**Universidade Federal de Campina Grande**
**Centro de Ciências e Tecnologia**
**Departamento de Sistemas e Computação**

**Programa Especial de Treinamento**
**PET**


# Um Processo de Desenvolvimento de Software

**Campina Grande, fevereiro de 2004**

# Sumário

# 1. Introdução

A necessidade de se utilizar melhores práticas para desenvolvimento de software no meio acadêmico, que possibilitem maior sucesso na implementação de projetos oferecidos em algumas disciplinas, foi a principal motivação para a criação do easYProcess.

Sendo assim, o YP se apresenta como um processo simplificado, apoiado em práticas do XP, RUP e *Agile Modeling*. Surgido pela iniciativa da professora Drª Francilene Procópio Garcia, uma das responsáveis pelo Grupo de Pesquisa e Desenvolvimento em Engenharia de Software do Departamento de Sistemas e Computação da Universidade Federal de Campina Grande, a concepção do YP se deu no ambiente do grupo PET-Computação, do qual a professora é a atual tutora.

A equipe do YP é composta por 9 componentes, entre graduandos e alunos do PET. Na fase de desenvolvimento do processo, a equipe foi dividida em três grupos de estudos entre os processos XP, RUP e *Agile Modeling*, para que as práticas utilizadas por estes três processos fossem confrontadas no intuito de se extrair as melhores para a realidade do YP.

Nesta fase inicial o YP será utilizado na disciplina de Laboratório de Engenharia de Software como uma versão piloto para a análise dos resultados da sua aplicação na academia. Paralelamente, estará sendo avaliado pela equipe de criação a fim de identificar melhorias, buscando torná-lo um processo robusto e capaz de suprir as necessidades das equipes de desenvolvimento que o utilizam.

Espera-se que o easYProcess possa auxiliar o gerenciamento do progresso no desenvolvimento de aplicações durante as disciplinas que assim necessitem, além de se tornar uma metodologia utilizada pelos alunos da graduação e que seja expandido para o uso em projetos de pequeno e médio porte em empresas.

# 2. Equipe de Desenvolvimento

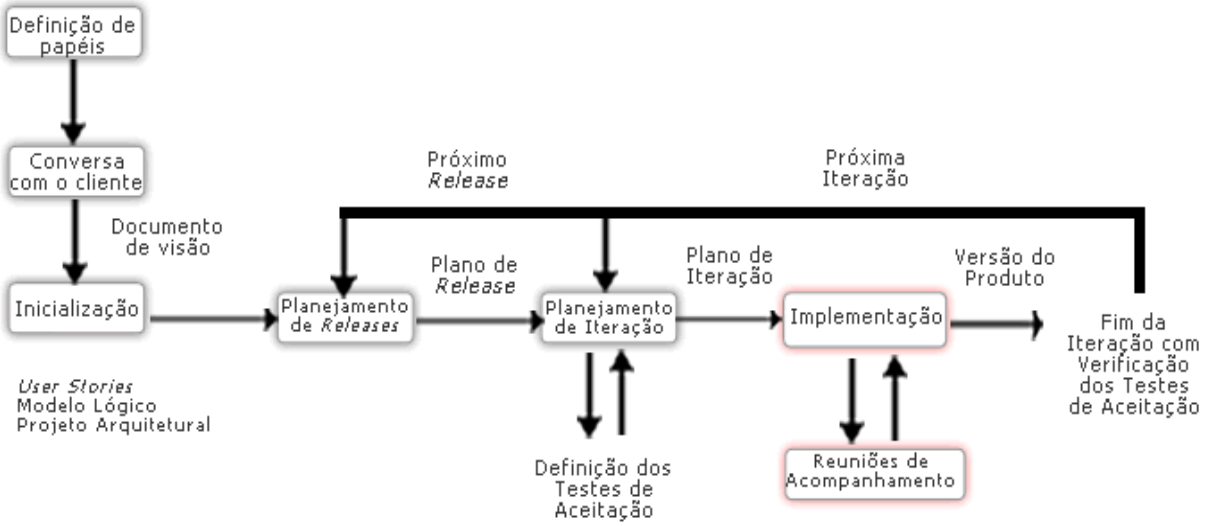
**Orientadora**

Francilene Procópio Garcia ..... garcia@dsc.ufcg.edu.br

**Equipe de desenvolvimento:**

Aliandro Higino Guedes Lima ..... aliandro@dsc.ufcg.edu.br
Danilo de Sousa Ferreira ..... danilo@dsc.ufcg.edu.br
Fábio Luiz Leite Júnior ..... fabio@dsc.ufcg.edu.br
Giselle Regina Chaves da Rocha ..... giselle@dsc.ufcg.edu.br
Gustavo Wagner Diniz Mendes ..... gustavo@dsc.ufcg.edu.br
Renata França de Pontes ..... renata@dsc.ufcg.edu.br
Verlaynne Kelley da Hora Rocha ..... verla@dsc.ufcg.edu.br
Vinicius Farias Dantas ..... vinicius@dsc.ufcg.edu.br
Yuska Paola Costa Aguiar ..... yuska@dsc.ufcg.edu.br

# 3. Síntese do easYProcess



# 4. Fluxo de Trabalho

1. Identificação do escopo do problema;
2. Especificação de Papéis;
3. Conversa com o cliente;
4. Visão sobre processo de negócio;
5. Inicialização:
   5.1 Especificação das *User Stories*;
   5.2 Projeto Arquitetural;
   5.3 Modelo Lógico;
6. Planejamento:
   6.1 Alocação de *User Stories* nos *releases*;
   6.2 Definição do *release*:
      6.2.1 Alocação das *User Stories* do *release* em questão nas iterações;
      6.2.2 Definição da Iteração:
         6.2.2.1 Definição dos testes de aceitação das *User Stories*;
         6.2.2.2 Desdobramento das *User Stories* da iteração em tarefas;
         6.2.2.3 Construção da Tabela de Alocação de Tarefas (TAT) e preenchimento com os seguintes dados: tarefa, responsável e tempo estimado para realização da tarefa;
7. Implementação:
   7.1 Testes contínuos;
   7.2 Revisão de código;
   7.3 Refatoramento;
   7.4 Pequenos *releases*;
8. Finalização da Iteração. A TAT deve ser concluída (preenchimento dos campos status e tempo real do desenvolvimento de cada tarefa). Isto deve ser feito na reunião de acompanhamento, ao final da iteração;
9. Os passos 6.2 até 8 devem ser repetidos enquanto houver *releases* ou iterações.

