelo mostrado anteriormente, o objeto que envia a mensagem *move* não muda.
  - Isso reduz a sobrecarga de manutenção e, em geral, melhora a qualidade do código e do *design*.

Evolução da Eng. de Software OO



Evolução da Eng. de Software OO

- A ES começou a evoluir na década de 1960, quando uma linguagem chamada Simula 67 foi desenvolvida no Norwegian Computing Centre.
- Simula 67 era uma linguagem de programação baseada em objetos com dados e operações armazenados juntos, permitindo código fortemente encapsulado e reutilizável.
  - Esse código tinha maior qualidade e precisava de menos tempo e esforço.
  - No entanto, esse esforço de programação não seguiu um modelo formal ou padrão em nível comercial.

Evolução da Eng. de Software OO (cont.)

- O Simula 67 foi seguido em meados da década de 1970 pelo Smalltalk no Centro de Pesquisa da Xerox Palo Alto (Xerox PARC)15.
  - Smalltalk compreendia uma linguagem e um ambiente de programação onde todos os seus elementos eram implementados como um objeto.
  - Smalltalk foi a primeira linguagem orientada a objetos completa e robusta que foi usada nos principais domínios industriais, como defesa e bancos.

Evolução da Eng. de Software OO (cont.)

- A década de 1970 também viu a chegada da COmmon Business Oriented Language (COBOL), que revolucionou o desenvolvimento de software comercial usado em setores como bancário, companhias aéreas e hospitais, para citar alguns.
  - COBOL, no entanto, era uma linguagem procedural que não aderiu a muitos fundamentos orientados a objetos, como encapsulamento, herança e polimorfismo.
  - As estruturas de dados subjacentes do COBOL foram suportadas principalmente por estruturas de arquivo de método de acesso sequencial de índice.
  - Isso foi rapidamente seguido por estruturas relacionais.

Evolução da Eng. de Software OO (cont.)

- Na mesma época em que Smalltalk começou a se tornar popular na década de 1980, também surgiu a necessidade de incorporar os conceitos de objetos em outra linguagem comercialmente popular chamada C.
  - O método de análise e projeto de sistemas estruturados (SSADM) tornou-se popular como uma abordagem para software de modelagem soluções.
  - C foi ampliado com recursos orientados a objetos que levaram ao C++, desenvolvido por Bjarne Stroustrup na AT&T, e Objective-C, desenvolvido por Brad Cox da Stepstone Corporation.

Evolução da Eng. de Software OO (cont.)

- Seguindo os sucessos dessas línguas, e fortemente influenciada pelo Simula, a língua Eiffel de Bertrand Meyer tornou-se comercialmente disponível no início dos anos 1990.
- Em meados da década de 1990, a *Unified Modeling Language* surgiu como um padrão para projetar soluções de software – desde a modelagem de objetos e classes até componentes e serviços.

Evolução da Eng. de Software OO (cont.)

- Mais tarde, na década de 2000, surgiram a computação móvel (incluindo aplicativos móveis) e a computação em nuvem – com base no conceito de oferta e consumo de serviços.
- A arquitetura de sistemas de software passou de um sistema único para um grupo colaborativo de serviços, reunidos de várias fontes tanto em *design* quanto em tempo de execução.
- O desenvolvimento e a implantação de soluções de software começaram a mudar para a nuvem.

Evolução da Eng. de Software OO (cont.)

- Desde 2001, os ciclos de vida de projetos baseados em Métodos Ágeis (*Agile Methods*) revolucionaram a maneira como as soluções são projetadas e implantadas.
- A mudança devido aos métodos ágeis está criando modelos extensivos para colaborar com os usuários.
- Por exemplo, visibilidade (baseada em colar e rastrear histórias de usuários em uma parede) e colaboração (essencialmente por meio de reuniões diárias) são técnicas ágeis altamente populares e bastante comumente aplicadas em projetos de ES.

Evolução da Eng. de Software OO (cont.)

- A era atual de Big Data inclui análises abrangentes que são realizados em dados estruturados e não estruturados que residem em bancos de dados NoSQL.
- Linguagens de programação como Python e R estão em voga.
- O fornecimento desses dados vai muito além da entrada humana e no sensor da máquina e no espaço IoT, e a exibição dos resultados da análise ocorre de inúmeras maneiras, incluindo e especialmente em dispositivos portáteis.

