Ready for the Solution domain?

Analysing the Problem Domain

- **Problem domain** here corresponds to the analysis of the **essence of the problem**, independently of any type of solution or implementation
- **Solution domain** aims at proposing a solution for the problem, taking into consideration **physical technological constraints**
- Boundary between analysis (problem domain) and design (solution domain) is fuzzy
- Analysis = what
- Design = how

So, what do we know about the problem domain?

- System's functionality
- Domain concepts and their relationships
- Dynamic behaviour
- Static structure
- Constraints
- Domain complexity

3

UML techniques for design

- Many we already know
  - With more detail now
  - Our current sequence diagrams: amount of detail therein already suggests solution (algorithms, for example)
- A few new ones, for example
  - Component diagrams
  - Deployment diagrams
- **State transition diagrams** can be used for analysis and design, just like sequence diagrams

What do we need next?

- What are we still missing?
  - Structure, behaviour, constraints
- Anything else left unattended?
  - Complex requirements or requirements we do not quite understand?
- System's boundary
  - What is and what isn't?

State Transition Diagrams

- Solution domain
  - Persistency, concurrency
  - But also, security, performance, compatibility, …

5

Diagramas de Transição de Estado

Generic Steps for OOA

1. Elicit customer requirements for the system.
2. Organize these requirements using use cases.
3. Identify scenarios for use-cases.
4. Select classes and objects using basic requirements as a guide.
5. Identify attributes and operations for each system object.
6. Define structures and hierarchies that organize classes.
7. Build a behavioral model.
8. Review the OO analysis model against use-cases or scenarios.

Object-Oriented Design (1)

- Design of the system
  - Software architecture (application): structure the sytem into subsystems or components (component diagram)
  - Technical architecture (physical): hardware and software configuration (through a deployment diagram)
- Design of the objects
  - Refine the analysis models and propose a solution (in the implementation domain) for each artifact (e.g. Java, C++, C#)

Object-Oriented Design (2)

- Design of the system
  - Composition/decomposition (Component diagram)
  - Concurrency
  - Persistence
- Design of the objects/components
  - Component diagram (well-defined interfaces)
  - Detailed class diagram
  - Detailed sequence diagrams (if needed)
  - Use of design patterns
  - Use of frameworks

Design of the system

1. Partition the analysis model into components or subsystems (design packages). (Not needed for small systems)
2. Identify concurrency that is dictated by the problem.
3. Allocate components/subsystems to processors and tasks.
4. Develop a design for the user interface.
5. Choose a basic strategy for implementing data management.
6. Identify global resources and the control mechanisms required to access them.
7. Design an appropriate control mechanism for the system, including task management.
8. Consider how boundary conditions should be handled.
9. Review and consider trade-offs.

Generic Components for OOD

- **Problem domain component**: the subsystem responsible for implementing customer requirements directly;
- **Human interaction component**: the subsystem responsible for implementing the user interface (this includes reusable GUI subsystems);
- **Task Management Component**: the subsystem responsible for controlling and coordinating concurrent tasks;
- **Data management component**: the subsystem responsible for the storage and retrieval of objects.

Component Design Criteria

- Components should have a well-defined interface through which all communication with the rest of the system occurs.
- With the exception of a small number of "communication classes", the classes within a component should collaborate only with other classes within the component.
- The number of components should be kept small (at a given level of abstraction).
- A component can be partitioned internally to help reduce complexity.