**Obs.:** As reuniões de acompanhamento ocorrem ao longo de todo o processo, consistindo da análise do *Big Chart*, análise de riscos, e preenchimento da TAT, este último na reunião ao final da iteração.

# 5. Papéis no processo

Ao montar uma equipe de desenvolvimento de software sugere-se uma divisão de tarefas entre os membros da mesma, de forma que cada um assuma um determinado papel no desenvolvimento. Um papel constitui um conjunto de responsabilidades que determinam qual será o comportamento de uma pessoa durante o processo. No YP recomenda-se a presença de cinco papéis: cliente, usuário, gerente, desenvolvedor e testador.

A alocação dos papéis deve ser feita de acordo com as necessidades e escopo do projeto, levando-se em conta as habilidades e características de personalidade das pessoas envolvidas no processo. A equipe de desenvolvimento deve ser estruturada de forma a se obter a maior produtividade possível.

É essencial que todos os papéis estejam presentes na equipe, pois caso contrário pode ocorrer que algumas etapas importantes do processo sejam esquecidas ou não recebam a atenção necessária. A alocação dos papéis de gerente e testador é dinâmica, significando que um integrante do projeto pode assumir ou se liberar de um destes papéis em determinada etapa do processo.

Um papel não corresponde necessariamente a uma pessoa da equipe, ou seja, uma mesma pessoa pode desempenhar vários papéis simultaneamente. Em equipes pequenas, por exemplo, com três integrantes apenas, isto é geralmente necessário, porém deve-se ter cuidado para que o acúmulo de papéis não leve a uma sobrecarga de responsabilidades de algum membro da equipe.

O papel do cliente é um dos mais importantes, mas nem sempre o cliente possui tempo ou está disposto a participar do desenvolvimento do software. É importante então deixar bem claro para ele quais são as suas responsabilidades e por que sua participação ativa é tão importante para o bom andamento e sucesso do projeto.

**PAPÉIS**

**Cliente** – papel desempenhado por quem solicitou o desenvolvimento do software. Sua presença ativa no processo é de suma importância, já que o software está sendo desenvolvido para ele, e deve portanto atender às suas necessidades e preferências. A ausência do cliente pode levar ao fracasso do projeto, com um produto final que não atende a suas necessidades ou cuja interface é de difícil aprendizado. As principais responsabilidades do cliente são:

- Definir as funcionalidades do sistema: O cliente deve transmitir o que ele espera que seu sistema seja capaz de fazer, ou seja, quais as funcionalidades e características que o software deve ter para resolver o seu problema;
- Priorizar as funcionalidades: Devem ser especificadas quais partes do sistema são de maior urgência e devem ser implementadas

primeiramente, de forma a permitir que seja feito o planejamento adequado das tarefas a serem realizadas;

- Ajudar na elaboração do plano de *releases*: O cliente deve participar da definição do plano de *releases* junto aos desenvolvedores, de forma que ambas as partes concordem sobre o que deve estar pronto ao final de cada etapa do desenvolvimento;
- Explicitar os testes de aceitação: Testes de aceitação são testes que avaliam a eficiência do sistema ao cumprir uma determinada situação. Servem como referencial para que o cliente verifique se o software está realizando a função que lhe foi prometida. Maiores informações sobre testes de aceitação podem ser obtidas no tópico sobre Testes, página 27;
- Ser ativo no processo de desenvolvimento do sistema: O cliente deve estar disponível o máximo de tempo possível para interagir com os desenvolvedores, seja na definição do sistema, no refinamento de protótipos ou na implantação dos testes de aceitação. Um cliente ausente pode resultar em modelos mal-definidos e sistemas que não atendem aos requisitos do problema.

**Usuário** – o usuário é quem vai de fato utilizar o sistema que será produzido. Muitas vezes, no desenvolvimento de um sistema, o desenvolvedor tende a projetá-lo de acordo com suas próprias preferências, ignorando as preferências de quem realmente irá usar o sistema. Este procedimento geralmente leva à produção de um software de difícil uso e que não satisfaz os desejos do usuário final. A presença do usuário no processo permite que o sistema seja desenvolvido de acordo com seu perfil, e relembra constantemente o desenvolvedor de que ele está desenvolvendo um sistema que será utilizado por outras pessoas. Vale lembrar que em algumas situações podemos ter que o cliente seja também o usuário do sistema. As principais responsabilidades do usuário são:

- Ajudar a definir os testes de aceitação: O usuário pode ajudar a definir os testes de aceitação para garantir que determinada funcionalidade do sistema que ele considera importante esteja sendo realizada do modo como ele deseja;
- Avaliar continuamente a interface do sistema: O usuário deve estar constantemente avaliando a interface do sistema, para sugerir alterações que tornem esta interface mais agradável e de fácil utilização. Uma interface mal-projetada e que não leva em consideração as preferências do usuário resulta em um sistema de difícil aprendizado e que o usuário provavelmente evitará utilizar.

**Gerente** – é o responsável por coordenar as atividades de todos os outros membros da equipe. O gerente deve ser capaz de gerenciar o desenvolvimento e tomar decisões referentes aos riscos e rumos do projeto. As principais competências do gerente são:

- Conduzir o planejamento e as ações dos desenvolvedores: O gerente deve garantir que os desenvolvedores estejam executando corretamente suas atividades, e no tempo estabelecido, para assegurar o bom andamento do projeto;
- Elaborar o plano de desenvolvimento (*release* e iteração): Deve ser realizado junto com os desenvolvedores um planejamento das tarefas a serem realizadas, alocando-se responsáveis para cada tarefa;
- Avaliar sistematicamente os riscos descobertos: A análise de riscos é essencial e deve ser um elemento constante durante o processo. Os riscos devem ser confrontados e sua influência na viabilidade do projeto deve ser considerada;
- Coletar e analisar métricas: As métricas são informações que permitem a análise do andamento do projeto pelo gerente, tais como número de linhas de código e número de testes implementados. O gerente deve coletar métricas periodicamente junto à equipe de desenvolvimento e utilizá-las na identificação de possíveis riscos e deficiências no projeto;
- Presidir as reuniões de acompanhamento: O gerente deve realizar semanalmente reuniões de acompanhamento com a equipe de desenvolvimento, onde serão avaliados o progresso e as pendências do projeto, utilizando a análise de riscos e o *Big Chart*. Este acompanhamento constante é essencial para que se tenha uma gerência eficiente no processo;
- Alocar testadores: É responsabilidade do gerente de projeto decidir quem serão os testadores do código produzido por cada um dos desenvolvedores. O código e os testes de unidade produzidos por um desenvolvedor devem ser revisados por outro membro da equipe, a fim de identificar falhas e áreas do software não-testadas;
- Gerenciar configurações: O gerente de projeto deve manter um controle das versões dos artefatos gerados, de forma a garantir que os desenvolvedores estejam sempre utilizando a mesma versão destes artefatos, e que as alterações feitas por um desenvolvedor não produzam artefatos inconsistentes. É aconselhável a utilização de uma ferramenta específica para este controle de versões;
- Resolver conflitos internos: O gerente de projeto deve atuar para a manutenção de um ambiente agradável para o desenvolvimento, evitando conflitos e outras perturbações;
- Tornar acessível à documentação do projeto: A documentação do projeto deve estar disponível para que possa ser consultada a qualquer momento pelo time de desenvolvimento. Também é interessante que o cliente possa ter acesso a estas informações, como forma de acompanhar o progresso do projeto.

