Pós-Graduação em Computação Distribuída e Ubíqua

INF612 - Aspectos Avançados em Engenharia de Software
Gerência de Qualidade e Integração Contínua

[Software Engineering. Sommerville. 9ª Edição. Capítulos 3, 24 e 25]
[Continuous Integration. Duvall. Capítulos 1, 2 e 3]

Sandro S. Andrade
sandroandrade@ifba.edu.br

Pós-Graduação em Computação Distribuída e Ubíqua

INF612 - Aspectos Avançados em Engenharia de Software
Gerência de Qualidade - Fundamentos

[Software Engineering. Sommerville. 9ª Edição. Capítulos 3, 24 e 25]
[Continuous Integration. Duvall. Capítulos X Y Z]

Sandro S. Andrade
sandroandrade@ifba.edu.br

Objetivos

- Apresentar as motivações para estudo e uso prático da Gerência de Qualidade
- Apresentar as relações entre Gerência de Qualidade e Processo de Desenvolvimento de Software
- Apresentar como utilizar inspeções, revisões, medições e integração contínua durante o processo de gerência de qualidade

Gerência de Qualidade

- Principais facetas:
  - Organizacional:
    - Estabelecimento de um *framework* de processos organizacionais e padronizações que conduzem a *software* de alta qualidade
  - De projeto:
    - Envolve a aplicação de processos de qualidade específicos, garantindo que os artefatos produzidos estão em conformidade com as padronizações utilizadas no projeto
    - Envolve o estabelecimento de um plano de qualidade, indicando as metas de qualidade para o projeto e definindo quais processos e padronizações serão utilizados

Gerência de Qualidade

Quality Assurance (QA): definição dos processos e padronizações que devem conduzir a produtos de alta qualidade e introdução de processos de qualidade na manufatura

Quality Control: aplicação dos processos de qualidade com o objetivo de remover aqueles produtos que não possuem o nível desejado de qualidade

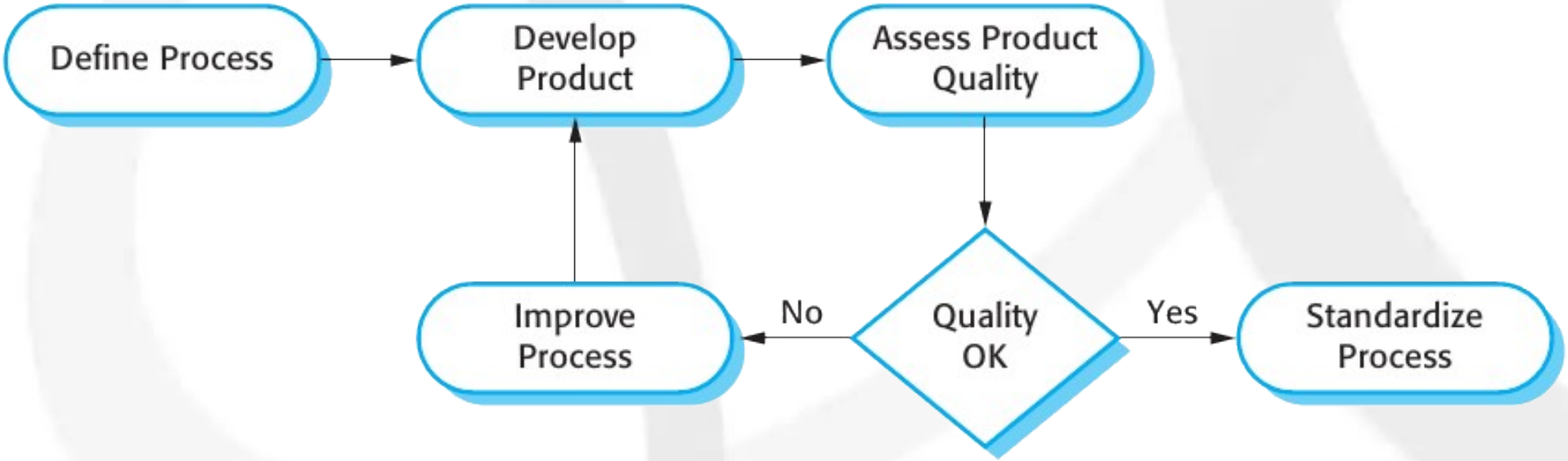
Gerência de Qualidade

- Atributos de Qualidade:

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

Gerência de Qualidade

- Qualidade suportada pelo processo:



Gerência de Qualidade

- Importância de adoção de padronizações:
  - São baseadas em conhecimentos sobre as melhores ou mais apropriadas práticas da organização. Geralmente este conhecimento é construído através de tentativa-e-erro. Documentá-lo facilita o reuso e evita repetição de erros
  - Disponibilizam um *framework* para definir o que "qualidade" significa em uma situação particular. Estabele um critério para decidir se determinado nível de qualidade foi alcançado
  - Garante uma continuidade quando o trabalho de uma pessoa é continuado por outra. Todos os engenheiros adotam a mesma prática