Component diagrams
Package diagrams (design)

Concurrency

- Objects co-exist (in class diagrams and real world)
- Identify those objects that handle events simultaneously, with no interactions among them
- Response time can be improved (for real-time systems) using multitasking (if multiprocessing not possible);
- Using hardware may be a possibility (for certain systems)
- Use the dynamic models to identify concurrency (activity, sequence and state diagrams)

Components to processors or tasks

You need to

- Estimate the needs for performance and the resources required to satisfy them.
  - One single processor is enough? More than one? (the decision depends on the volume and machine)
- Decide whether any component should be implemented as hardware or software.
  - Hardware is less flexible, but is more efficient, fast and exact.
- Allocate the software components to processors to satisfy performance criteria, and reduce communication among processors, since :
  - Some components may have to be executed in specific physical locations
- Determine the physical connections and communication protocols between different physical units (deployment diagram)

Concurrency: identify tasks

- By analysing activity and state diagrams of several objects we can identify those that require a single control flow
  - Control flow passes from the sender object to the receiver object and is returned to the sender
- Different control flows are required when an object sends an event and continues its execution
- Each control flow is allocated to a task

Choose implementation control flow

- In analysis the interactions between objects are shown as events
- Decide on the control flow:
  - based on procedure-calls: control belongs to code
  - based on events: control belongs to an event manager that dispatches the events to be processed
  - parallel: control belongs to multiple tasks, independently

Persistency

- Database management systems
  - Advantages: powerful, portable, common interface for all applications, standard language, data integrity facilitated, as well as fault tolerance, multiple access, etc.
  - Disadvantages: complex, harder to integrate with programming languages, insuficient functionality for advanced applications (ex. CAD/CAM), lower performance.
- Files
  - Advantages: simple and cheap
  - Disadvantages: low-level code to handle the data; low portability (implementation platforms may vary).

Persistency

Which objects belong to a database?

- Those that are accessed by multiple users.
- Those that can be effectively handled through DBMS.
- Those that need to be portable.
- Those that need to be accessed by more than one application.

Persistency

Which data should be in files?

- Those that are difficult to structure according to the constraints of a DBMS.
- Large quantities and low density of information (history, archives).
- Raw data (summarized in a database).
- Those that need to be maintained for short periods only.

Boundary conditions

- Initialization
  - Constants, parameters, global variables, tasks, etc. should be initialized
- Termination
  - In a concurrent system, a task should let others know about its termination. A tasks should release locked external resources
- Fault tolerance
  - Unexpected situations should be considered

Design of the objects

- A *protocol description* establishes the interface of an object by defining each message that the object can receive and the related operation that the object performs
- An *implementation description* shows implementation details for each operation implied by a message that is passed to an object.
  - information about the object's private part
  - internal details about the data structures that describe the object's attributes
  - procedural details that describe operations

Desenho dos objectos: processo

- Desenhar os algoritmos.
- Optimizar caminhos de acesso a dados.
- Desenhar associações.
- Reutilizar classes já implementadas (bibliotecas) ou mesmo componentes (COTS).

Desenhar os algoritmos (1)

- A especificação de análise mostra o *que* a operação faz do ponto de vista dos seus clientes. O algoritmo mostra *como* fazer isso.
- Um algoritmo deve ser estruturado em chamadas a operações mais simples até que a operação de mais baixo nível possa ser implementada directamente.

Desenhar os algoritmos (2)

1. Escolher algoritmos.
2. Seleccionar as estruturas de dados.
3. Definir novas classes de apoio e operações internas (privadas).

Escolher algoritmos que minimizem o custo da implementação das operações

- Algoritmos complexos às vezes são necessários porque:
  - não há uma especificação procedimental da operação e nesse caso podemos ter que inventar um algoritmo;
  - é preciso optimizar funções descritas por algoritmos simples mas ineficientes.
- Muitas vezes, a definição mais simples da operação constitui o melhor algoritmo;
- Outras vezes, a definição da operação é tão ineficiente que precisamos encontrar um algoritmo melhor.
  - Por exemplo, procurar um valor num conjunto de tamanho *n*, com pesquisa sequencial precisamos em média *n/2* operações;
  - se a pesquisa for binária necessitamos de *log n* operações;
  - no entanto, se usarmos uma tabela de *hash* o número de operações é independente do tamanho da tabela e depende fundamentalmente da taxa de ocupação. Se a taxa de ocupação for de 80% o número de operações requeridas é 3 em média.