**Desenvolvedor** – O papel do desenvolvedor consiste em modelar os requisitos do sistema, e posteriormente produzir um código eficiente e

correto que corresponda a tais requisitos. É seu dever verificar a existência de falhas em seu código e, se encontradas, corrigi-las. Para esta verificação devem ser gerados testes de unidade relativos ao código produzido. As principais responsabilidades do desenvolvedor são:

- Levantar os requisitos funcionais e não funcionais junto ao cliente: Deve participar da modelagem do sistema, procurando extrair do cliente todas as informações necessárias para o desenvolvimento do software;
- Auxiliar o gerente na elaboração de um plano de desenvolvimento: O desenvolvedor deve participar dos planejamentos de *releases* e de iterações, pois ele tem noção de sua capacidade de geração de código e das dificuldades envolvidas no cumprimento de determinada tarefa. Sua participação no planejamento evita que o gerente superestime as habilidades de seu time de desenvolvimento e atribua prazos irreais à realização das tarefas;
- Gerar testes de unidade: Cada desenvolvedor deve gerar testes de unidade para o código por ele produzido. O ideal é que tais testes sejam gerados em paralelo com a codificação, o que possibilita um maior entendimento do código e, portanto, a realização de testes mais completos;
- Manter a integração contínua de código: Em projetos onde há mais de um desenvolvedor é necessário que o código gerado por cada um seja integrado com o código produzido pelos demais, para identificar eventuais conflitos. Esta integração deve ocorrer de forma contínua para evitar a perpetuação de conflitos entre partes do software até etapas posteriores, onde tais conflitos serão mais difíceis de serem resolvidos.

**Testador** – O testador é o responsável por revisar o código e os testes gerados pelos desenvolvedores, além de implementar os testes de aceitação definidos pelo cliente. As principais responsabilidades do testador são:

- Efetuar revisões sobre o código de outro desenvolvedor: O testador deve revisar o código produzido pelos desenvolvedores, com o propósito de identificar possíveis falhas e necessidades de refatoramento, garantindo assim a qualidade do código gerado;
- Elaborar testes sobre o código de outro desenvolvedor: O testador deve analisar os testes de unidade feitos pelos desenvolvedores, identificar partes do código não testadas e refinar os testes para que cubram estas partes. O testador não deve ser a mesma pessoa que desenvolveu o código, pois o desenvolvedor tende a não enxergar as deficiências de seu próprio código;
- Gerar testes de aceitação: Antes de cada entrega de uma versão funcional, os testadores devem implementar os testes de aceitação definidos pelo cliente.

# Resumo

**Papéis:**

- Cliente;
- Usuário;
- Gerente;
- Desenvolvedor;
- Testador.

**Responsabilidades do Cliente:**

- Definir as funcionalidades do sistema;
- Priorizar as funcionalidades;
- Ajudar a elaborar o plano de *releases*;
- Explicitar os testes de aceitação;
- Ser ativo no processo de desenvolvimento de sistema.

**Responsabilidades do Usuário:**

- Ajudar a definir os testes de aceitação;
- Avaliar continuamente a interface do sistema.

**Responsabilidades do Gerente:**

- Conduzir os planejamentos e as ações dos desenvolvedores;
- Elaborar o plano de desenvolvimento (*release* e iteração);
- Avaliar sistematicamente os riscos descobertos;
- Coletar e analisar métricas;
- Alocar testadores;
- Gerenciar configurações;
- Presidir as reuniões de acompanhamento;
- Resolver conflitos internos;
- Tornar acessível à documentação do projeto.