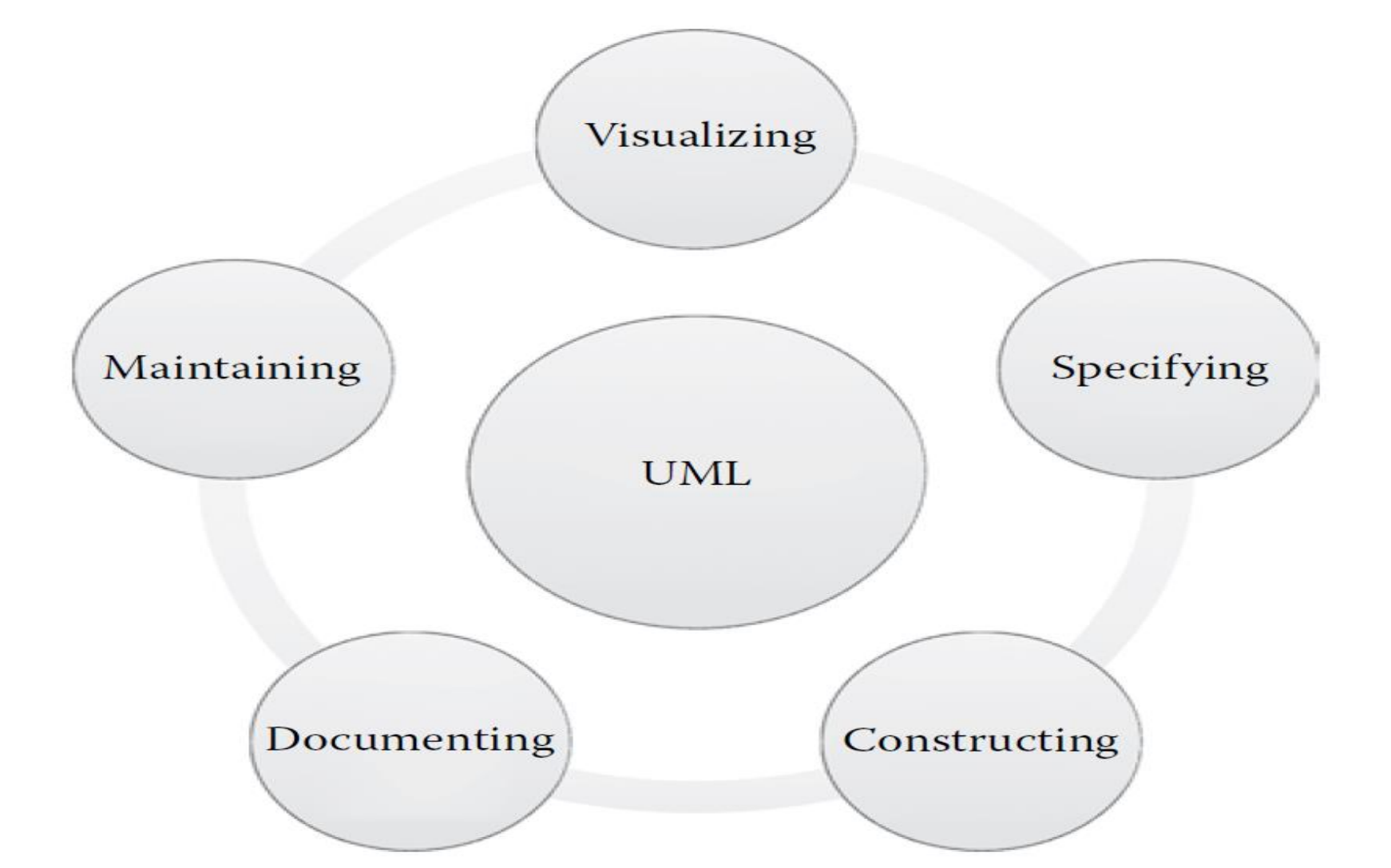
UML

- A UML não é o resultado de um único indivíduo, mas sim um esforço coletivo de vários profissionais, metodologistas, pensadores e autores.
  - O OMG (*Object Management Group*) facilitou essa entrada e incorporou os resultados em um metamodelo robusto, resultando em uma notação de modelagem padrão da indústria utilizável chamada UML.
  - A UML foi proposta pela primeira vez por volta de 1995 como uma combinação dos três métodos (processos) mais populares da época: Booch, Técnica de Modelagem de Objetos e Objectory.
  - Mais tarde, vários outros métodos se fundiram na UML, resultando eventualmente na popular versão 1.4 da UML.

