Changes Based Approach
or, Data First

Preparado para Mudanças, começando pelos dados

O que é

- Desenvolver software bem. Com alta qualidade.
- Uma forma diferente de desenvolver software
  - baseada em experiência, análise e estudo.
- Resumo:
  - Pensado para mudanças
  - Prioridade aos dados
  - Desenvolvimento por adaptação progressiva ao cliente
  - **Excelência**

Metodologias

- Confundidas com software:
  - acredita-se nelas, acredita-se que aplicar a metodologia vai necessariamente dar bons resultados
- Ou com magia
- Viram burocracias
- Acabam servindo para vender ferramentas inúteis
- O que vale são os preceitos: o manifesto de Agile, o conceito de microserviço, o espírito de desenvolver incluindo os testes

O fazedor de software é um artesão

- Preocupado com qualidade
- Tem sua maneira de trabalhar e de se organizar; seu método, que é próprio de cada um
- Tem sua forma de lidar com a equipe
- Suas preferências e habilidades

Como fazer

- Idéia: não modelar uma realidade que é fluida e difusa
- Foco nos dados
  - Organizados em documentos
  - Devem ter o máximo de qualidade: validação, compliance com normas, atualizados, documentados, identificados, com história, referências, rastreamento de mudanças
- Fluxos: é aquilo que se faz com os dados
  - Os fluxos serão implementados progressivamente a partir dos dados: "o que se faz com este documento?"
- Mudanças: o software não é pensado como uma solução completa para um problema. É pensado como uma sucessão de mudanças. Tem que ser desenvolvido pensando que vai mudar:
  - Documentar
  - Isolar funcionalidades
  - Organizar em microserviços
  - Não pensar em resolver de uma vez uma estrutura complexa. Em vez disso implementar microserviços (ou módulos) que:
    1. Farão corretamente uma só funcionalidade
    2. Tem o mínimo de complexidade
    3. Serão mudados quando necessário
    4. Sua mudança não deve afetar outros. Independência.
    5. Eventualmente poderão ser re-usados. Como módulos ou como microserviços.
    6. Ao longo do tempo vão se constituir numa rica biblioteca de recursos
  - Simplicidade

Roteiro (1)

- (Manning): **não modelar uma realidade que é mutante. Seguir a trilha codificando o software.**

1. Breve conversa: identificar papéis e entidades, visão simples do fluxo
   - Papéis vão corresponder a "cadastros": clientes, peças, produtos...
   - Fluxo vai definir um primeiro protótipo, "toy"
   - Breve conversa pode/deve ser complementada por estudo de referências do empreendimento. Isto é essencial para dimensionar o projeto e ter uma idéia de arquitetura.
2. Toy
   - Desenvolver um programa o mais simples possível que realiza um fluxo mínimo: compra de um produto, depósito numa conta bancária, admissão num hospital
   - Interface mínima, provavelmente CLI
   - Funciona, pode ser mostrado ao cliente como modelo simplístico. Certamente não tem bugs.
3. Os dados
   - Identificar os documentos que são usados no fluxo
     - e os eventos que são gerados ou geram documentos
   - Detalhar completamente o documento: data, identificação, campos, tipos de dados, validação (nome do programa, norma), forma de entrada, formato de saída(s), regras e regulamentos aplicáveis, histórico, microserviço que trata, mudanças ocorridas
     - ❖ Formatar num padrão (schema): Json, xml...
     - ❖ Desenvolver os programas necessários: entrada, validação, saída. Testes.
     - ❖ O programa deve simplesmente percorrer a definição e executar: entrada, validação, disparo do programa que trata o documento.

   Confirmar continuamente com o cliente

   Guardar as definições (schema) e os dados em arquivo de documentos (MongoDB, Redis...). Os fluxos em grafos.

