PROCESSOS DE DESENVOLVIMENTO

PLANEJE-E-DOCUMENTE

SIN5005 — TÓPICOS EM ENGENHARIA DE SOFTWARE

Daniel Cordeiro

21 de agosto de 2019

Escola de Artes, Ciências e Humanidades | EACH | USP

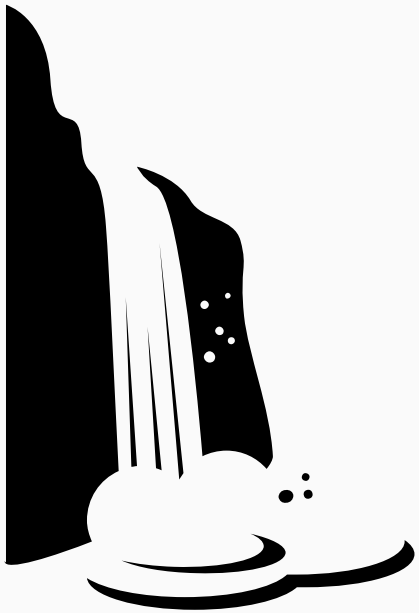
COMO EVITAR AS DESONRAS DO DESENVOLVIMENTO DE SOFTWARE?

- Será que não podemos construir software de forma que possamos predizer o cronograma, custo e qualidade da mesma forma que engenheiros constroem pontes?
- Se for possível, que tipo de processo de desenvolvimento tornaria a construção de software uma atividade mais previsível?

- A ideia é trazer a disciplina usada pelos engenheiros para a construção de software
  - Termo criado 20 anos após o primeiro computador
  - Estuda métodos de desenvolvimento tão previsíveis em qualidade, custo e tempo como a engenharia civil
- “Planeje-e-Documente”
  - Antes de escrever qualquer linha de código, o gerente do projeto estabelece um plano
  - Escreve uma descrição detalhada de todas as fases do plano
  - Progresso do projeto é medido sempre em comparação ao plano original
  - Mudanças no projeto devem ser repassadas à documentação e, possivelmente, também ao plano original

1º processo de desenvolvimento: cascata (1970)

Processo de desenvolvimento com um “ciclo de vida” em Cascata (*Waterfall*) composto de 5 fases:



Created by Robert Bjurshagen from the Noun Project

1. Análise de requisitos & especificação
2. Projeto de arquitetura
3. Implementação & Integração
4. Verificação
5. Operação & Manutenção

Ideia: completar cada fase antes de iniciar a próxima

- Por quê? Quanto antes encontrarmos um bug, mais barato será
- Uso extensivo de documentos/fases para novos desenvolvedores

QUÃO BOM É O MODELO EM CASCATA?

E os usuários exclamaram com uma risada e uma provocação: "isso é exatamente o que nós pedimos, mas não o que nós queríamos. — Anônimo"

- Normalmente quando um consumidor vê o produto pronto, ele quer mudar o produto

QUÃO BOM É O MODELO EM CASCATA?