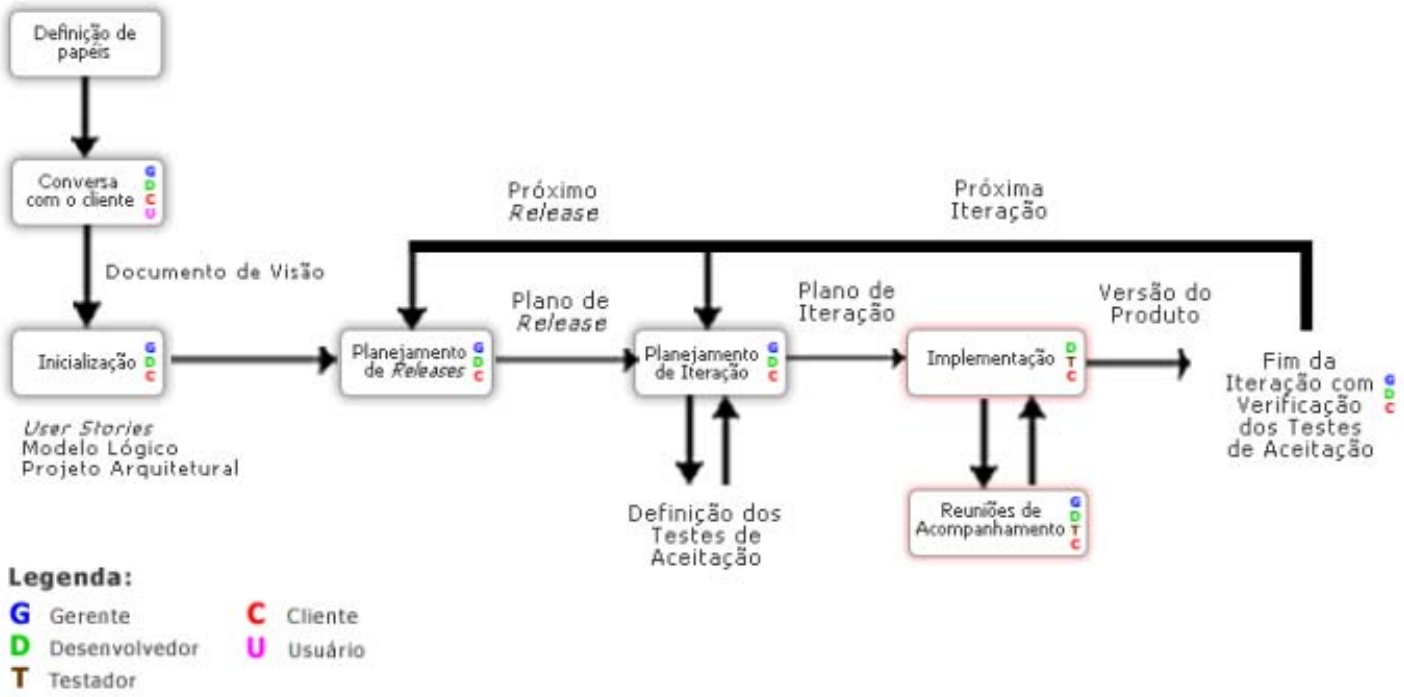
**Responsabilidades do Desenvolvedor:**

- Levantar os requisitos funcionais e não funcionais junto ao cliente;
- Auxiliar o gerente na elaboração de um plano de desenvolvimento;
- Gerar testes de unidade;
- Manter a integração contínua de código.

**Responsabilidades do Testador:**

- Efetuar revisões sobre o código de outro desenvolvedor;
- Elaborar testes sobre o código de outro desenvolvedor;
- Gerar testes de aceitação.

## Representação dos papéis no YP:



# 6. Primeira Conversa com o Cliente

A partir da conversa com o cliente, após uma identificação do escopo do problema, um artefato contendo a visão sobre os processos de negócios do cliente deve ser gerado. Deve-se ter em mente que perguntas serão feitas durante a conversa a fim de extrair informações importantes sobre o que é realmente importante para o cliente. Esta conversa deve ser entre o cliente e a equipe de desenvolvimento.

**O que deve ser feito antes:**

- Entender o escopo do problema: os desenvolvedores devem se informar sobre o domínio do problema;
- Elaborar um roteiro de perguntas: este roteiro tem o objetivo de ajudar a equipe a levantar as informações mais importantes.

**Durante a conversa:**

- Tornar a reunião produtiva: procurar extrair o máximo de informações em um tempo mínimo (depende da disponibilidade do cliente) e incentivar a participação do cliente para que este se sinta parceiro do processo;
- Usar uma linguagem simples: evitar o uso de termos técnicos, para que não haja dúvidas nem interpretações distintas por nenhuma das partes;
- Fazer uso de analogias: analogias podem ser usadas para possibilitar o maior entendimento entre o cliente e a equipe de desenvolvimento;
- Informar ao cliente o seu papel: esta informação deve ser dada de forma sutil, mostrando ao cliente a necessidade de sua participação para o sucesso do desenvolvimento do software;
- Agendar reuniões periódicas com o cliente: antecipar uma agenda de reuniões com o cliente é importante. O cliente deve se sentir parte integrante do projeto com a motivação de garantir que as suas necessidades sejam satisfeitas.

**O que a equipe deve estar sabendo ao final da conversa:**

- Ter uma idéia bem definida do escopo do problema: um passo importante para elaboração do documento de visão;
- Conhecer o perfil dos usuários do sistema e suas necessidades;
- Saber quais os são requisitos funcionais e não funcionais do sistema;
- Listar os riscos do projeto: a equipe de desenvolvimento deve saber quais as dificuldades mais críticas do projeto e os riscos envolvidos. Alguns desses riscos podem levar o projeto ao fracasso, expondo a necessidade de se analisar sua viabilidade.

## Resumo

**O que deve ser feito antes da conversa com o cliente:**

- Entender o escopo do problema;
- Elaborar um roteiro de perguntas.

**Durante a conversa:**

- Tornar a reunião produtiva;
- Usar uma linguagem simples;
- Fazer uso de analogias;
- Informar ao cliente o seu papel;
- Agendar reuniões periódicas com o cliente.

**O que a equipe deve estar sabendo ao final da entrevista:**

- Ter uma idéia bem definida do escopo do problema;
- Conhecer o perfil dos usuários do sistema e suas necessidades;
- Saber quais são os requisitos funcionais e não funcionais do sistema;
- Listar os riscos do projeto.

# 7. Documento de Visão

Concluída a conversa inicial com o cliente, a primeira tarefa a ser feita passa a ser a elaboração de um documento de visão, contendo as idéias gerais sobre o que o sistema se propõe a fazer de forma que possa apoiar os processos de negócios do cliente.

Neste momento, o desenvolvedor possui as informações fornecidas pelo cliente e já tem uma idéia bem amadurecida acerca do sistema a ser desenvolvido. Por isso, esta é a ocasião ideal para se pensar no sistema como um todo, documentando suas características gerais.

O documento de visão deve ser feito utilizando uma linguagem de fácil entendimento para o cliente, uma vez que deve ser usado como elo de comunicação entre este e o desenvolvedor. Uma vez produzido, o documento deve ser validado junto ao cliente no intuito de se descobrir se a visão de negócio do sistema, definida pela equipe de desenvolvimento, realmente está correta. Sendo validada, a visão servirá como base para a atividade de planejamento e poderá sempre ser consultada pela equipe quando houver qualquer dúvida sobre o que foi acertado com o cliente no momento inicial. Analisando desta maneira, nota-se que o documento de visão funciona como um contrato entre cliente e desenvolvedor, de forma que a equipe de desenvolvimento não deve ”imaginar” coisas novas incoerentes com o que especifica este “contrato”.

Entretanto, o documento de visão não deve ser visto como o ponto final sobre o que deve ser feito. Ocorrendo mudanças de requisitos durante o andamento do processo, a visão deve refletir essas modificações. Porém, o desenvolvedor deve estar sempre atento, pois nem sempre uma mudança de requisitos se reflete em mudanças na visão de negócio do cliente.

É importante que o documento de visão seja objetivo e traga uma quantidade mínima de informações. É essencial se começar com uma breve descrição do problema, na intenção de se tornar mais fácil descrever os processos de negócios do cliente e como o sistema se propõe a resolver. Se o problema possui regras de negócio e termos pouco familiares à equipe, talvez seja necessário se construir um glossário. A má compreensão de idéias acerca do problema pode vir a colocar o projeto por água abaixo. Deve-se aproveitar o documento de visão para definir quem são os usuários finais e demais envolvidos com o sistema, além de procurar saber quais as suas necessidades.