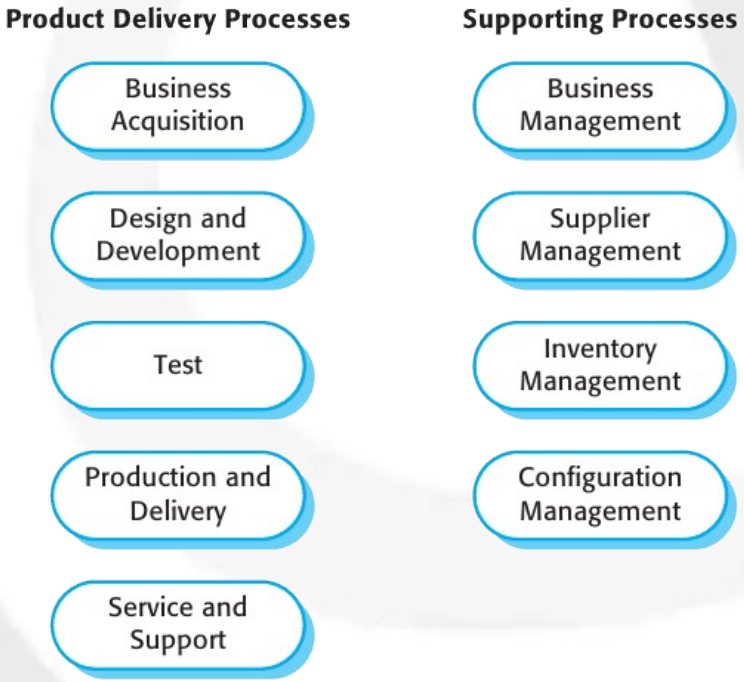
Gerência de Qualidade

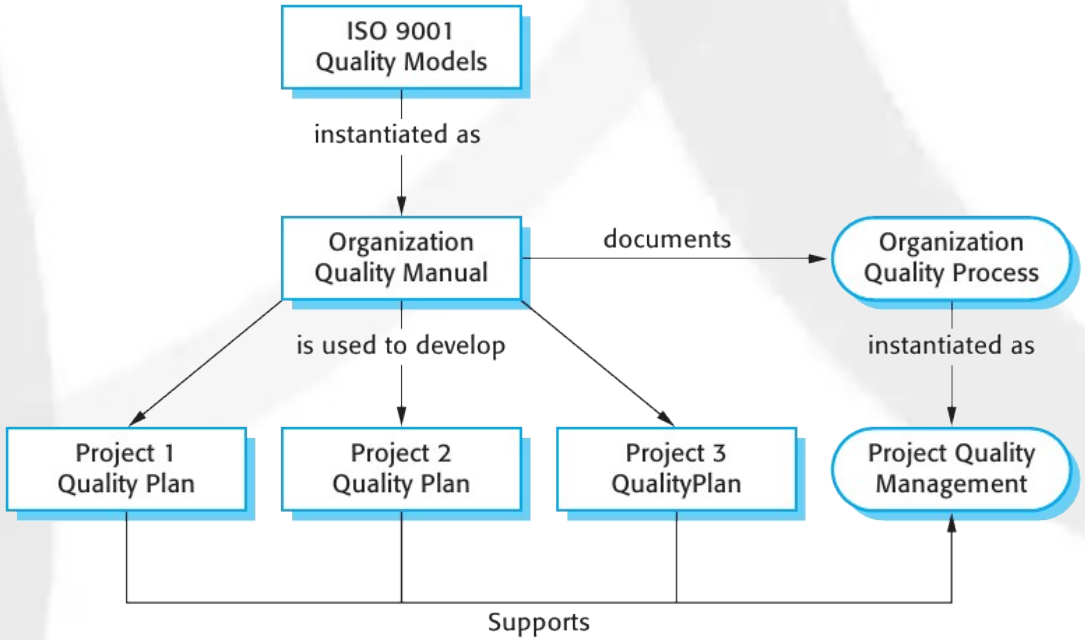
- Padronizações de processo e de produto:

Product standards	Process standards
Design review form	Design review conduct
Requirements document structure	Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Gerência de Qualidade

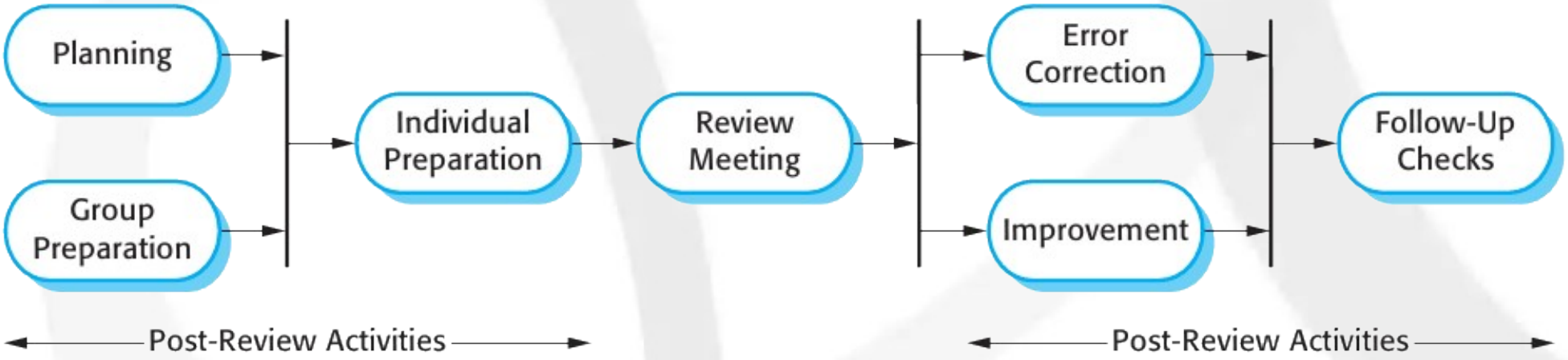
- Framework de padronizações ISO 9001
  - *Framework* para desenvolvimento de padronizações de *software*