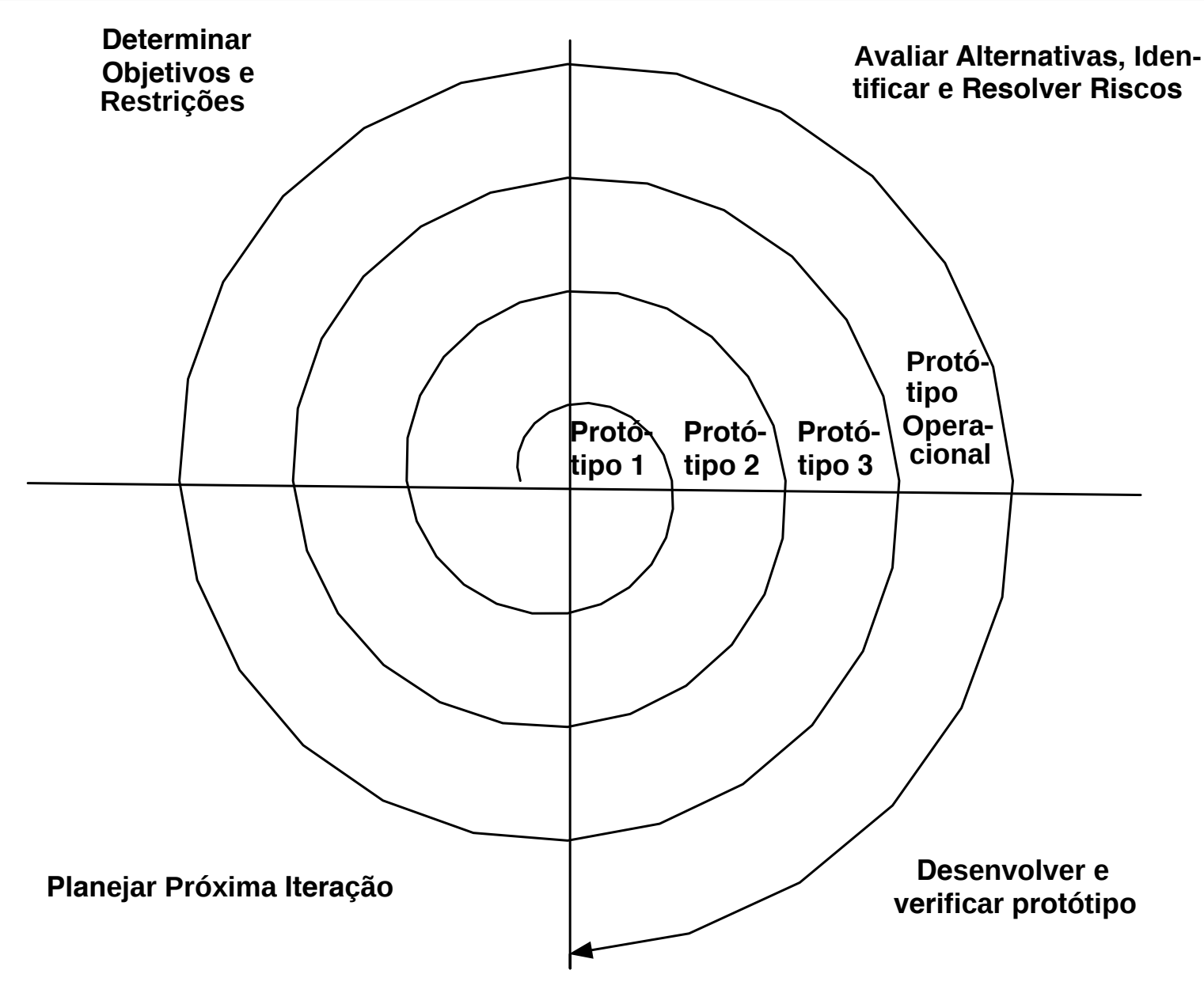
- “Se planeje para jogar fora uma [implementação]; você vai acabar tendo que fazer isso de qualquer jeito” – Fred Brooks, Jr. (vencedor do prêmio Turing em 1999)
- Normalmente, após terminar a primeira implementação, os desenvolvedores aprendem qual o jeito certo de resolver esse problema que deveria ter sido usado desde o início

Ciclo de vida Espiral (1986)

Ciclo de vida Espiral (*Spiral Lifecycle*)

- Combine o Planeje-e-Documente com protótipos
- Ao invés começar com o planejamento e documentação de todos os requisitos desde o início, planeje e documente os requisitos e desenvolva o protótipo do sistema, mas em várias iterações.