UML (cont.)

- Por volta de 2004, uma versão UML 2.0 parcialmente formalizada foi lançada.
  - Essa versão da UML continha 13 diagramas oficiais e alterações correspondentes no metamodelo, levando a iniciativas como a arquitetura orientada a modelos (MDA).
  - Uma década depois, a UML 2.5, composta por 14 diagramas é considerada a linguagem de modelagem de fato para SE.

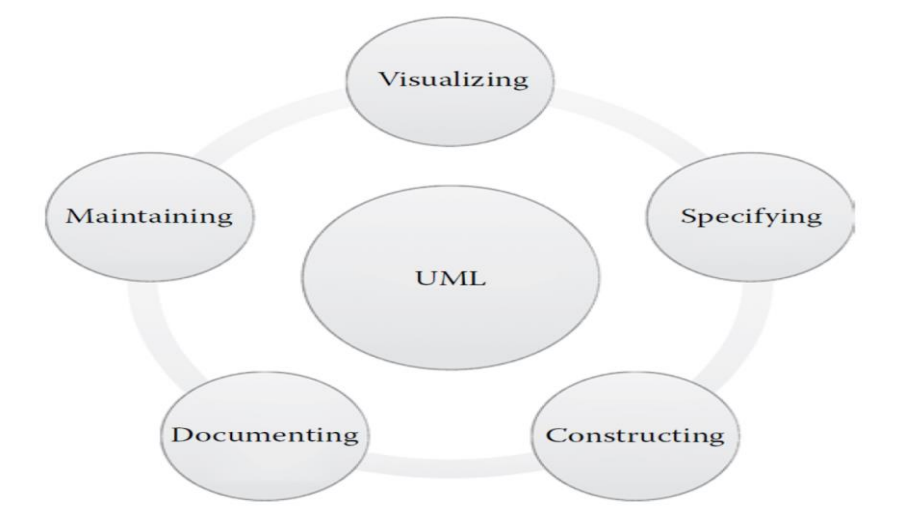
UML (cont.)



Características da UML

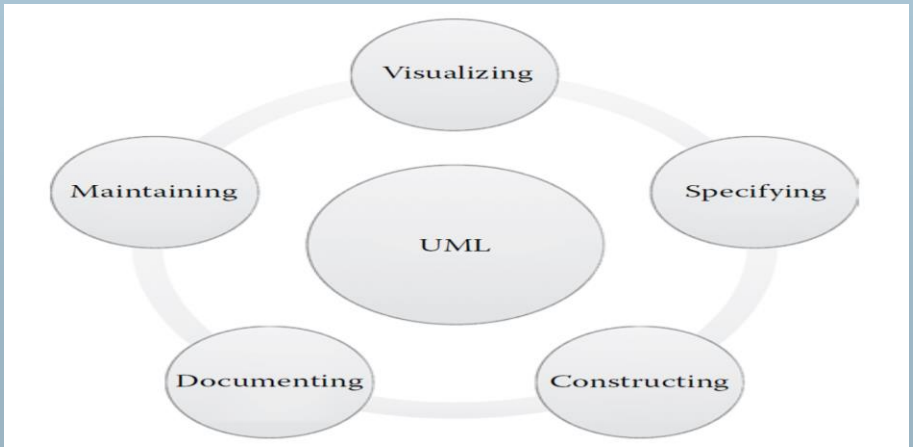
UML

- Visualização - esse é o objetivo principal da UML, pois suas notações e diagramas fornecem um mecanismo padrão para representação pictórica de requisitos, processos, *design* de solução e arquitetura.
  - Esses modelos visuais são criados usando ferramentas CASE, que também permitem o compartilhamento de trabalho de modelagem baseado em equipe.