Gerência de Qualidade

- Processo de revisão de software



Gerência de Qualidade

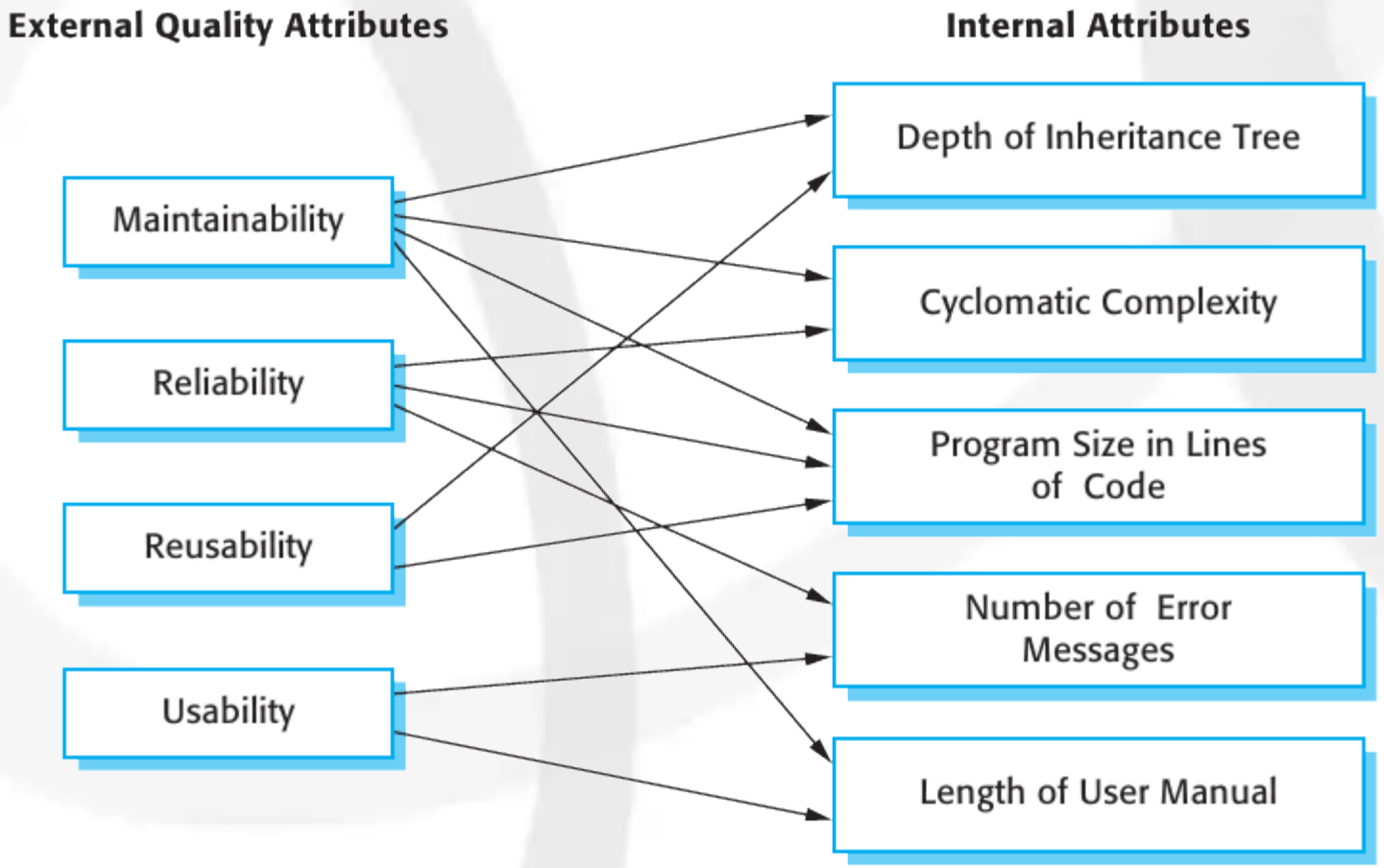
- Inspeção x Revisão

Inspeção (*Inspection*): *peer review* de qualquer trabalho produzido pelos indivíduos do grupo, objetivando a descoberta de erros através da aplicação de um processo bem definido (*Fagan Inspection*). Verifica se as padronizações e diretrizes estão sendo seguidas

Revisão (*Code Review*): tipo especial de inspeção onde uma equipe examina um código-fonte e corrige os defeitos nele encontrados. Um defeito pode ser: um requisito inapropriadamente implementado, uma funcionalidade que não funciona ou uma funcionalidade que pode ser melhorada. Importantes para realizar treinamento cruzado dos programadores e ajudar os menos experientes com boas práticas de desenvolvimento

Gerência de Qualidade

- Atributos de qualidade internos e externos:



Gerência de Qualidade

- Algumas métricas de produto:

Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.

Gerência de Qualidade

- Algumas métricas de produto:

Length of identifiers	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.

Gerência de Qualidade

- Suite CK de métricas OO

Object-oriented metric	Description
Weighted methods per class (WMC)	This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so cannot be reused effectively as superclasses in an inheritance tree.
Depth of inheritance tree (DIT)	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.

Gerência de Qualidade

- Suite CK de métricas OO

Coupling between object classes (CBO)	Classes are coupled when methods in one class use methods or instance variables defined in a different class. CBO is a measure of how much coupling exists. A high value for CBO means that classes are highly dependent, and therefore it is more likely that changing one class will affect other classes in the program.
Response for a class (RFC)	RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class. Again, RFC is related to complexity. The higher the value for RFC, the more complex a class and hence the more likely it is that it will include errors.
Lack of cohesion in methods (LCOM)	LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes. The value of this metric has been widely debated and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics.

Pós-Graduação em Computação Distribuída e Ubíqua

INF612 - Aspectos Avançados em Engenharia de Software
Gerência de Qualidade e Desenvolvimento Ágil

[Software Engineering. Sommerville. 9ª Edição. Capítulos 3, 24 e 25]
[Continuous Integration. Duvall. Capítulos X Y Z]

Sandro S. Andrade
sandroandrade@ifba.edu.br

Gerência de Qualidade

- Características comuns dos métodos ágeis de desenvolvimento:
  - Os processos de especificação, projeto e implementação acontecem de forma entrelaçada (*interleaved*)
    - Não há uma especificação detalhada do sistema. Documentação de projeto é minima ou gerada automaticamente
  - O sistema é desenvolvido em uma série de versões. Usuários finais e outros *stakeholders* especificam e avaliam cada versão
  - GUIs são desenvolvidas de forma rápida e facilmente transformadas para tecnologias de apresentação específicas

Gerência de Qualidade

- Manifesto Ágil:
  - "Estamos descobrindo formas melhores de desenvolver *software*, praticando e ajudando outros desenvolvedores. Através deste trabalho passamos a dar valor a:
    - Indivíduos em vez de interações através de processos e ferramentas
    - *Software* que funciona em vez de extensas documentações
    - Colaboração direta do cliente em vez de negociação de contrato
    - Resposta rápida a mudanças em vez de seguir um plano pré-definido
  - Ao mesmo tempo em que os items da direita são importantes, nós decidimos valorizar muito mais os itens da esquerda"

Gerência de Qualidade

- Exemplos de Métodos Ágeis:
  - eXtreme Programming (Beck, 1999-2000)
  - Scrum (Cohn, Schwaber, Beedle, 2001-2009)
  - Crystal (Cockburn, 2001-2004)
  - Adaptive Software Development (Highsmith, 2000)
  - DSDM (Stapleton, 1997-2003)
  - Feature Driven Development (Palmer, Felsing, 2002)
  - Instanciações ágeis do RUP (*Rational Unified Process*)