CICLO DE VIDA ESPIRAL



ESPIRAL: PRÓS & CONTRAS

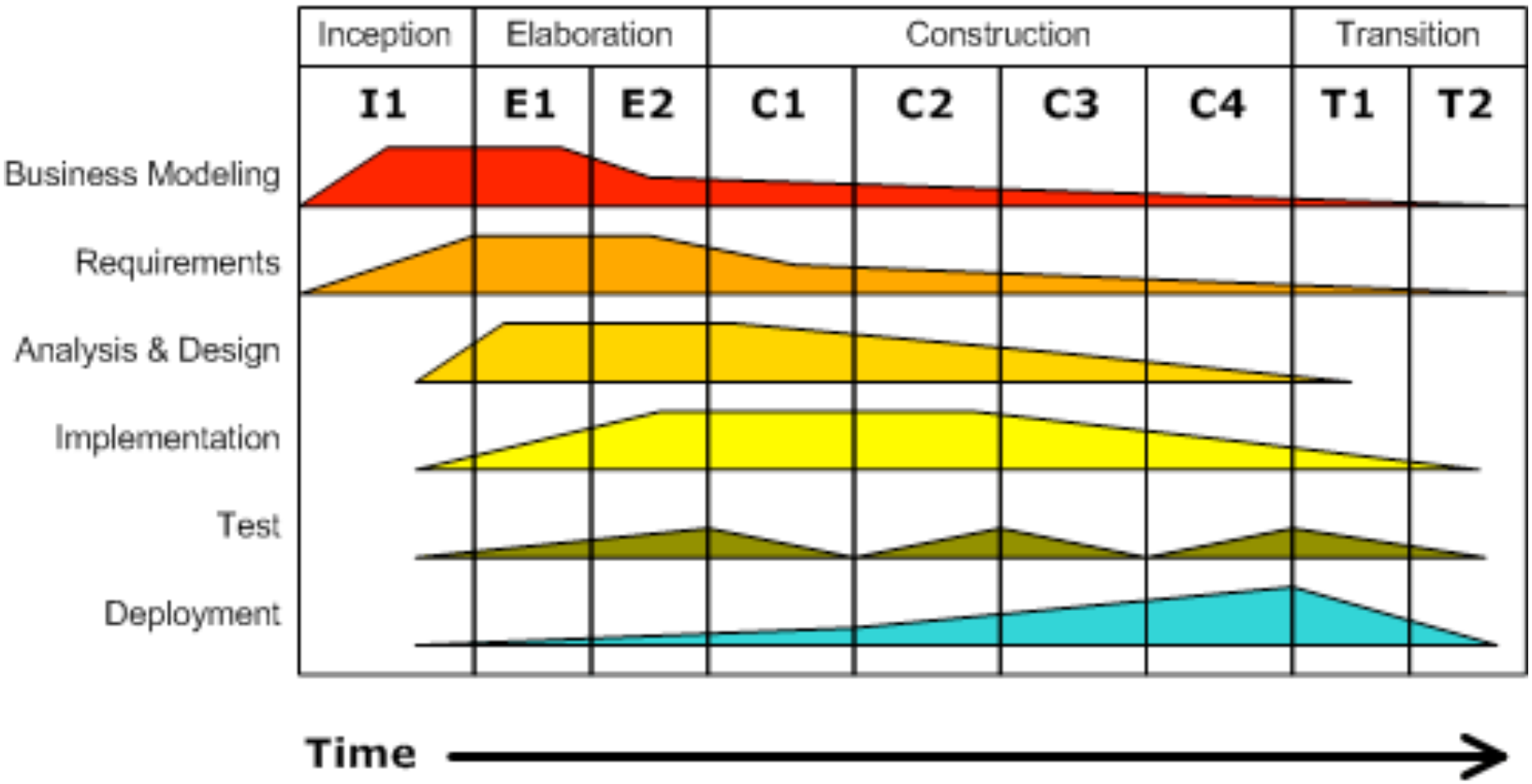
Prós

- Realizar iterações envolve o cliente no processo antes do produto ser terminado
  - reduz as chances de mal entendidos
- O gerenciamento de risco é parte do ciclo de vida
- Monitorar o projeto é mais fácil
- Prazos e custos ficam mais realísticos ao longo do tempo

Contras

- Iterações de 6 a 24 meses
  - dá tempo do cliente mudar de ideia
- **Muita** documentação por iteração
- Várias regras a serem seguidas, difícil seguir todas em todos os aspectos do projeto
- Custo do processo alto
- Torna difícil manter o orçamento e prazo dentro do combinado

Rational Unified Process (RUP, 2003)



FASES DO RUP

RPU possui 4 fases (que podem ser iteradas)

1. **Iniciação**: define o negócio do software, define cronograma e avaliação inicial de riscos
2. **Elaboração:** casos de uso, arquitetura do software, protótipo
3. **Construção:** codifica e testa o produto, 1ª versão
4. **Transição:** desloca o produto do desenvolvimento para o ambiente real de produção, inclui a aceitação do cliente

DISCIPLINAS DE ENGENHARIA DO RUP

Há 6 disciplinas (ou fluxos de trabalho) que as pessoas que trabalham no projeto devem cobrir?