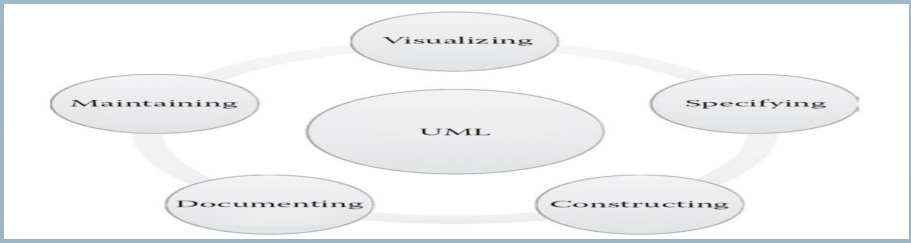
UML

- Especificação - A UML facilita a especificação de artefatos de modelagem.
  - Por exemplo, especificações para atores, casos de uso, classes, atributos e operações fornecem detalhes adicionais para notações visuais.
  - Essas especificações contribuem muito para melhorar a qualidade das soluções, pois as revisões das especificações ajudam a resolver muitos mal-entendidos entre os usuários e os desenvolvedores.



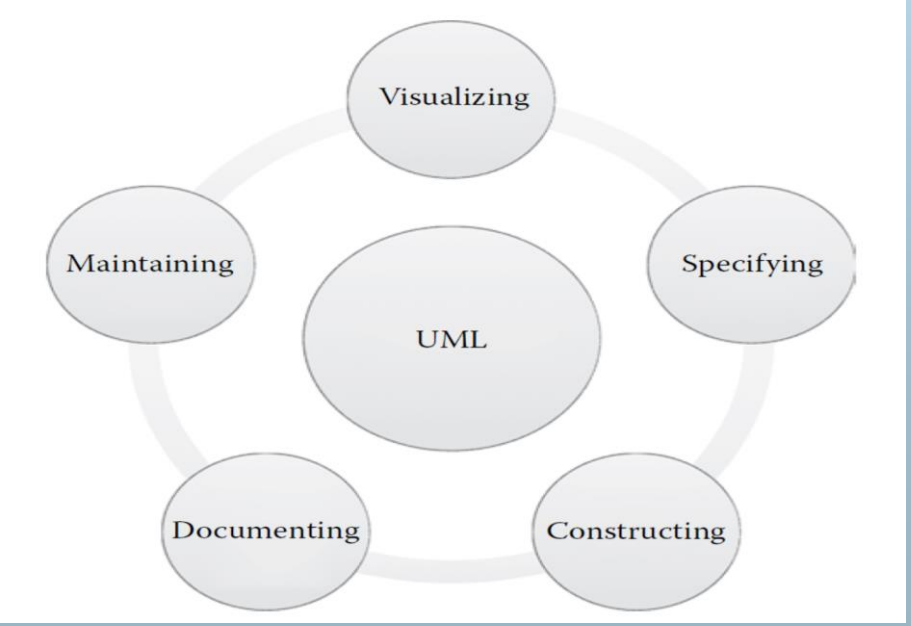
UML

- Construção – A UML é usada para construção de software porque permite a geração de código (por exemplo, C++ e Java), dependendo da ferramenta CASE que está sendo usada.
  - No entanto, esse recurso de construção da UML tem aplicação limitada, principalmente porque uma vez que o código é gerado, a maioria dos projetos práticos trabalha diretamente na modificação do código e não nos projetos.
  - A engenharia direta e reversas foi feita para ajudar a modificar os designs com base no código atualizado, mas esse recurso não é tão popular na prática quanto se pensava anteriormente.