Escolher algoritmos

algumas considerações a ter em conta

- **Complexidade computacional**: é essencial analisar a complexidade do algoritmo (tempo de execução e memória ocupada).
- **Flexibilidade**: algoritmos muito refinados são, de um modo geral, difíceis de compreender e estender.
- **Facilidade de implementação e compreensão**: muitas vezes é vantajoso implementar um algoritmo simples, mesmo se perdermos no desempenho nas operações não-críticas.
- **Sintonia fina com o modelo de objectos**: será que vale a pena repensar na estrutura do modelo de objectos? Queremos adicionar outras classes de objectos?

Seleccionar estruturas de dados

apropriadas para cada algoritmo

- Escolher algoritmos passa por escolher as estruturas de dados sobre as quais o algoritmo vai funcionar.
  - Durante a análise encontramos uma forma lógica de representação, mas durante o desenho precisamos escolher as estruturas de dados que permitem algoritmos eficientes.
- Exemplos de estruturas de dados são:
  - Vectores, listas, filas, pilhas, sacos, árvores, tabelas de *hash*, etc.
- Alguns ambientes de desenvolvimento oferecem bibliotecas de estruturas de dados.

Definir novas classes e operações internas

- Durante a definição e refinamento dos algoritmos podem aparecer novas classes, para suportar valores intermédios.
- Durante a decomposição de operações complexas podemos novas encontrar operações mais simples.
- Estas classes e operações só se definem no desenho, já que não são visíveis externamente.

Optimizar o desenho

- Não esquecer que:
  - o modelo de análise é um modelo lógico;
  - o modelo de desenho deve pormenorizar o modelo de análise de modo a suportar uma solução;
  - um modelo de análise correcto pode não ser o mais eficiente;
  - optimizações tornam os modelos obscuros e menos reutilizáveis;
  - é preciso encontrar um equilíbrio entre eficiência e clareza.
- Optimizações possíveis:
  - Adicionar associações redundantes.
  - Guardar resultados calculados.
  - Rearranjar algoritmos para maior eficiência.

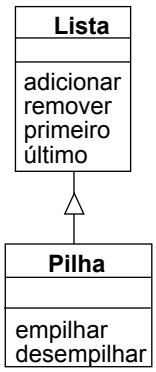
Reajustar a herança (1)

- À medida que vamos progredindo no desenho, podemos aumentar a herança.
- Devemos:
  - ajustar as definições de classes e de operações para aumentar a herança;
  - factorizar comportamento comum entre classes para criar superclasses (não esquecer que podemos ter adicionado classes do domínio da solução);
  - usar delegação para partilhar comportamento quando a herança não faz muito sentido semanticamente.

Reajustar a herança (2)

Cuidado!

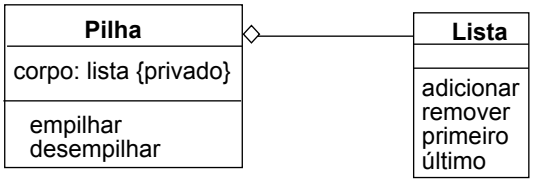
Não é muito recomendável recorrer à herança apenas para efeitos de reutilização de código!



Será boa ideia?

Reajustar a herança (3)

- Neste caso, o melhor mesmo seria usar delegação, através da agregação.



Desenhar associações (1)

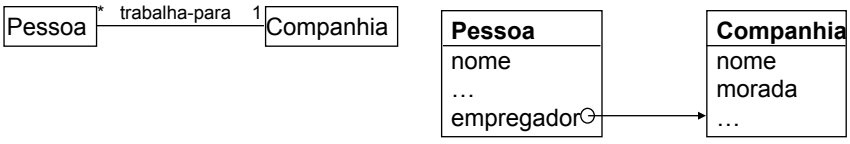
- Associações são um mecanismo conceptual útil em análise mas que terá que ser implementado no desenho.
- Podemos implementar associações de uma maneira uniforme, ou seleccionar técnicas particulares para cada uma, dependendo do que ela representa na aplicação.
- O melhor é analisar primeiro o papel que cada associação desempenha no modelo de objectos.