Roteiro (2)

4. O design começa aqui (ou antes):
    - UI
    - Programação visual
    - Segue uma trilha paralela ao desenvolvimento do software
5. O(s) fluxo(s):
    - Tendo os documentos detalhados, com entrada, validação, saída prontos, seguir o fluxo a partir de um documento/evento inicial
    - Diálogo com o cliente identificando passo a passo o fluxo do documento
        - Procurar o caminho mais simples e mais genérico, deixar exceções para uma segunda passagem, seguir o fluxo principal
        - Implementar cada uma das etapas, com confirmação pelo cliente. Testes.
        - Repetir para cada documento
        - Criar os cadastros cuja necessidade for aparecendo durante o fluxo
    - Validar o fluxo todo com o cliente
    - Fazer uma revisão de segurança em cada etapa
    - Implementar uma a uma as exceções, validando com o cliente
        - Estes serão casos de mudanças.
        - A vida inteira do software será implementar mudanças: correções, novas funcionalidades, modificações introduzidas, regulamentações.
    - Em cada caso, avaliar criar um microserviço ou um módulo. Documentar.

6. Mudanças

Mudanças

A partir daí resta implementar mudanças continuamente

- Suporte e desenvolvimento são uma coisa só
- Todas as etapas do software são software:
  - Deploy
  - Devops, CI/CD
  - Infra as software
  - Testes e homologação
  - Revisão independente de segurança
  - Revisão de qualidade
- Com o tempo surgirão mudanças grandes. Estratégia e arquitetura mudaram ou o sistema ganhou complexidade:
  - Refactoring a partir de uma arquitetura que já foi comprovada

Qualidade

- É um espírito
- Testes exaustivos
  - e mais testes
- Revisão, avaliação, rejeição:
  - por pares
  - por chefe
  - pelo cliente
- Fundamentada nos princípios e boas práticas das normas de qualidade
  - ISO/IEC 25010
  - (ISO/IEC 9126)
  - Open Group Application Platform Service Qualities

Ferramentas

- O mínimo possível
- Com as quais o artesão está confortável
- Editor, compilador, link, git
- Bancos de dados:
  - de documentos, key-value, graph quando necessário
  - Redis, Mongodb, Neo4j

- Evitar:
  - Banco de dados relacional
  - Orientação a objetos

Deploy: a ser pensado antes

- Procurar fazê-lo total software
- Segurança
- Contingência
- *Pensar num modelo edge-computing*

Geraldo Coen

geraldo.coen@sixpartners.com.br

http://www.coeninfo.com/

https://www.linkedin.com/in/geraldo-coen-495130/

coen.geraldo@gmail.com

Changes Based Approach
or… Data First

Changes ready, starting with data

What is CBA?

- Develop software well. With high quality.
- A different way of developing software
  - based on years of experience, analysis and study
- Summary:
  - Ready for changes
  - Data is the primary focus
  - Progressive adaptation to customer needs
  - **Excellence**

Methodologies

- Methodologies are confused with software: people believe that applying the methodology will necessarily get good results
- or with magic
- They become bureaucracies.
- They end up helping to sell useless tools
- What counts are the precepts: the Agile manifesto, the microservice concept, the spirit of developing including testing

The software maker is a craftsman

- Concerned about quality
- It has its own way of working and organizing, his method, which is unique to each one
- He has his way of dealing with the team.
- His preferences and skills

CBA: how to do it

- Do not model a reality that is fluid and diffuse
- Focus on data
  - Organized by documents
  - Must have the highest quality: validation, compliance with standards, updated, documented, identified, with history, references, changes tracking
- Flows: what you do with the data
  - Flows will be implemented progressively starting with the data: "what do you do with this document"
- Changes: Software is not thought of as a complete solution to a problem. It is thought of as a succession of changes. It has to be developed thinking that it will change:
  - To document
  - Isolate features
  - Organize into microservices
  - Do not think about solving a complex structure at once. Instead implement microservices (or modules) that:
    1. Will do just one feature correctly
    2. Have the least complexity
    3. Will be changed when needed
    4. Changes must not affect others. Independence.
    5. Eventually can be reused. As modules or as microservices.
    6. Over time will build up a rich library of resources
  - Simplicity

Script (1)

- (Manning): **do not model a changing reality that is changing. Track it by coding software.**
- 1. Short talk: identify roles and entities, simple flow view
  - Roles will correspond to files: customers, parts, products...
  - Flow will define a first prototype, "toy"
  - A brief talk can/should be complemented by a study of the enterprise's references. This is essential for sizing the project and having an architectural idea.
- 2. Toy
  - Develop a program as simple as possible that performs a minimal flow: purchase of a product, deposit into a bank account, admission to a hospital
  - Minimal interface, probably CLI
  - It works, it can be shown to the customer as a simplistic model. It certainly has no bugs.
- 3. The data
  - Identify the documents that are used in the flow
  - and the events that are generated or generate documents
  - Fully detail the document: date, identification, fields, data types, validation (program name, standard), input form, output format(s), applicable rules and regulations, history, microservice that handles it, changes that have occurred
    - Format the document in a schema: Json, xml...
    - Develop the necessary programs: input, validation, output. Tests.
    - The program must simply go through the definition and execute: input, validation, triggering the program that handles the document.
  - Continuously confirm with the customer
  - Save definitions (schema) and data in a document file (MongoDB, Redis...). Flows in graph database.