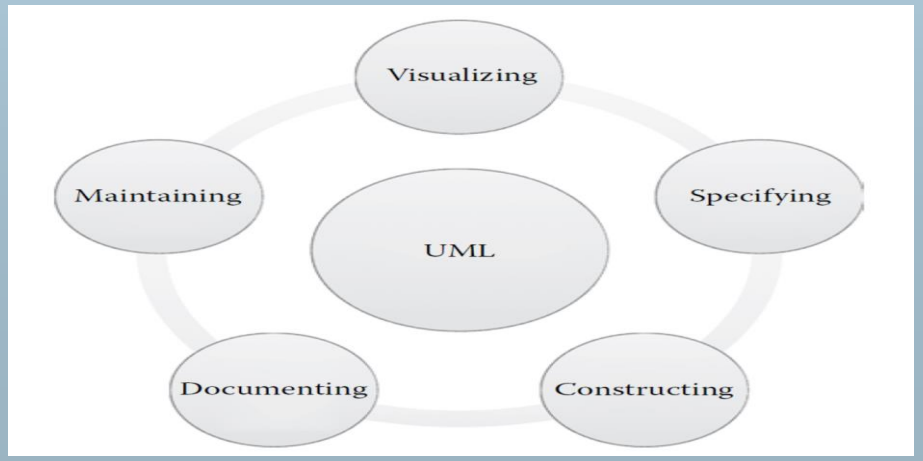
UML

- Documentação - Com a ajuda da UML, é fornecida documentação adicional e detalhada para requisitos, arquitetura, planos de projeto, testes e protótipos para aprimorar especificações e representações visuais.



UML

- Manutenção - Bons modelos UML são uma ajuda significativa na manutenção contínua de sistemas de software.
  - Os modelos permitem uma visualização fácil de um sistema existente e sua arquitetura.
  - Isso permite que os programadores identifiquem os locais corretos dentro do sistema para alterações e, mais importante, entendam o efeito de suas alterações no restante do sistema.