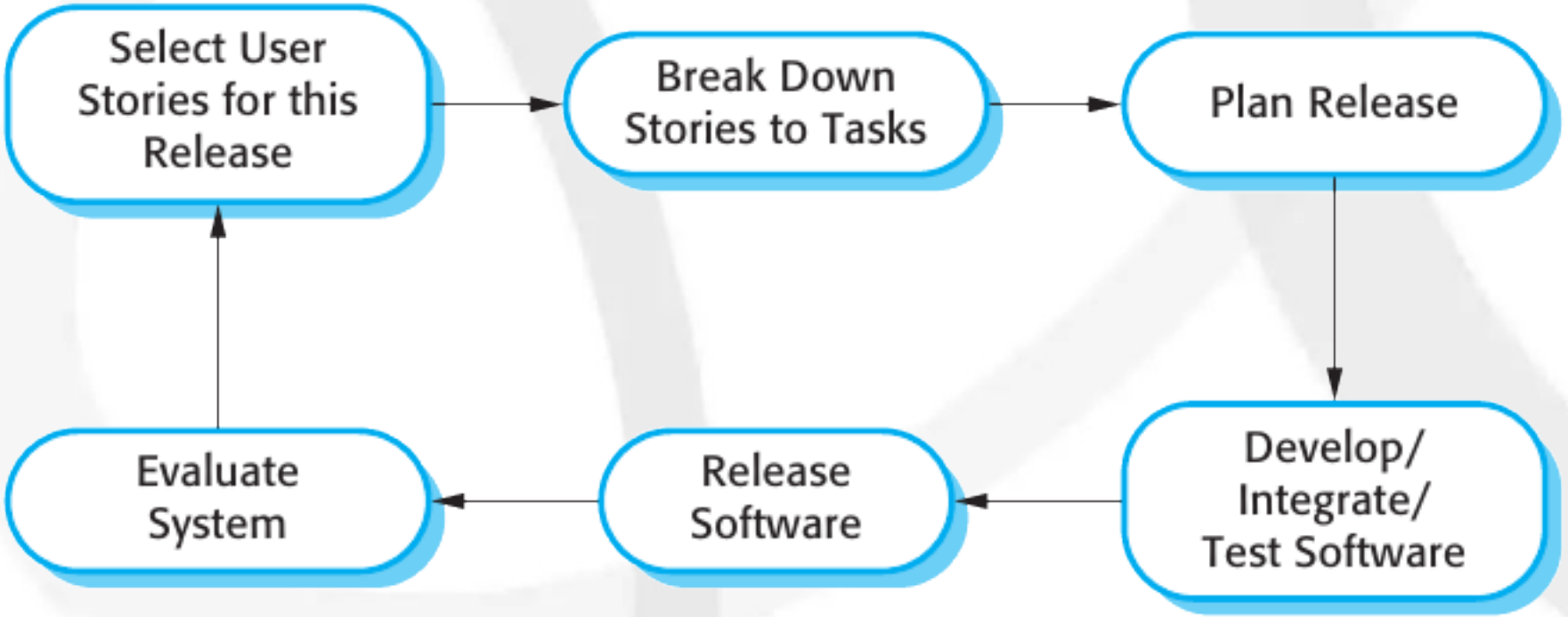
Gerência de Qualidade

- Princípios dos métodos ágeis:

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

Gerência de Qualidade

- Ciclo de *release* do eXtreme Programming:



Gerência de Qualidade

- Práticas do eXtreme Programming:

Principle or practice	Description
Incremental planning	Requirements are recorded on Story Cards and the Stories to be included in a release are determined by the time available and their relative priority. The developers break these Stories into development 'Tasks'. See Figures 3.5 and 3.6.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

Gerência de Qualidade

- Práticas do eXtreme Programming:

Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

Gerência de Qualidade

- Programação em pares – vantagens:
  - Apoia a ideia de propriedade e responsabilidade coletivos para o sistema (*egoless programming*)
  - Funciona como um processo informal de *review*. É menos formal que uma inspeção porém é mais barato
  - Ajuda a suportar *refactoring*. Evita que o trabalho de uma única pessoa (que faz o *refactoring*) seja visto como um produtor de resultados a longo prazo

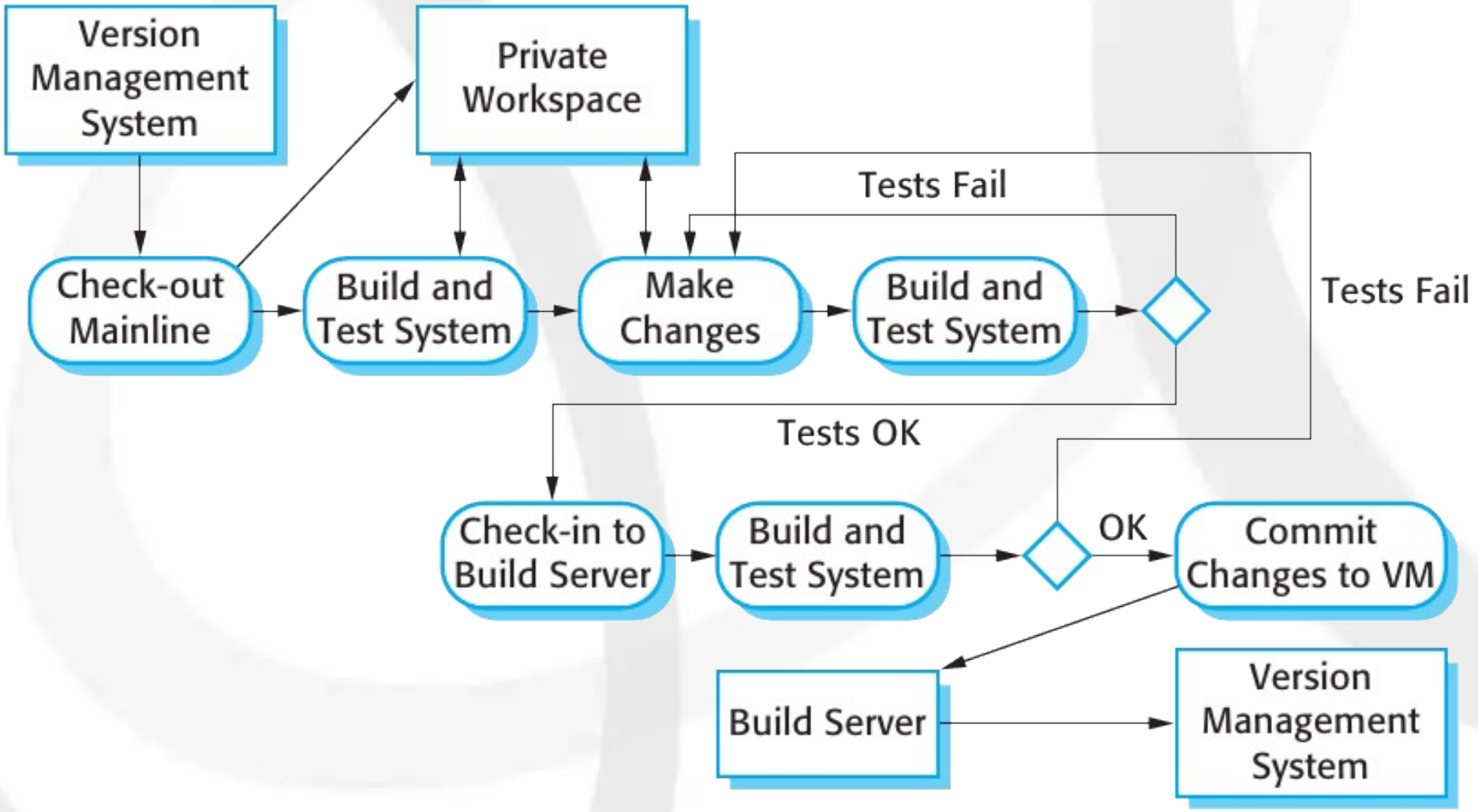
Pós-Graduação em Computação Distribuída e Ubíqua