Deve-se então definir as características gerais do produto, bem como os requisitos funcionais e não funcionais. É sempre bom ter em mente que este é um documento com informações de alto nível. Os detalhes de baixo nível sobre os requisitos serão oportunamente definidos ao longo da etapa de inicialização. Na definição das características, é recomendável o uso de analogias, de forma que a compreensão da equipe seja maximizada. Analogias são bastante úteis quando o domínio do problema não é especialidade da equipe, ajudando todo o grupo a ter uma

idéia única e correta no que diz respeito ao negócio. Analogias também podem ser utilizadas durante a descrição do problema.

Para fechar o grupo de elementos essenciais que compõem o documento de visão, sugere-se a inclusão das limitações do projeto. A equipe deve ter consciência dos fatores que podem causar desvios indesejáveis no desenvolvimento. Destaca-se mais uma relevância do documento de visão: servir para que se possa analisar se o projeto é realmente viável.

## Resumo

**O que é?**

É um artefato que define em alto nível o domínio do problema, mapeando os processos de negócios do cliente a serem suportados pelo sistema. Uma maneira direta de comunicar o que se pretende no projeto para todos os envolvidos.

**Para que serve?**

- Avaliar se a equipe de desenvolvimento entendeu corretamente o domínio do problema descrito pelo cliente;
- Após ser validado pelo cliente, passa a ser um contrato entre este e o desenvolvedor;
- Sinalizar se houve mudanças muito grandes no projeto durante o processo;
- Faz com que o desenvolvedor pense no sistema como um todo, inserido no cenário de negócio do cliente, antes de começar a desenvolvê-lo, evitando alterações constantes e minimizando alguns tipos de riscos;
- É útil para avaliar se o desenvolvimento do projeto é viável.

**Quem faz?**

Os desenvolvedores, a partir de conversas com o cliente sobre o domínio do problema.

**Quando deve ser gerado?**

Logo após a primeira conversa com o cliente.

**Duração** [esforço para geração da visão]

Cerca de 30 minutos.

**O que o documento deve responder**:

- Quais os principais termos envolvidos no escopo do problema (glossário)?

- Qual(is) o(s) problema(s) a ser(em) resolvido(s)? Como tal(is) problema(s) afeta(m) o processo de negócio do cliente?
- Quem são os usuários e os demais envolvidos com o projeto? E quais as suas necessidades?
- Quais são as características do produto?
- Quais os requisitos funcionais?
- Quais são os requisitos não funcionais?
- Quais são as limitações do projeto?

# 8. Requisitos Não-funcionais

Requisitos são objetivos e/ou restrições, listados por clientes e usuários, que definem as características e funções de um sistema.

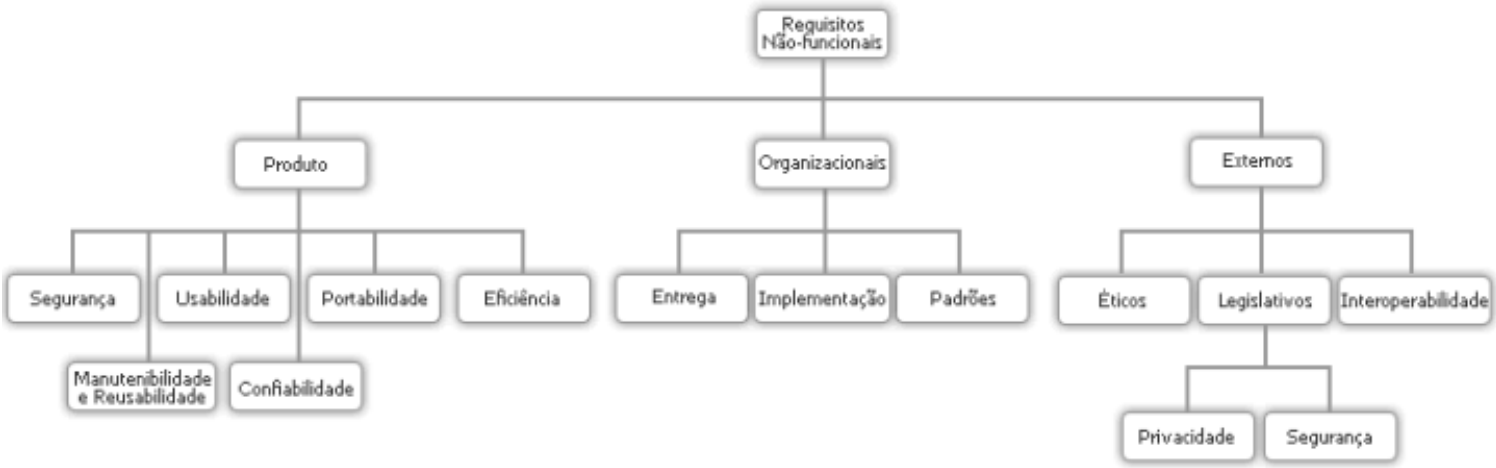
Existem dois tipos de requisitos, os funcionais e os não-funcionais. Os requisitos funcionais descrevem as várias funções que o software deve possuir, como por exemplo: “o sistema deve emitir relatórios de vendas diárias automaticamente”. Os requisitos não-funcionais, também chamados de requisitos de qualidade, definem as propriedades e restrições, tais como: manutenibilidade, usabilidade, desempenho. Um exemplo seria: “o tempo de resposta do sistema não deve ser superior a 30 segundos”.

Muitos erros ainda são cometidos durante a especificação dos requisitos, muitas vezes estes nem são detectados. Para que os requisitos possam ser verificados e validados uma especificação de requisitos deve ser:

- **Correta** – todo e qualquer requisito representa algo necessário;
- **Não ambígua** – possui uma única interpretação;
- **Completa** – tudo o que o sistema deve fazer está incluso na especificação;
- **Verificável** – para todo e qualquer requisito especificado deve existir um processo que permita verificar se o produto o atende.

Os requisitos não-funcionais podem ser classificados em três categorias de acordo com o tipo de especificações atendidas. São elas:

- **Requisitos do Produto** – requisitos que especificam a forma como o produto deve se comportar;
- **Requisitos Organizacionais** – requisitos que são conseqüências das políticas e dos procedimentos organizacionais;
- **Requisitos Externos** – requisitos que surgem a partir de fatores externos ao sistema.



# 9. Inicialização

Logo após a equipe de desenvolvimento adquirir uma idéia geral sobre problema a ser resolvido, devem ser iniciadas algumas atividades de análise do sistema.