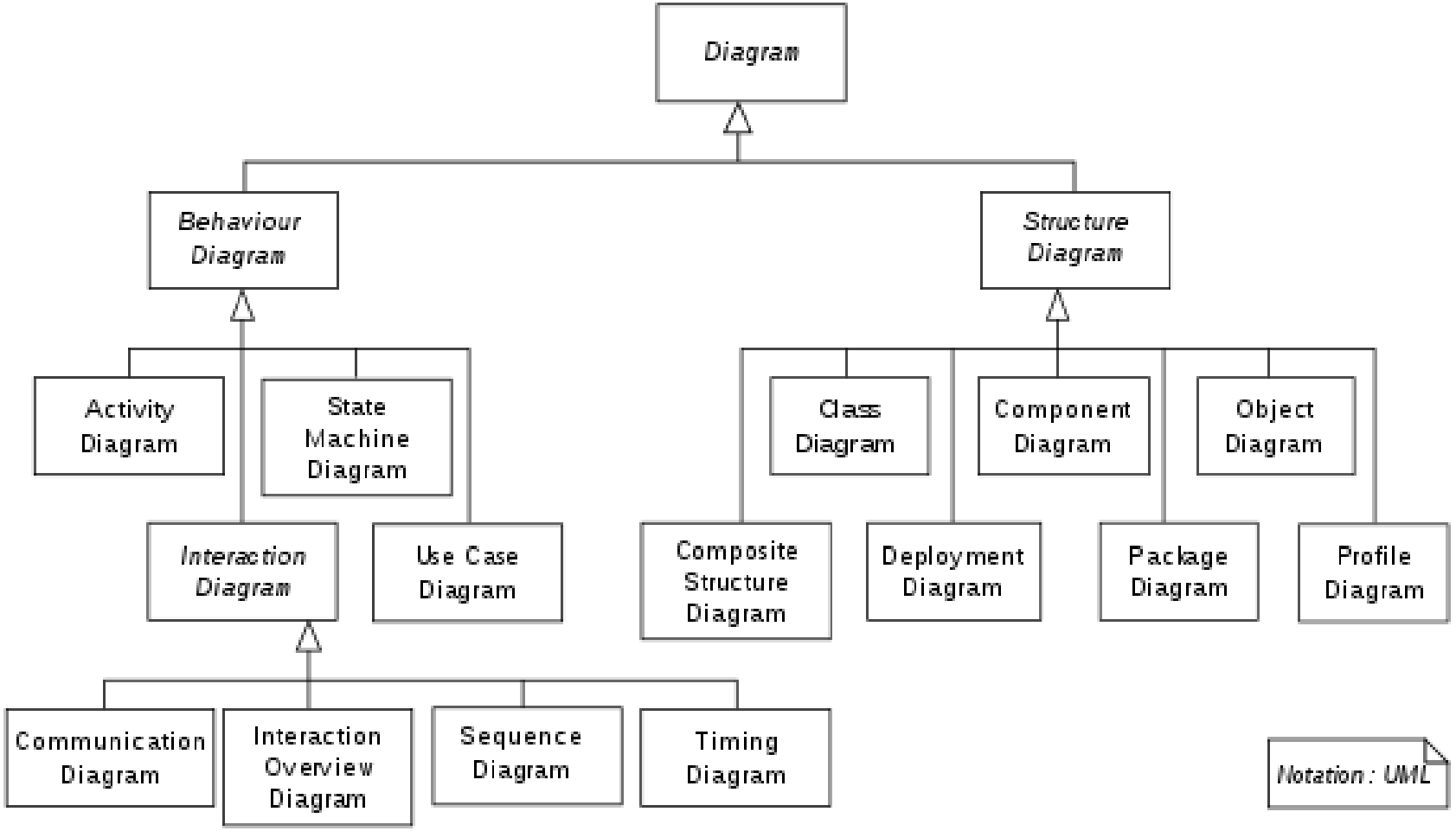
System perspectives

- **System perspective** is an external perspective, where ***you model the context*** or ***environment*** of the system.
- **System perspective** is an interaction perspective, where you ***model the interactions*** between ***a system and its environment***, or ***between the components of a system***.
- **System perspective** is a structural perspective, where ***you model the organization of a system*** or ***the structure of the data*** that is processed by the system.
- **System perspective** is a behavioral perspective, where ***you model the dynamic behavior of the system*** and ***how it responds to events.***