Desenhar associações (2)

- A maioria das associações no diagrama de classes são bidireccionais;
  - se houver algumas que são atravessadas num só sentido, a sua implementação pode ser simplificada,
  - mas não esquecer que mais tarde podemos querer mudar de ideias;
- Seja qual for a técnica escolhida, devemos esconder a sua implementação, usando operações que actualizam e atravessam a associação.

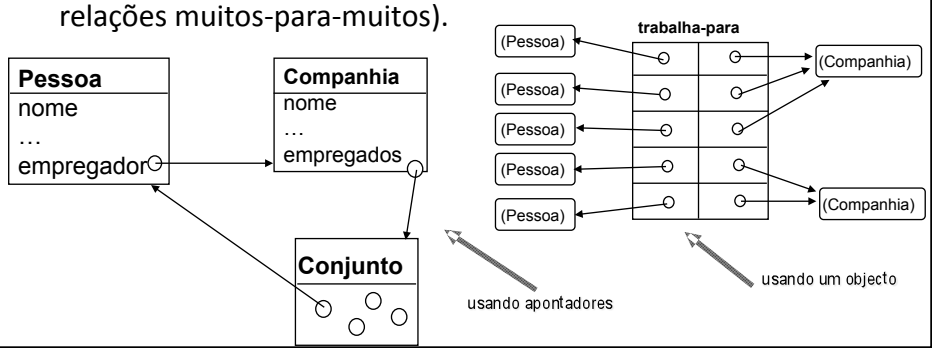
Desenhar associações unidireccionais

- Se uma associação é unidireccional, pode ser implementada da seguinte forma:
  - se a cardinalidade é "1" usar um apontador simples;
  - se a cardinalidade é "N" usar um conjunto cujos elementos são apontadores, ou ainda
  - se a cardinalidade é "N" ordenada usar uma lista;



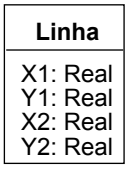
Desenhar associações bidireccionais

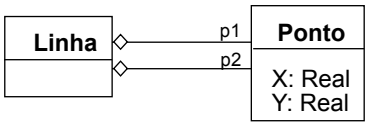
- Se uma associação é bidireccional:
  - implementá-la com um atributo num dos sentidos; implementar no sentido inverso fica caro se for para ser usada raramente; antes de o fazer verificar se é mesmo necessário;
  - implementá-la com atributos nos dois sentidos, usando as ideias apresentadas para associações unidireccionais;
  - implementá-la como um objecto de associação diferente (e.g. p/ relações muitos-para-muitos).



Determinar a representação dos objectos (tipos primitivos e bibliotecas)

- Implementar um objecto é simples, mas é preciso saber se se **usam tipos primitivos, classes já existentes na biblioteca ou combinações de objectos relacionados**.
- Classes podem ser implementadas à custa de outras classes, mas mais cedo ou mais tarde as classes componentes terão que ser classes primitivas do tipo *String*, *Integer*, etc.





Documentar as decisões de desenho

- É importante documentar todas as decisões de desenho, especialmente se:
  - o projecto for grande;
  - trabalharmos em equipa.
- Não é difícil recordarmos decisões simples num projecto pequeno, mas é difícil recordá-las num projecto grande.
- A documentação final é uma extensão da documentação de análise, com:
  - Modelo de componentes
  - Modelo de classes pormenorizado: modelo gráfico, descrição de cada classe de objectos e notação apropriada para mostrar decisões de implementação (por exemplo, para as associações);
  - Modelo dinâmico pormenorizado: modelo gráfico e pseudocódigo, especificação das operações com argumentos e resultados.