1. Modelagem de negócios
2. Requisitos
3. Análise e Projeto
4. Implementação
5. Teste
6. Implantação (*deployment*)

RUP: prós & contras

Prós

- Práticas de negócio amarradas ao processo de desenvolvimento
- Várias ferramentas desenvolvidas pela Rational (agora IBM)
- Ferramentas ajudam na melhoria gradual do projeto

Contras

- As ferramentas custam caro (não são de código aberto)
- Muitas opções para ajustar RUP a cada empresa, impede o uso só das ferramentas
- Funciona bem apenas para projetos grandes ou de tamanho médio
- O tamanho de cada iteração é decisão do gerente

PLANEJE-E-DOCUMENTE: GERENTE DE PROJETO

P-e-D depende de **Gerentes de Projeto**:

- Escrevem contratos para ganhar os projetos
- Recrutam o time de desenvolvimento
- Avalia o desempenho dos desenvolvedores de software, o que define os salários
- Estima custos, mantém o cronograma, avalia riscos e os supera
- Documenta o plano de planejamento do projeto
- Recebe crédito pelo sucesso ou é o culpado se o projeto atrasa ou estoura o orçamento

P-E-D TAMANHO DO PROJETO

> *“Adicionar mais pessoas para ajudar em um projeto que está atrasado só faz o projeto atrasar mais.” — Fred Brooks Jr.*, The Mythical Man-Month

- Um novo desenvolvedor demora algum tempo para aprender sobre o projeto
- O tempo gasto com comunicação cresce com o tamanho, deixando menos tempo para o trabalho de desenvolvimento
- Grupos de 4 a 9 pessoas, mas organizadas de forma hierárquica para compor o projeto

PERGUNTA:

Qual afirmação abaixo é FALSA sobre o ciclo de vida dos processos em Cascata, Espiral e Rational Unified Process?

- Todos dependem de um planejamento meticuloso e medem o progresso do projeto de acordo com o plano inicial
- Todos dependem de uma documentação completa e minuciosa
- Todos dependem de um gerente que será responsável pelo projeto
- Todos usam iterações e protótipos

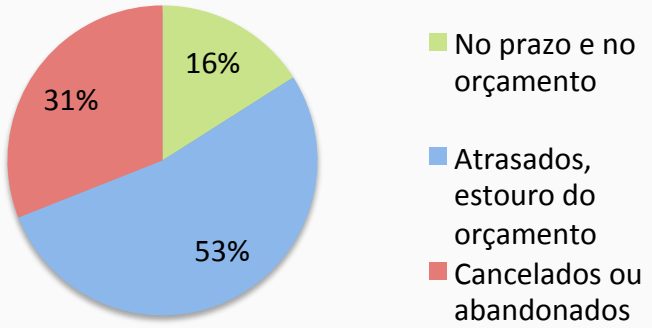
HÁ OUTROS PROCESSOS ALTERNATIVOS?

- Quão bem os processos Planeje-e-Documente atingem os objetivos de custo, cronograma e qualidade?
- P-e-D requerem extensa documentação e planejamento que dependem de um gerente com muita experiência
  - Podemos construir software de forma eficaz sem um planejamento cuidadoso e sem documentação?
  - Como evitar que todos “só saiam programando”?

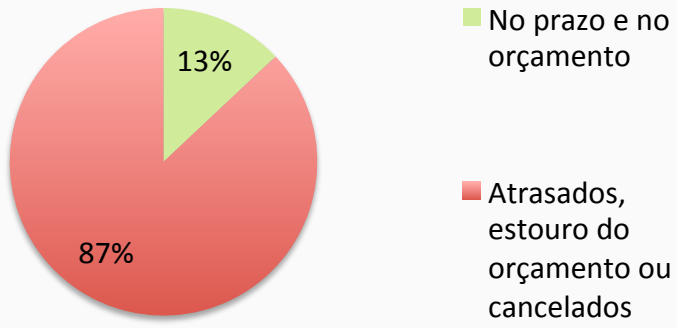
QUÃO BEM OS PROCESSOS PLANEJE-E-DOCUMENTE FUNCIONAM?

- IEEE Spectrum “Software Wall of Shame”
- 31 projetos (4 citados anteriormente + 27 outros) perderam US$ 17 bilhões

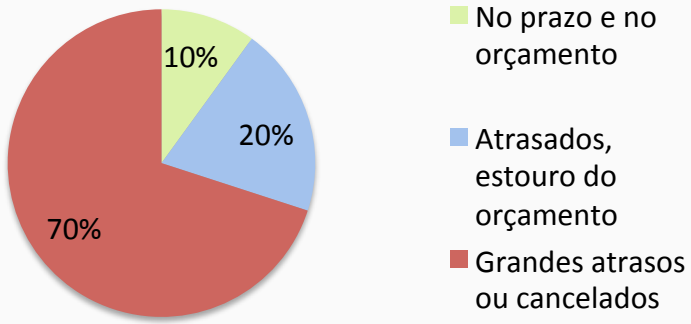
a) Projetos de Software (Johnson 1995)