UML diagram types

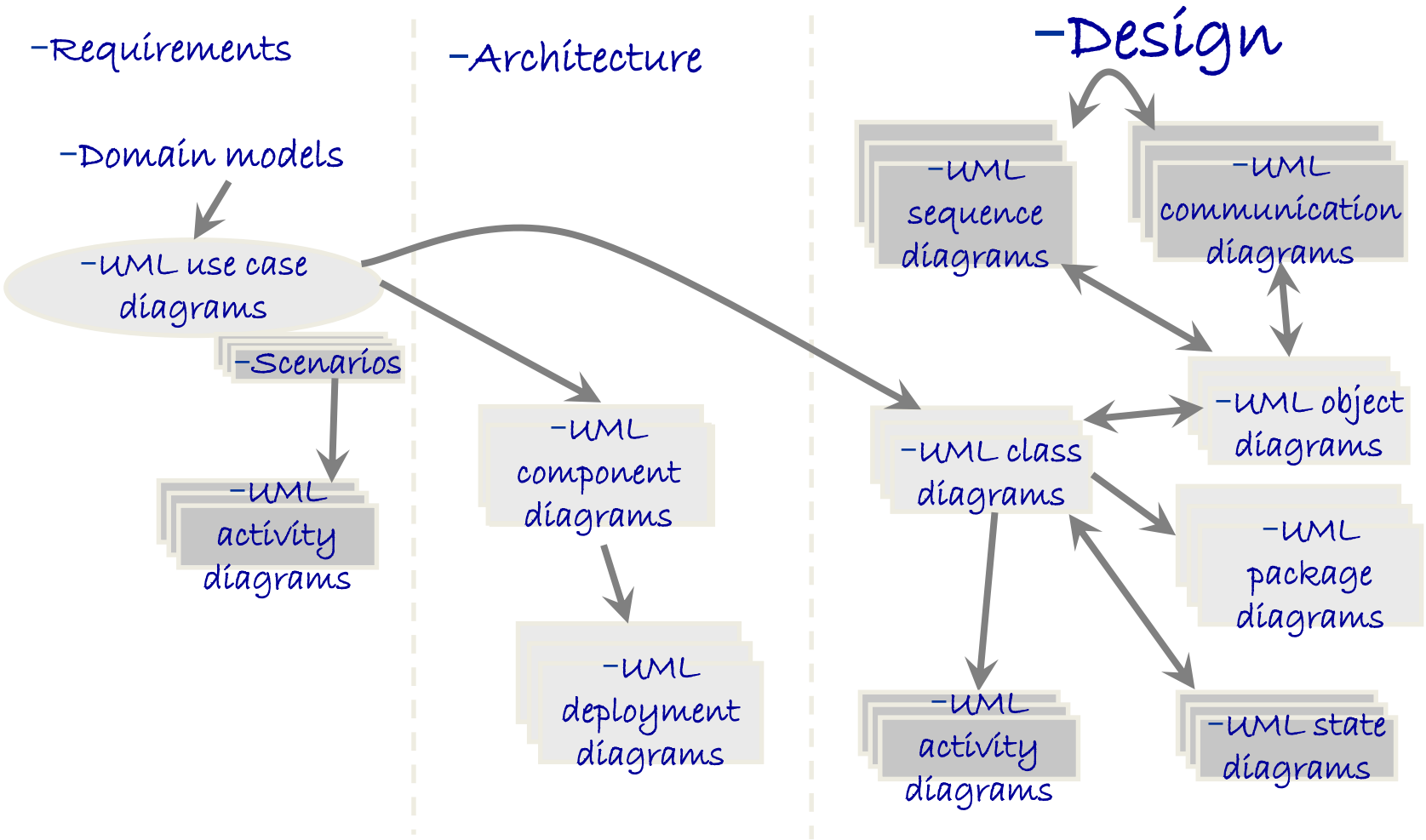
1. **Activity diagrams**, which show the **activities involved in a process** or **in data processing**.
2. **Use case diagrams**, which show the **interactions** between **a system** and **its environment**.
3. **Sequence diagrams**, which **show interactions between actors and the system** and **between system components**.
4. **Class diagrams**, which **show the object classes** in the system and **the associations** between these classes.
5. **State diagrams**, which **show how the system reacts** to internal and external events.