O objetivo é identificar as funcionalidades do sistema, elaborar uma arquitetura que o descreva em alto nível e modelar o banco de dados, caso o projeto necessite de algum. Os três tópicos a seguir descrevem como alcançar tal objetivo:

### Definição das *User Stories*

Primeiramente, o cliente deve definir as *User Stories* do sistema. Não é necessário que todas elas sejam definidas neste momento. Durante o andamento do projeto outras poderão surgir, assim como as existentes podem ser modificadas ou até mesmo eliminadas. Uma dica: se a *User Story* for muito grande, divida-a. Se for muito pequena, agrupe-a com outras. Ao ser feita uma divisão, novos testes de aceitação devem ser definidos para cada *User Story* resultante, o que só pode ser feito pelo cliente.

Durante esta fase, deve ser feita uma estimativa do tempo de desenvolvimento de cada *User Story* (ao desdobrar em tarefas a estimativa inicial poderá ser confirmada ou não), para que a equipe possa expor ao cliente o tempo que o projeto requer e o tempo disponível. É importante também que o cliente identifique as *User Stories* que devem ser priorizadas. No início do desenvolvimento de um projeto é comum existir estimativas incorretas, mas com o tempo a equipe de desenvolvimento sente-se madura o suficiente para relacionar bem a sua velocidade de trabalho com o tempo de desenvolvimento, passando a acertar mais as estimativas.

A partir da definição das *User Stories* e de todos os artefatos anteriormente produzidos, é possível elaborar um projeto arquitetural e um modelo lógico de dados, caso haja um banco de dados envolvido.

### Projeto Arquitetural

Veja tópico “Projeto Arquitetural”, página 19.

### Modelo Lógico de Dados

Se houver um banco de dados envolvido, um modelo lógico de dados deve ser elaborado. Este modelo deve ser planejado com cuidado, pois mudanças nele freqüentemente acarretam atividades trabalhosas e demoradas de migração de dados. Com um modelo lógico de dados estável, a adaptação a mudanças ocorrerá facilmente.

## Resumo

**O que é?**

A fase de inicialização consiste em 3 atividades: (1) definição das *User Stories* do sistema, (2) elaboração do projeto arquitetural e (3) construção de um modelo lógico de dados, caso exista um banco de dados envolvido.

**Para que serve?**

Serve para identificar as funcionalidades do sistema por meio da especificação das *User Stories*, descrever o sistema em alto nível através do projeto arquitetural e definir a estrutura de algum banco de dados envolvido, construindo-se um modelo lógico de dados.

**Quem faz?**

- Definição de *User Stories*: cliente;
- Elaboração do projeto arquitetural: equipe de desenvolvimento;
- Aprovação do projeto arquitetural: cliente;
- Construção de um modelo lógico de dados: equipe de desenvolvimento.

**Duração**

Cerca de 8 horas.

**Obs:** Vale a pena investir tempo aqui, pois um projeto arquitetural consistente e um modelo lógico bem definido acomodam mudanças facilmente, reduzindo atividades trabalhosas.

# 10. Projeto Arquitetural

Este artefato tem como objetivo descrever o funcionamento do sistema num alto nível de abstração. Ele é útil quando se deseja explicitar como as partes do sistema a ser desenvolvido interagem entre si ou com outros sistemas.

Em geral, o projeto arquitetural contém informações que sofrem pouca ou nenhuma alteração ao longo do processo, a menos que grandes mudanças aconteçam. Os detalhes de projeto ou implementação não devem ser descritos.

O YP sugere que para compor o projeto arquitetural seja construído um diagrama contendo a estrutura da arquitetura, de forma que as possíveis dependências entre o sistema que está sendo desenvolvido e demais sistemas sejam explicitadas. Esse diagrama, se necessário, pode vir acompanhado de um texto que explique de forma objetiva os aspectos de maior relevância do sistema.

Ao elaborar o projeto arquitetural, a equipe deve estar ciente dos pontos críticos do sistema, das dependências existentes e dos requisitos levantados junto ao cliente (particularmente os não funcionais). Depois de pronto, o cliente deve aprovar o que foi elaborado pela equipe de desenvolvimento.

Esse artefato deve ser produzido durante a fase de inicialização, pois os conhecimentos necessários estão bem amadurecidos nessa fase. Sugere-se, no entanto, que a sua construção ocorra após a definição e priorização das *User Stories*. Não é necessário que as *User Stories* estejam divididas em tarefas.

Todos os desenvolvedores devem participar da elaboração deste artefato. A participação do cliente é importante para validar a arquitetura definida.

## Resumo

### O que é?

É um artefato que define em alto nível o domínio e o funcionamento do sistema, mostrando de forma clara as possíveis dependências.

### Para que serve?

- Mostrar a estrutura do sistema;
- Mostrar dependências existentes entre os componentes do sistema;
- Validar junto ao cliente a estrutura do sistema.

### Quem faz?

Os desenvolvedores, a partir de conversas com o cliente e artefatos gerados anteriormente, como por exemplo, o documento de visão.

**Quando deve ser gerado?**

Durante a inicialização, depois que as *User Stories* sejam definidas e priorizadas, mas antes que sejam divididas em tarefas.

**Duração**

Entre 60 e 90 minutos.

**O que o documento deve responder:**

- Como ocorre o funcionamento do sistema?
- Quais os principais componentes do sistema?
- Quais as dependências entre eles?

# 11. Planejamento

Concluídas as atividades de inicialização, deve ser iniciado o planejamento do projeto.

Primeiramente, a equipe deve estar ciente do tempo disponível para o desenvolvimento do projeto, e a partir de então, com o cliente, definir o número de *releases* e iterações necessárias para a conclusão do mesmo.

Tratando-se do escopo de uma disciplina, o YP recomenda 3 *releases* por semestre letivo, cada uma contendo duas iterações de 2 semanas cada – três meses no total.

É interessante gerar uma matriz mapeando as habilidades de cada membro da equipe de desenvolvimento, assim como o número de horas por semana que cada um poderá dedicar ao projeto, pois isso facilitará bastante a alocação das tarefas, atividade descrita mais à frente.

Como as *User Stories* já foram definidas, deve ser iniciado o planejamento de *release*. É interessante observar se *User Stories* que abordem suporte ao usuário estão sendo consideradas. *User Stories* que abordem um guia para o usuário, ajuda on-line, etc, influenciam diretamente no sucesso do projeto.