Script (2)

4. Design starts here (or before):
    - UI
    - Visual programming
    - Follows a parallel path with software development
5. The flow(s):
    - Having detailed documents, with input, validation, output ready, follow the flow from an initial document/event
    - Dialogue with the customer identifying the document flow step by step
        - Look for the simplest and most generic path, leave exceptions for a second pass, follow the main flow
        - Implement each of the steps, with customer. Tests.
        - Repeat for each document
        - Create the files/records whose need appears during the flow
    - Validate the entire flow with the client
    - Do a security review at each step
    - Implement exceptions one by one, validating with the client
        - These will be cases of change.
        - The entire life of the software will be implementing changes: fixes, new features, introduced modifications, regulations.
    - In each case, consider creating a microservice or module. Document it.

6. Changes

Changes

From then on, it remains to continuously implement changes

- Support and development are one and the same
- All software steps are software:
  - Deploy
  - Devops, CI/CD
  - Infra as software
  - Tests and homologation
  - Independent security review
  - quality review
- Over time, big changes will emerge: strategy and architecture have changed or the system has gained complexity:
  - Refactoring from a proven architecture

Quality

- It's a spirit
- exhaustive tests
  - and more tests
- Review, evaluation, rejection:
  - by pairs
  - by boss
  - by the customer
- Based on the principles and good practices of quality standards
  - ISO/IEC 25010
  - (ISO/IEC 9126)
  - Open Group Application Platform Service Qualities

Tools

- The minimum
- Which the craftsman is comfortable with
- Editor, compiler, link, git
- Databases:
  - documents based, key-value, graph when needed
  - Redis, Mongodb, Neo4j

- Avoid:
  - relational database
  - Object orientation

Deploy: to be thought before

- Seek to do it 100% software
- Safety
- Contingency
- *Think about an edge-computing model*

References

- Data-Oriented Programming, Yehonathan Sharvit
  https://www.manning.com/books/data-oriented-programming
- Introduction to Interface-Driven Development (IDD)
  https://dzone.com/articles/introduction-to-interface-driven-development-idd
- A Complete Guide to Agile Methodologies and Their Operation
  https://dzone.com/articles/a-complete-guide-to-agile-methodologies-and-their
- Assessing the DCI Approach to Preserving Use Cases in Code: Qi4J and Beyond (pdf)
  DCI = Data, Context and Interaction
- We Are Testing Software Incorrectly and It's Costly
  https://dzone.com/articles/we-are-testing-software-incorrectly-and-its-costly
- A Guide to Data-Driven Design and Architecture
  https://dzone.com/articles/a-guide-to-data-driven-design-and-architecture
- Optimizing Success With Data-Driven and Custom Software Development Services
  https://dzone.com/articles/optimizing-success-with-data-driven-and-custom-sof

Reduce System Complexity with Data-Oriented Programming

Complexity is one of the main difficulties in the development of successful software systems. Modern programming languages and frameworks make it easy to develop and deploy our code quickly, but as the code base grows, complexity makes it challenging to add new features.

Data-oriented programming is a paradigm that aims at reducing the complexity of information systems such as back-end applications, web services, web workers, and front-end applications by rethinking data. Data-oriented programming treats data as an immutable value that is manipulated by general-purpose functions. Moreover, data is validated à la carte.

In this talk, we illustrate the principles of data-oriented programming in the context of a software production system. After attending this talk, you will be able to apply data-oriented programming principles in your preferred programming language and reduce the complexity of the systems you build.

Takeaways

1. Apply Data-Oriented Programming principles in your preferred programming language.
2. Apply data validation techniques without using static types.
3. Represent data with immutable data structures.
4. Manipulate data with generic functions.