UML diagram types



UML in the SDLC Process

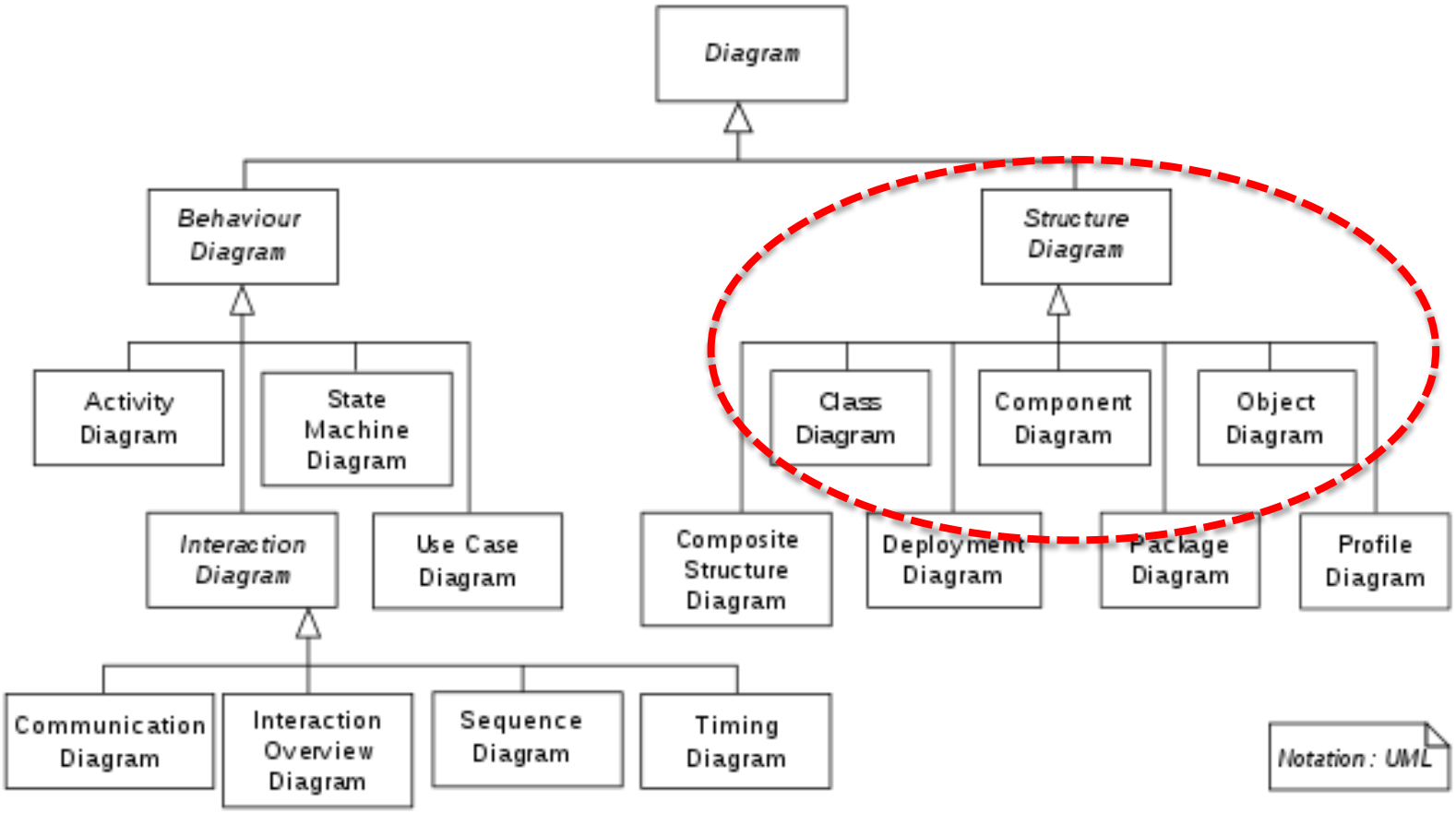
✧ How UML is used in the SDLC



Use of graphical models

- It is **good to use graphical models** to shorten discussions about an existing or new system:
  - Incomplete and incorrect models (designs) are OK.
  - Because their role is to support the discussions.
- It is **good to use graphical models** to ease documenting of an existing system;
  - Models should be an accurate/correct representation of the system but need it is not be complete.
- As a detailed system description that can be used to create a system application, the models developed must be both correct and complete.

Structural models



Structural models

- **Structural software models** show the organization of a system ***IN TERMS OF THE COMPONENTS OF THE SYSTEM*** *and* ***THEIR RELATIONSHIPS***.
- **Structural models** may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing.
- **You develop structural models of a system** when you are ***discussing and designing the system architecture***.

Class diagrams

- **Class diagrams are used** when developing an OBJECT-ORIENTED SYSTEM model to **show the classes** in a system and **the associations** between these classes.
- **An object class** can be thought of as **a general definition of one kind of system object**.

```
ClassName obj=new ClassName();
```

- **An association is a link** between classes that indicates that there is **some relationship between these classes**.
- When you are developing models during the early stages of the software engineering process, **OBJECTS represent** *something in the real world*, *such as a patient, a prescription, doctor, etc.*

Erros Fundamentais em ES

Common Error	*Rectifying the Errors*	*Examples*
Not being able to differentiate between a class and corresponding objects	Treat class as a shell or a template; object subscribes to that shell's definition.	Class = Person; Object = Sam, Mary, Ram.
Objects belong to a class, so they get treated as a subclass; objects get shown on a class diagram	Objects and classes cannot be shown on the same diagram because they are intrinsically different. Only subclasses derived from classes can be shown on a class diagram. Objects can be separately shown on object diagrams.	Class = Person; Subclass = Student; Objects belonging to Subclass Student = Sam, Zahid, Ram; Student "is a" Person can be shown on a class diagram (even though object Sam is also a person)
Data hiding vs. encapsulation	Don't write a method (function) to +set and +get every attribute; that is data hiding; instead, write meaningful operations that encapsulate multiple attributes and provide value to the calling classes.	For attribute Date, writing +setDate() and +getDate() is data hiding—which is not very helpful, but writing +getAge() is more meaningful and encapsulates date.

Erros Fundamentais em ES (cont.)

Common Error	*Rectifying the Errors*	*Examples*
Inheritance vs. directional association	Be clear about the arrowheads (also see Chapter 9 on basic class relationships). Association is between two classes with no commonality; inheritance is between classes that have commonality.	Use directional association when customer accesses account; use inheritance when customer is a person.
Overclassification	Study the requirements before classification. Otherwise every object can end up as a class on its own.	Revisit Figure 1.5 to study how classification is happening.
Assuming UML is a method or a process for developing software	UML is a modeling standard; it does not specify the sequence in which models are to be developed.	A method makes use of UML. A method (or process) for developing software can specify the starting UML diagram to be used for (say) capturing requirements or developing an architecture and the ensuing diagrams.