Inicialmente, as *User Stories* devem ser alocadas nos *releases* e a partir daí nas iterações, levando-se em consideração a priorização sugerida pelo cliente. Porém, a equipe de desenvolvimento deve alertá-lo para as *User Stories* que tecnicamente devem ser primeiramente implementadas. Por exemplo, aconselha-se que *User Stories* de maior risco para o projeto sejam atacadas no início, pois o risco de um projeto ser abortado depois de muito tempo de desenvolvimento é minimizado.

A alocação das *User Stories* deve ser feita apenas para o próximo *release*, pois planejar *releases* à frente significa que mudanças ou aparecimento de novos requisitos podem vir a invalidar um planejamento de atividades ainda distantes. Todas as *User Stories* do *release* devem ser divididas em tarefas.

É relevante observar que, durante a alocação das *User Stories*, o tempo de um *release* ou iteração é fixo. O escopo, ou seja, a quantidade de tarefas e/ou *User Stories* que serão implementadas, é que deve variar.

Para completar o planejamento da iteração, os testes de aceitação para cada *User Story* devem ser definidos pelo cliente. Cada *User Story* deve ter pelo menos um teste de aceitação. O YP sugere fazer uso da tabela de alocação de tarefas (TAT), para listar todas as tarefas da iteração, especificando o tempo de desenvolvimento e o responsável por cada uma delas, levando em consideração as habilidades de cada um e o seu tempo de dedicação ao projeto. Ao especificar o tempo de desenvolvimento das tarefas, talvez seja interessante estimar "por cima", resultando assim num tempo de contingência. É importante perceber que neste momento a TAT não é completamente construída, pois campos como tempo real para conclusão da tarefa e *status* só serão preenchidos no final da iteração.

Um cronograma da iteração, especificando a data de início e término de cada tarefa pode ser construído a fim de melhor gerenciar o andamento das tarefas.

Assim como no planejamento de *release*, o planejamento da próxima iteração só se dá ao término da anterior.

Durante todo o planejamento, riscos devem ser identificados e analisados. Em geral, os riscos são realmente percebidos no momento da implementação. Veja mais sobre análise dos riscos no documento Reunião de Acompanhamento.

Ao elaborar o planejamento das iterações ou dos *releases*, deve-se levar em consideração as tarefas e/ou *User Stories* não concluídas.

Percebe-se então que o planejamento de *release* termina juntamente com o planejamento da última iteração deste *release*.

É extremamente importante que o gerente disponibilize todo o planejamento em um local onde a equipe de desenvolvimento e o cliente possam acessar, como por exemplo, um site.

## Resumo

### O que é?

O planejamento consiste de dois outros planejamentos, o de *release* e o da iteração. Estes descrevem de forma clara e objetiva, para toda a equipe de desenvolvimento, um planejamento das atividades que serão desenvolvidas. Alocam-se as *User Stories* no *release* durante o planejamento de *release*. A alocação das *User Stories* deve ser feita apenas para o próximo *release* e elas devem ser divididas em tarefas.

### Como se faz

- Alocar as *User Stories* nos *releases*;
- Planejamento de *release*;
- Dividir as *User Stories* do *release* em tarefas;
- Alocar as tarefas nas iterações deste *release*;
- Planejamento de Iteração;
- Especificação dos testes de aceitação das *User Stories* com o cliente;
- Especificação do tempo de desenvolvimento e o responsável por cada tarefa.

### Para que serve?

É útil na organização, no planejamento a curto e em longo prazo e no cumprimento das tarefas do projeto. Auxilia o gerente a avaliar o desempenho de toda a equipe de desenvolvimento quanto ao andamento do projeto.

### Quem faz?

O cliente e toda a equipe de desenvolvimento participam do planejamento do *release* e da iteração. Mas é responsabilidade do gerente documentar e disponibilizar de forma clara estes planos.

### Duração

Cerca de 2 horas.

**Obs:** Como o planejamento depende do cliente, e este nem sempre está acessível, o tempo dito acima tende a ser maximizado. Vale a pena investir tempo nesta fase, um bom planejamento reflete num ótimo andamento do projeto.

# 12. Propriedade Coletiva De Código

Propriedade coletiva é a habilidade de qualquer membro da equipe poder alterar o código elaborado por outros membros durante o desenvolvimento. Isto implica dizer que o código é de posse coletiva, ou seja, toda a equipe é responsável por ele.

Esta prática é uma excelente ferramenta para a melhoria contínua de código, além de facilitar a recuperação de falhas, pois possíveis trechos de código problemáticos podem não estar claros para um dos desenvolvedores, mas podem ser facilmente percebidos pelos outros membros da equipe.

Para que estas idéias funcionem bem é imprescindível que haja um controle contínuo e eficiente de versões, para que se controle quem está alterando cada trecho de código em determinado momento, evitando conflitos. Outro pré-requisito fundamental é a certeza de se ter sempre o código testado de uma maneira eficaz, eficiente e funcional, de forma que, ao se alterar o código, a funcionalidade especificada permaneça a mesma. Outras necessidades importantes são a de se ter um código limpo e uma boa e consistente documentação, pois essas características facilitam o entendimento do código e diminuem o tempo gasto para compreensão do deste quando é necessário efetuar alterações.

# 13. Padrões De Codificação

Uma vez que o YP sugere a propriedade coletiva de código, deve-se ter em mente a necessidade da geração de um código limpo (evitar: linhas de código desnecessárias, repetição de trechos de código, implementação de funcionalidades não condizentes com as *User Stories*...) e de fácil entendimento. Com isso, um membro da equipe de desenvolvimento que não tenha codificado aquela parte do código será capaz de entendê-la e modificá-la, sem que seja necessário muito tempo ou esforço nessa tarefa. Sugere-se aplicar ao projeto as práticas de *Design* Simples, Padrões de Codificação, Padrões de Projeto e Refatoramento, de forma que o código gerado possa ser considerado limpo. Segue uma breve descrição de cada uma dessas práticas.

*Design* Simples: Consiste em perseguir a geração do melhor código possível, ou seja, de fácil entendimento, sendo praticamente auto-explicativo, exigindo apenas comentários essenciais. Fazer testes com uma boa cobertura do código é imprescindível. O código deve ter um desempenho aceitável. Com relação à flexibilidade do código, deve-se ter cautela, uma vez que esta prática, quando em excesso, pode levar à produção de código desnecessário (linhas de código que não serão utilizadas durante a execução do sistema). A adição de código baseado na previsão de futuras funcionalidades deixa-o mais complexo. Seja sensato ao definir o nível de flexibilidade do código.