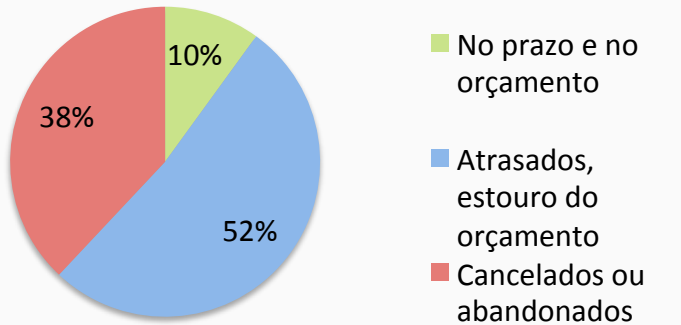
b) Projetos de Software (Taylor 2000)



c) Projetos de Software (Jones 2004)



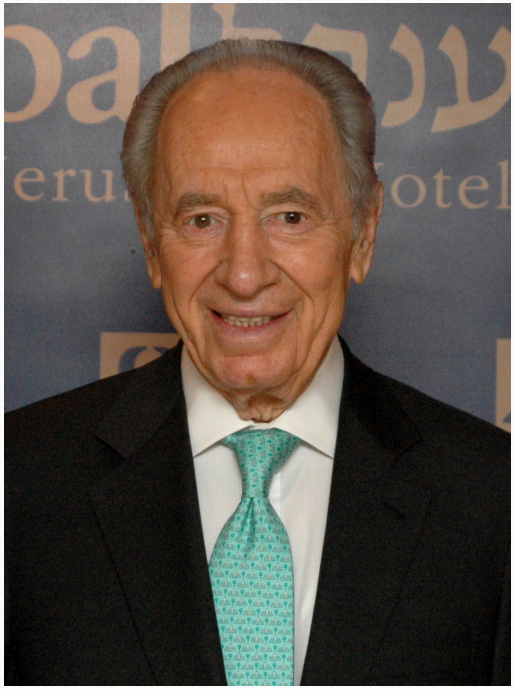
d) Projetos de Software (Johnson 2013)



LEI DE PERES

“Se um problema não tem solução, ele pode não ser um problema, mas sim um fato – não a ser resolvido, mas sim para lidarmos com ele ao longo do tempo.”

— Shimon Peres (vencedor do Prêmio Nobel da Paz em 1994)



MANIFESTO ÁGIL, 2001

"Estamos descobrindo maneiras melhores para se desenvolver software e de ajudar as pessoas a fazê-lo. Ao longo desse trabalho viemos a valorizar mais:

- **Indivíduos e interações** *ao invés de processos e ferramentas*
- **Software que funciona** *ao invés de documentação extensa*
- **Colaboração com o cliente** *ao invés de negociação de contratos*
- **Resposta a mudanças** *ao invés de seguir um plano*

Isto é, enquanto há valor nos itens da direita, nós damos mais valor os itens da esquerda."