INF612 - Aspectos Avançados em Engenharia de Software
Integração Contínua

[Software Engineering. Sommerville. 9ª Edição. Capítulos 3, 24 e 25]
[Continuous Integration. Duvall. Capítulos X Y Z]

Sandro S. Andrade
sandroandrade@ifba.edu.br

Integração Contínua



Integração Contínua

- Integração Contínua:

Um ***build*** é muito mais que uma simples compilação (ou interpretação, em linguagens dinâmicas). Pode envolver a compilação, teste, inspeção e implantação, dentre outras coisas. Um *build* é o processo de integração de código-fonte e verificação que o *software* funciona corretamente, como uma unidade

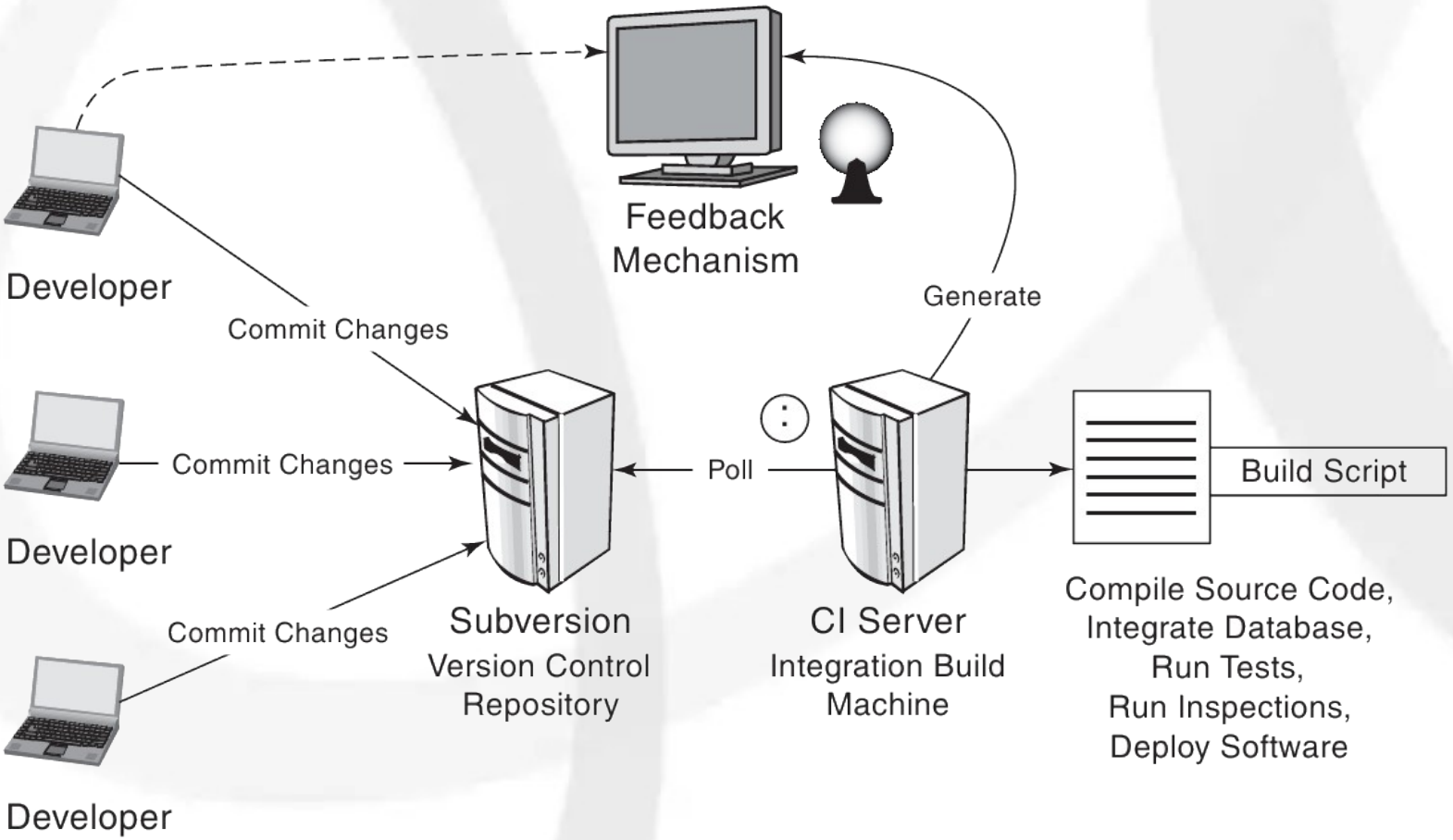
Integração Contínua

- Passos típicos em um cenário de Integração Contínua (IC):

1) O desenvolvedor realiza o *commit* das modificações que ele realizou no código-fonte. Initerruptamente, servidor de IC realiza *pooling* no repositório, buscando por mudanças
2) Logo após a realização do *commit*, o servidor de IC detecta a mudança, recupera a versão mais recente do código e execute um *build script*, para integração do *software*
3) O servidor de IC gera relatórios contendo os resultados do *build* e os envia para membros do projeto
4) O servidor de IC continua fazendo *pooling* no repositório

Integração Contínua

- Passos típicos em um cenário de Integração Contínua (IC):



Integração Contínua

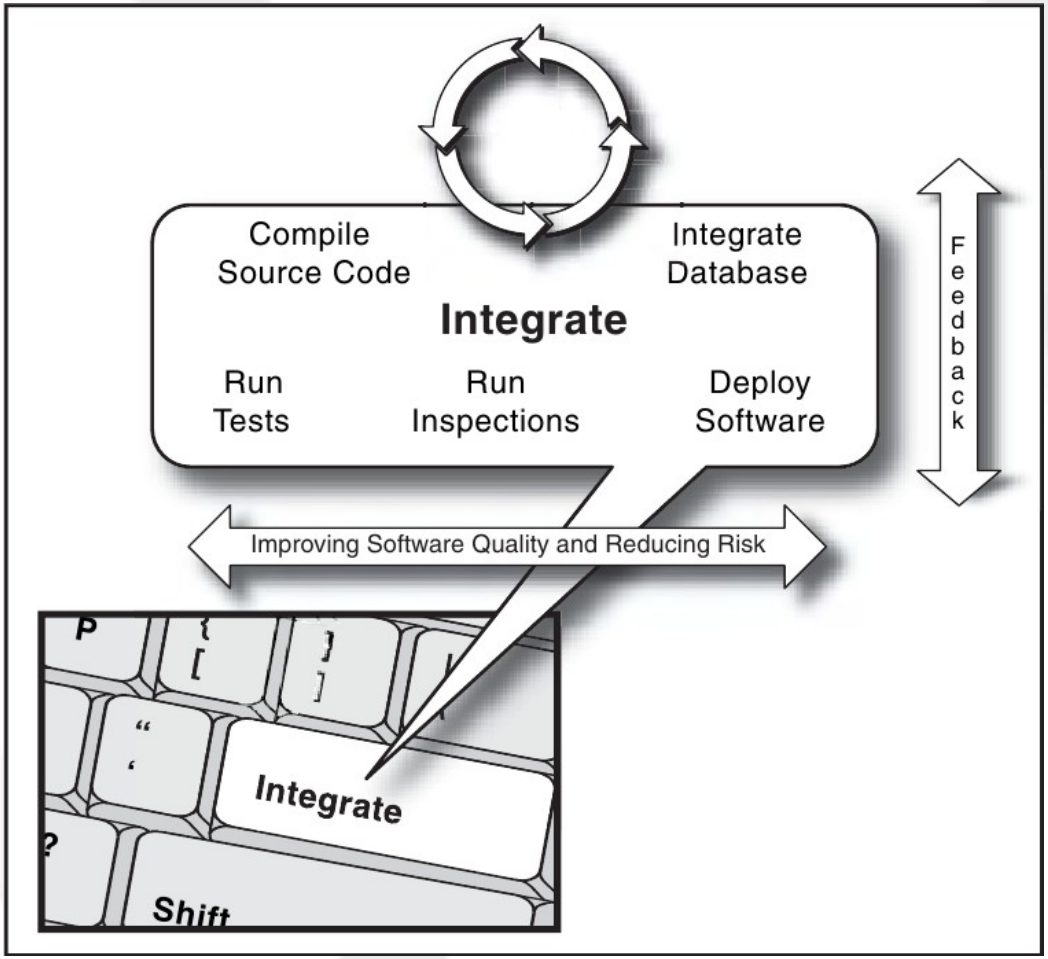
- Ao adotar IC, pode-se responder as seguintes questões:
  - Os componentes do *software* continuam funcionando quando estão operando em conjunto ?
  - Qual a complexidade do meu código ?
  - A equipe está aderindo às padronizações de codificação definidas ?
  - Qual porcentagem do código está coberto pelos testes automáticos ?
  - Todos os testes foram realizados com sucesso após a última mudança ?
  - A aplicação ainda atende aos requisitos de desempenho ?
  - Houve algum problema ao implantar o sistema após a última mudança ?

Integração Contínua

- O servidor de IC:
  - Executa o *build* sempre que uma mudança é enviada para o repositório
  - Tipicamente é configurado para fazer *pooling* no repositório com uma determinada frequência
  - Podem também ser configurados para executar o *build* com uma determinada frequência, independente das mudanças ocorridas (isto não é mais Integração Contínua, entretanto)
  - Geralmente disponibilizam um *dashboard* para publicação dos resultados
  - Geralmente utiliza alguma *build tool* (Ant, Nant, MSBuild, Rake, QMake, CMake, etc) – independente de IDE

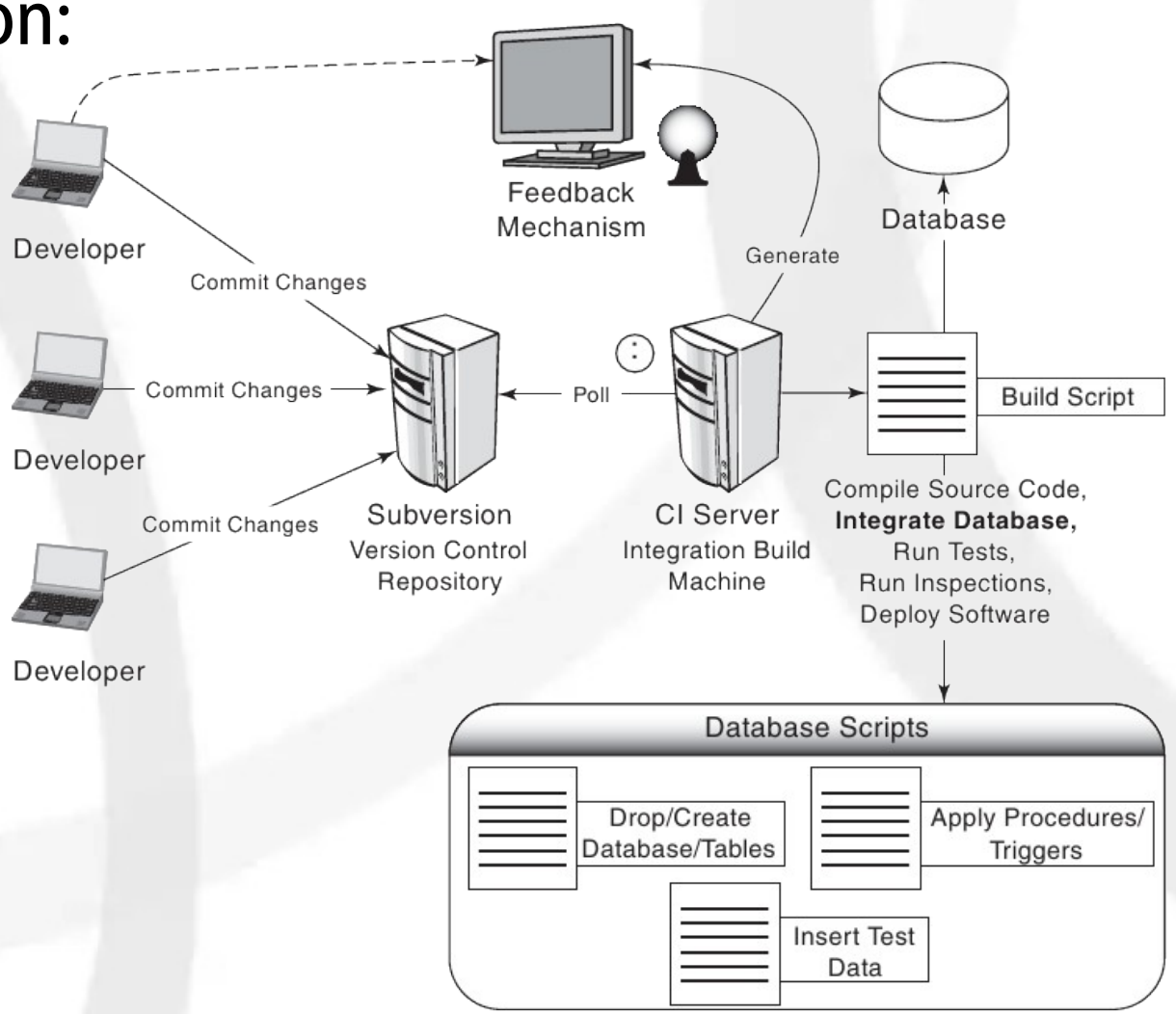
Integração Contínua

- "The Integrate Button":