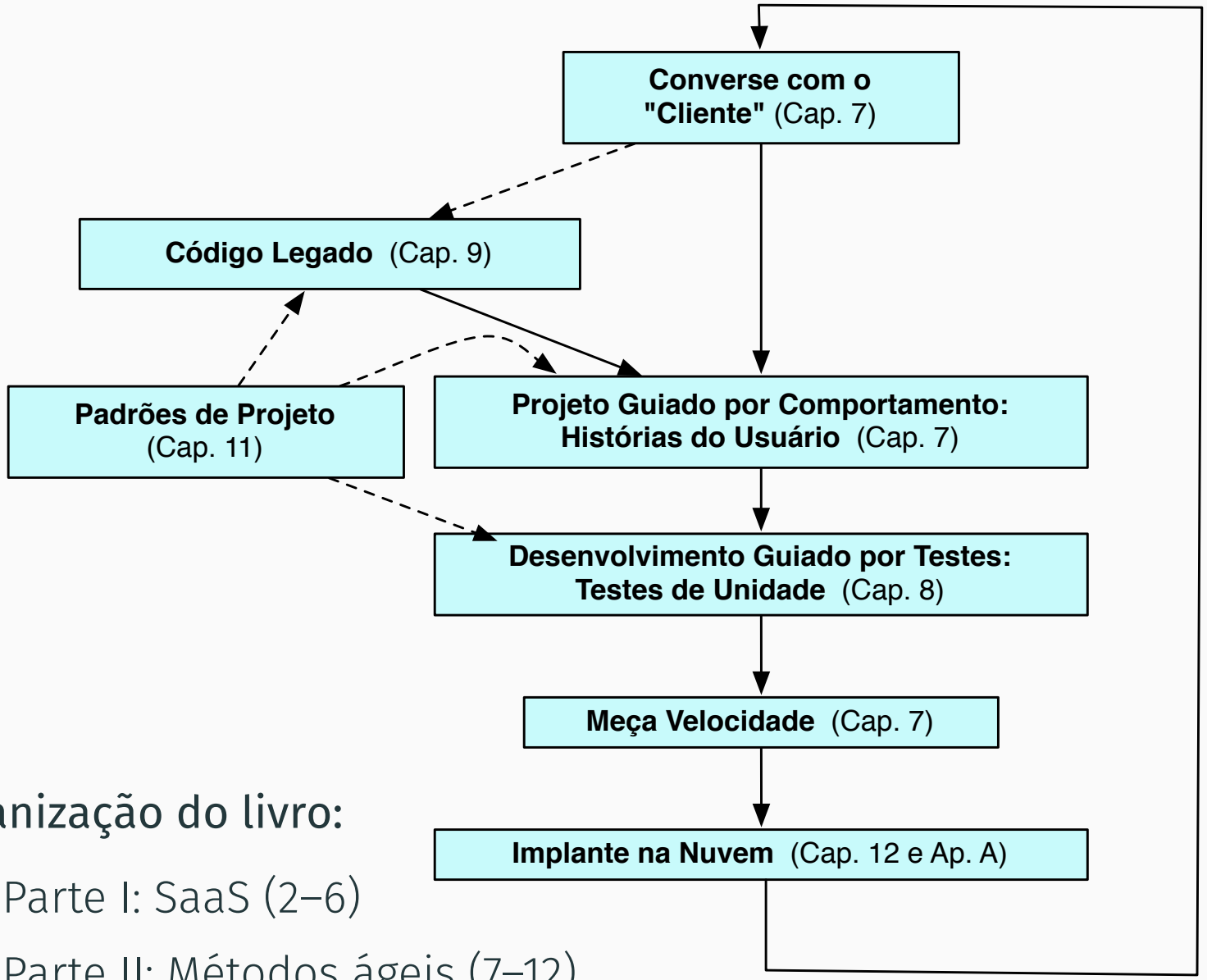
Programação extrema (XP), uma versão de método ágil

- Se iterações curtas são boas, faça que sejam as menores possíveis (semanas vs. anos)
- Se simplicidade é bom, sempre faça a coisa mais simples possível que possa fazer o trabalho
- Se testar é bom, teste o tempo todo. Escreva o código de teste antes mesmo de escrever o código que será testado
- Se revisões de código são boas, reveja código continuamente programando em pares, revezando quem ficará de olho no código do outro

CICLO DE VIDA ÁGIL

- Abrace mudanças como sendo um fato da vida: melhoria contínua vs. fases
- Desenvolvedores refinam continuamente um protótipo incompleto, mas funcional, até que o cliente esteja feliz com o resultado. *Feedback* do cliente em todas as iterações (cada 1–2 semanas)
- Métodos ágeis enfatizam que *Test-Driven Development* (TDD) diminuem os defeitos, *User Stories* validam os requisitos dos clientes e *Velocity* mede o progresso

Iterações de métodos ágeis / organização do livro-texto



Organização do livro:

- Parte I: SaaS (2–6)
- Parte II: Métodos ágeis (7–12)

- Controverso em 2001

  O Manifesto Ágil é uma tentativa a mais de minar a disciplina de engenharia de software. (...) Na profissão da engenharia de software existem engenheiros e hackers. (...) Me parece que isso não é nada mais do que uma tentativa de legitimar o comportamento do hacker. (...) A profissão de engenharia de software apenas mudará para melhor quando os clientes se recusarem a pagar por um software que não cumpre o esperado. (...) Mudar de uma cultura que encoraje a mentalidade do hacker para uma que seja baseada em práticas de engenharia de software previsíveis apenas ajudará a transformar a engenharia de software em uma disciplina de engenharia respeitada.

  — *Steven Ratkin, "Manifesto Elicits Cynicism",* IEEE Computer, *2001*

MÉTODOS ÁGEIS: ANTES E AGORA

- Controverso em 2001
- Aceito em 2013
  - um estudo de 2012 com 66 projetos verificou que a maioria usava métodos Ágeis, mesmo com times distribuídos

SIM: PLANEJE-E-DOCUMENTE
NÃO: MÉTODOS ÁGEIS

1. É necessário criar uma especificação?
2. Os clientes estarão indisponíveis durante o desenvolvimento?
3. O sistema a ser construído é muito grande?
4. O sistema a ser construído é muito complexo (ex: sistema de tempo real)?
5. O sistema terá uma vida útil muito longa?
6. Você usa ferramentas de softwares pobres?
7. O time está geograficamente distribuído?
8. A cultura do time (seu modo de pensar e planejar) é orientado à documentação?
9. O time possui habilidades de programação fracas?
10. O sistema a ser construído está sujeito a regulações e normas?

Qual dessas afirmações é verdadeira?

- Uma grande diferença entre métodos Ágeis e P-e-D é que Ágil não usa requisitos
- Uma grande diferença entre métodos Ágeis e P-e-D é medir o progresso em relação a um plano
- Você pode construir SaaS usando métodos Ágeis, mas não usando P-e-D
- Uma grande diferença entre métodos Ágeis e P-e-D é a construção de protótipos e as interações com os clientes durante o processo de desenvolvimento

Falácias e Armadilhas

FALÁCIAS E ARMADILHAS

- Falácia: o ciclo de vida Ágil é o melhor para o desenvolvimento de software
  - Ágil é uma boa escolha para alguns tipos de software, especialmente SaaS
  - Mas não é boa escolha para a NASA; código sujeito a regulamentações
- Em cada tópico do livro veremos como aplicar métodos Ágeis na prática, mas também veremos como seria a perspectiva de Planeje-e-Documente
  - obs: você irá se deparar com novas metodologias de desenvolvimento ao longo da sua carreira, então prepare-se para aprender outras no futuro

FALÁCIAS E ARMADILHAS

- Armadilha: ignorar o custo do *design* de software
  - como o custo de fabricação de um software é ≈ 0, alguns podem acreditar que também não existe custo para modificá-lo do jeito que um cliente quiser
  - ignora o custo de *design* e teste
- Será que não ter custo de fabricação de software/dados não é a mesma ideia que justifica cópias piratas de software/dados? Ninguém deveria pagar pelo desenvolvimento, mas só pela fabricação?