Integração Contínua

- Database Integration:



Integração Contínua

- Pode-se ainda integrar:
  - Testes
  - Inspeções
  - Implantações
  - Geração de documentação

Integração Contínua

- Um dia na vida de um desenvolvedor que usa IC:
  - Zé chega na empresa pela manhã e verifica o monitor *widescreen* exibindo informações em "tempo-real" sobre o seu projeto
  - O monitor informa que a última integração ocorreu com sucesso há pouco minutos
  - O monitor apresenta ainda uma lista de valores recentes para as métricas de qualidade, incluindo aderência às padronizações, duplicação de código, etc
  - Zé é um dos 15 desenvolvedores Java trabalhando em um *software* para gerenciamento de uma cervejaria

Integração Contínua

- Um dia na vida de um desenvolvedor que usa IC:
  - No início do dia, Zé refatora um sub-sistema cujo servidor de IC indicou possuir muito código duplicado
  - Antes de realizar o *commit* no repositório, Zé executa um *build* privado, que compila o sub-sistema e executa testes de unidade no novo código produzido
  - Após executar o *build* privado ele realiza o *commit* no repositório
  - Poucos minutos depois o servidor de IC detecta a mudança enviada por Zé e executa um *build* de integração

Integração Contínua

- Um dia na vida de um desenvolvedor que usa IC:
  - Este *build* de integração executa ferramentas automatizadas de inspeção para verifica se o código continua em conformidade com as padronizações adotadas
  - Zé recebe um *e-mail* indicando uma violação na padronização, rapidamente executa as correções e as envia de volta ao repositório
  - O servidor de IC executa novamente o *build* de integração
  - Zé constata no relatório web gerado pelo servidor que a quantidade de código duplicado no sistema foi reduzida

Integração Contínua

- Um dia na vida de um desenvolvedor que usa IC:
  - Após o almoço, outro desenvolvedor da equipe – Maria – chega à sala de Zé, falando:
    - Maria: "Eu acho que as mudanças que você fez pela manhã quebraram o último *build*"
    - Zé: "Hmm … mas eu executei os testes"
    - Maria: "Oh, eu não tive tempo de escrever os meus testes !"
    - Zé: "Você está seguindo a métrica de cobertura de código que definimos para o projeto ?"
  - Consequentemente, eles decidem considerar um *build* incorreto se a métrica de cobertura de código estiver abaixo de 85%
  - Maria programa o teste para o problema e o corrige

Integração Contínua

- O que se ganha com a IC ?
  - Redução de riscos
  - Redução de processos manuais repetitivos
  - Geração de *software* pronto para instalação a qualquer momento e em qualquer lugar (plataforma)
  - Melhora a capacidade de visualização do *status* do projeto
  - Faz com que a equipe de desenvolvimento fique mais segura em relação ao produto sendo desenvolvido

Integração Contínua

- Porque as equipes ainda não usam IC ?
  - Maior custo ao manter um servidor de IC
    - É um mito, porque a necessidade de integração, testes, inspeções e implantações continua existindo
    - Gerenciar um sistema robusto de IC é melhor que gerenciar um processo manual
  - Exige mudanças nos processos organizacionais
    - Uma abordagem incremental é mais efetiva: adiciona-se primeiro *builds* e testes com baixa frequência (ex: *nighly builds*)
  - Muitos *builds* com erros
    - Geralmente o desenvolvedor não executou o *build* privado ou algum arquivo ficou fora do *commit*

Integração Contínua

- Porque as equipes ainda não usam IC ?
  - Custos adicionais de *hardware* e *software*
    - É necessário uma máquina dedicada ao servidor de IC. Entretanto, este custo é menor do que o custo de encontrar um problema nas fases finais do processo
  - Estas atividades deveriam ser feitas pelos desenvolvedores
    - É verdade, mas com IC elas são realizadas com maior frequência, efetividade e confiabilidade, em um ambiente separado
    - Este ambiente separado pode ser "inicializado" antes de cada *build* , reduzindo hipóteses e conduzindo a decisões mais acertadas
- Integração Contínua não é Compilação Contínua

Integração Contínua

- Integração Contínua e você – dicas:
  - Faça *commits* frequentes
  - Não faça *commit* de código que não funciona
  - Corrija *build* com problema imediatamente
  - Escreva testes automatizados
  - O *build* deve passar em todos os testes e inspeções para ser considerado correto
  - Execute *builds* privados
  - Evite fazer *checkout* de código que não funciona

Integração Contínua

- Reduzindo riscos com IC:
  - "Mas … funciona na minha máquina !!!"
    - John: "Estamos tendo problema com o último *build* no servidor de IC"
    - Adam: "Estranho, estava funcionando na minha máquina. Deixe-me ver … veja, continua funcionando"
    - John: "Descobri o problema, você não fez o *commit* dos novos arquivos no repositório"
  - Solução:
    - Use uma máquina separada para fazer a integração e garante que tudo o que for preciso para construir o *software* está no repositório

Integração Contínua

- Reduzindo riscos com IC:
  - "Sincronizando com o banco de dados"
    - Lauren: "Estou tendo problemas ao usar o *build* 1345 com a versão 1.2.1.b1 do banco de dados"
    - Pauline: "Não, com o *build* 1345 você tem que usar a versão 1.2.1.b2 do banco de dados"
    - Lauren: "Acabei de perder 4h de trabalho para nada"
    - Pauline: "Bom, você devia ter falado comigo antes"
  - Solução:
    - Coloque os artefatos de banco de dados no repositório
    - Recrie o banco do zero e adicione os dados usando o *script* de *build*

Integração Contínua

- Reduzindo riscos com IC:
  - "The Missing Click"
    - Rachel: "O último *build* foi feito com a versão mais recente do código, presente no servidor de desenvolvimento ?"
    - Kelly: "John saiu para o almoço. Ele deve ter atualizado o servidor"
    - Rachel: "Bom, vamos esperar ele retornar"
    - … mais tarde …
    - Rachel: "John, o que acontecem com o último *build* ? Parece que os JSPs não foram pré-compilados, estamos recebendo *runtime errors*"
    - John: "Hmm, desculpe. Devo ter esquecido de marcar a opção de pré-compilação de JSP quando fiz a implantação ontem"

Integração Contínua

- Reduzindo riscos com IC:
  - Solução:
    - Automatize o processo de implantação através da sua adição nos *scripts* de *build*
    - Não é mais necessário esperar que alguém implante o *software* no servidor de desenvolvimento
    - Tem-se sempre uma "versão testável" do último código desenvolvido

Integração Contínua

- Reduzindo riscos com IC:
  - "Testes de Regressão":
    - Sally: "Eu percebi que a última versão implantada no ambiente de testes está com o mesmo *bug* que tínhamos alguns meses atrás. O que aconteceu ?"
    - Kyle: "Não sei. Eu testei todas as mudanças recentes que fiz"
    - Sally: "Você executou os outros testes para as outras partes do sistema ?"
    - Kyle: "Não, não tive tempo de manualmente executar todos eles. É provavelmente por isso que não encontrei este *bug* antes"

Integração Contínua

- Reduzindo riscos com IC:
  - Solução:
    - Programe testes para todo o seu código-fonte, provavelmente utilizando algum *framework* tipo JUnit, QTestLib
    - Rode os testes a partir do *script* de *build*
    - Execute os testes continuamente como parte do seu sistema de IC, de modo que eles sejam executados em cada *checkin*

Integração Contínua

- Reduzindo riscos com IC:
  - "Cobertura dos Testes":
    - Evelyn: "Você rodou os testes de unidade antes de realizar o *commit* do seu código ?"
    - Noah: "Sim"
    - Evelyn: "Ótimo. Como está a outra funcionalidade que você está implementando ?"

Integração Contínua

- Reduzindo riscos com IC:
  - "Cobertura dos Testes" - mais segurança:
    - Evelyn: "Você rodou os testes de unidade antes de realizar o *commit* do seu código ?"
    - Noah: "Sim"
    - Evelyn: "O código atual passou por todos os testes ?"
    - Noah: "Sim"
    - Evelyn: "Como você sabe se uma quantidade suficiente do código foi adequadamente testada ?"

Integração Contínua

- Reduzindo riscos com IC:
  - Solução:
    - Execute uma ferramenta de *code coverage* para avaliar a quantidade de código-fonte que é realmente executada pelos testes
    - A maioria das ferramentas apresenta a porcentagem da cobertura por pacote e por classe

Integração Contínua

- Reduzindo riscos com IC:
  - "Você recebeu o documento ?":
    - Evelyn: "Você está trabalhando em que, Noah ?"
    - Noah: "Estou esperando que o último *build* seja implantado para então iniciar os testes"
    - Evelyn: "O último *build* foi implantado no servidor de testes dois dias atrás. Você não soube ?"
    - Noah: "Não, estive fora do escritório nos últimos dias"
  - Solução:
    - O servidor de IC envia e-mails ou mensagens SMS para as partes interessadas sempre que um *build* falha

Integração Contínua

- Reduzindo riscos com IC:
  - "Incapacidade de visualizar o *software*"
    - Maile: "Olá, sou novo na equipe e gostaria de revisar o projeto. Há algum diagrama UML que eu possa olhar ?"
    - Allie: "Brr, não usamos UML aqui. Tudo o que você tem que fazer é olhar o código-fonte. Se você não consegue entender pelo código, talvez você não consiga trabalhar direito aqui"
    - Maile: "Ok, eu simplesmente pensei que se eu olhasse para uma figura geral do projeto eu conheceria mais rapidamente a arquitetura da aplicação. Sou uma pessoa visual"
  - Solução:
    - Gere automaticamente os diagramas: *doxygen*, Javadoc, etc

Integração Contínua

- Reduzindo riscos com IC:
  - "Conformidade com a padronização do código"
    - Brian: "Estou tendo dificuldade em ler o seu código. Você leu o documento de 30 páginas sobre a padronização de código ?"
    - Lindsay: "Estou usando a padronização de código do meu trabalho anterior. O código que escrevi é inerentemente complexo, então pode ser difícil pra você entendê-lo rapidamente"
    - Brian: "Escrever código que outros não conseguem ler não faz de você uma pessoa mais inteligente. Está fazendo com que eu demore mais tempo para revisá-lo e atualizá-lo. Por favor, revise a padronização e conserte o seu código"

Integração Contínua

- Reduzindo riscos com IC:
  - Solução:
    - Em vez de criar um documento de 30 páginas, cria-se uma classe anotada com todas as padronizações de codificação
    - Ferramentas automáticas de inspeção verificam a aderência ao estilo, como parte do *script* de *build*
    - Exs: checkstyle, PMD, beautifiers

Integração Contínua

- Reduzindo riscos com IC:
  - "Aderência arquitetural"
    - Jenn: "Vocês estão seguindo o estilo arquitetural adotado ? Encontrei problemas em um dos controladores: um de vocês está acessando a camada de dados diretamente"
    - Mark e Charlie: … perplexos …
    - Jenn: "O motivo de eu ter criado todos aqueles diagramas UML é para que todos seguissem o estilo arquitetural. Vocês não estão seguindo o protocolo estabelecido"
    - Charlie: "Eu olhei todos estes diagramas no início do projeto, mas a arquitetura mudou algumas vezes de lá pra cá, então é difícil segui-la"

Integração Contínua

- Reduzindo riscos com IC:
  - Solução:
    - Adicione ferramentas automatizadas de inspeção para verificar a aderência ao estilo arquitetural
    - Exs: JDepend, NDepend

Integração Contínua

- Reduzindo riscos com IC:
  - "Código duplicado":
    - Mary: "Você sabe como faço para iterar em uma coleção de objetos User ?"
    - Adam: "Sim, fiz um código para isso na semana passada. Você pode encontrá-lo no pacote User"
    - Mary: "Ótimo. Vou copiá-lo e aplicar no meu problema. Obrigado"
  - Solução:
    - Use ferramentas de inspeção tipo PMD e CPD para reportar código duplicado
    - Reduza a duplicação através de *refactoring*

Integração Contínua

- Conclusões:
  - Integração Contínua como ferramenta indispensável para melhorar a produtividade e gerenciar a qualidade do processo e do produto
  - Mecanismos cada vez mais sofisticados de integração estão sendo desenvolvidos
  - Estudo de caso: Jenkins

Pós-Graduação em Computação Distribuída e Ubíqua

INF612 - Aspectos Avançados em Engenharia de Software
Gerência de Qualidade e Integração Contínua

Sandro S. Andrade
sandroandrade@ifba.edu.br