RESUMO: ENGENHARIA DE SOFTWARE É MAIS DO QUE PROGRAMAR

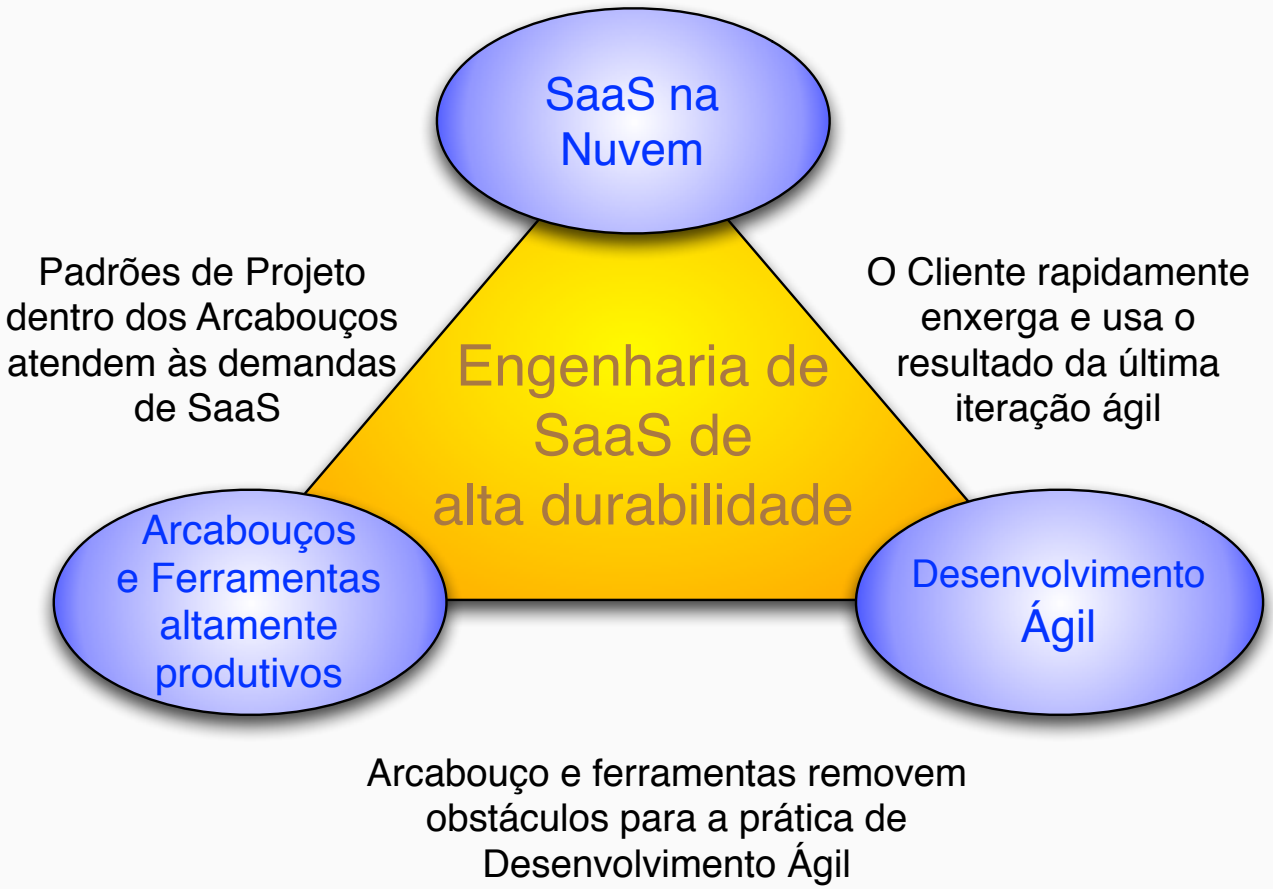


Figura 1: O triângulo da virtude da engenharia SaaS é formado a partir de três jóias da coroa da engenharia de software, (1) SaaS na Nuvem, (2) Desenvolvimento Ágil e (3) Arcabouços e ferramentas de alta produtividade.

SOBRE O PROJETO

- Times de 4–5 alunos (aparentemente, 2×4 alunos)
- Ideia: projetar, planejar, desenvolver, testar e implantar um software como um serviço
- Recomendação: usar Rails para o *backend* e HTML5+JavaScript para o *frontend*; outras linguagens/plataformas/arcabouços podem ser usados, porém você **deve** ter:
  - um arcabouço para os testes de Unidade & funcionais (recomendação: RSpec para Ruby/Rails e Jasmine para JavaScript)
  - uma forma de medir a cobertura dos testes (Coveralls e Travis CI)
  - um arcabouço de teste completo que possa expressar testes que corresponderão às histórias dos usuários (Cucumber e Capybara)
  - um arcabouço para a medição da qualidade do código, que ajude a identificar mal cheiros de projetos, problemas no estilo do código, etc. (CodeClimate)
  - arcabouço de SaaS para o lado do servidor (Rails)
  - arcabouço de SaaS para o lado do cliente (HTML5+JavaScript)

ITERAÇÃO 0 I

A iteração 0 compreende (sugiro que a ordem dos itens seja seguida):

1. Criar um projeto no GitHub
2. Adicionar *todos os integrantes* do grupo como colaboradores
3. Todos os membros devem conseguir fazer um clone do repositório e executar o projeto em seu ambiente de desenvolvimento local (recomendação: use Linux/MacOS ou Cloud9)
4. `rake spec` e `rake features` deve rodar sem erros (mesmo que ainda não haja testes escritos)
5. Integrar o repositório com Travis CI, o que significa ter o Travis CI reportando `Build: passing`
6. Integrar o repositório com o CodeClimate

ITERAÇÃO 0 II

7. Um `README.md` no repositório do projeto que inclua todos os itens abaixo:
   - "CodeClimate badge" mostrando o GPA do projeto
   - "Travis CI badge" mostrando o status do *build* no *branch master* (deve ser: `passing`)
   - Link para a app implantada no Heroku
   - Integrantes do projeto e uma pequena descrição de qual problema o software de vocês resolverá

Prazo:
Quarta, 28 de agosto de 2019 às 23h55