Padrões de codificação: É bastante positivo que, antes de começar a codificar, a equipe de desenvolvimento defina um padrão de codificação. Deve-se determinar, por exemplo, o tamanho da identação utilizada, a forma como devem ser posicionados os parênteses e chaves, a maneira como devem ser nomeados os métodos e variáveis. Tudo isso para que os desenvolvedores não desperdicem tempo "corrigindo" a forma do código gerado por outro membro da equipe, e o entendimento seja o mesmo por toda a equipe.

Padrões de Projeto: O uso de soluções previamente pensadas por grandes projetistas da Orientação a Objetos fornece a possibilidade de reutilizar micro-arquiteturas de classes, objetos, suas funcionalidades e suas colaborações. Ao utilizar essas soluções em diferentes tipos de problemas e situações, se ganha tempo, garantindo eficiência e fazendo com que o produto final aproxime-se de um software de qualidade. Mais detalhes sobre os Padrões de Projeto podem ser encontrados no livro da "Gang of Four" (GoF) [1], ou em
http://www.dsc.ufpb.br/~jacques/cursos/map/html/map2.htm

Refatoramento: Pode ser visto com uma "pequena" modificação no código do sistema que não altera o seu comportamento funcional, mas que melhora algumas qualidades não-funcionais, tais como: simplicidade, flexibilidade, desempenho e clareza do código. Essa é uma prática

bastante eficiente quando o objetivo é a limpeza de código. Porém, para que ocorra com sucesso, é necessário que todo o código esteja funcionando perfeitamente, ou seja, os testes de unidade devem cobrir todo o código, devem estar rodando 100%. Recomenda-se ter um controle de versões (por exemplo, através do CVS), assim como condutas de integração contínua. É aconselhável fazer refatoramento constantemente, o que faz com que o código apresente-se cada vez mais legível. Para tanto, além das exigências anteriores, é preciso que o tempo para efetuá-lo seja estabelecido durante o planejamento da iteração. Existem diversos padrões de refatoramento, alguns deles podem ser encontrados em http://www.refactoring.com/catalog/index.html.

## Resumo

### O que é?

São técnicas e/ou práticas que auxiliam o programador a gerar código limpo e de qualidade. Exemplos: *Design* Simples, Padrões de Codificação, Padrões de Projeto e Refatoramento.

### Para que serve?

Para facilitar o trabalho de uma equipe de desenvolvimento que adota a propriedade coletiva de código, pois o código legível é mais bem manuseado pelos desenvolvedores.

### Quem faz?

Os desenvolvedores e testadores.

### Quando deve ser gerado?

Durante todo o período de implementação e codificação.

### Duração

Não definida.

# 14. Pequenos *Releases*

Normalmente, no desenvolvimento de software, o prazo para finalização do sistema é uma preocupação crítica. Tratando-se do YP, é necessário que o planejamento esteja sempre visando o cumprimento dos requisitos alocados em cada *release*.

O YP propõe que o planejamento seja focado em pequenos *releases,* garantindo uma maior interação com o cliente. As *User Stories* alocadas em cada *release* só podem ser consideradas como finalizadas após a realização dos testes de aceitação. Um benefício desta prática é a redução do impacto das mudanças de requisitos.

Considerando que cada *User Story* alocada seja finalizada após a aceitação do cliente, pode-se considerar que tudo o que esteja implementado funcione satisfazendo a especificação do projeto. Mesmo que haja erros na implementação de algum requisito, o impacto é bem menor do que realizar uma implementação com *release* mais longo.

Um período letivo nos moldes atuais possui quatro meses de duração, o ideal é que cada *release* dure cerca de um mês, contendo duas iterações, cada qual com duas semanas de duração. A cada finalização de iteração sugere-se um novo encontro com o cliente do projeto.

## Resumo

**Pequenos *releases* resultam em:**

- Melhorias na interação com o cliente do projeto;
- Minimização do impacto sobre os erros na implementação;
- Redução do impacto nas mudanças de requisitos observados pelo cliente;
- Melhorias na identificação e superação dos riscos.

**Tempo do *release*:**

Um mês, contendo duas iterações. Cada iteração deve ser conduzida no período de duas semanas.

# 15. Integração Contínua

O YP propõe que a integração dos módulos gerados do sistema seja feita de forma contínua, com o apoio de alguma ferramenta de controle de versão. Na verdade, é interessante que a integração seja feita cada vez que algum código novo seja implementado e esteja pelo menos sem erros de compilação para não comprometer o andamento do projeto.

A integração contínua no YP será um benefício para a equipe de desenvolvimento, sobretudo devido à falta de horários comuns de trabalho entre os membros. A equipe não necessitará de um novo encontro para integrar tudo o que foi implementado separadamente, além de diminuir a quantidade de erros na implementação devido à integração de código.

O gerenciamento do progresso do projeto pelo gerente, através da coleta de métricas com a geração de um *Big Chart*, é feita de forma consistente e simples devido à integração, além de facilitar a análise real do desenvolvimento pela equipe.

# 16. Testes

Testar o código significa verificar se a funcionalidade atende ao que foi especificado. Trata-se de uma análise das pré-condições e pós-condições diante do contexto no qual está inserido o módulo a ser testado. Definir e elaborar testes antes de implementar é a prática correta, pois auxilia a pensar mais no código a ser desenvolvido.

Os testes de unidade validam as menores partes do sistema, ou seja, os métodos, classes ou até trechos de códigos confusos. Os testes de unidade devem ser escritos antes da construção do código, uma conduta que ajuda na solução do problema e melhora a qualidade do código desenvolvido.

Uma prática, antes presente em condutas de inspeção, que prima pela qualidade e capacidade do código, é a revisão de código. A revisão de código pode auxiliar na melhoria da qualidade e da funcionalidade do código através de revisões internas. Tais revisões são realizadas pelos testadores. Cabe ao gerente determinar o momento mais adequado para a equipe efetuar a revisão de código. É importante não acumular código para revisão, pois quando isso ocorre, o tempo de revisão pode ser maior do que o desejado. Recomenda-se que cada testador revise o código gerado por um outro membro da equipe, reforçando a idéia de propriedade coletiva de código.

A necessidade de revisões sucessivas de código pode ser minimizada através da prática de programação em pares, na qual a produção de código é realizada por duas pessoas, lado a lado numa mesma máquina. Uma pessoa programa enquanto a outra pensa se aquela é a melhor maneira de se construir aquele código. Estas duplas devem se alternar após um certo período, geralmente a cada dia, por exemplo. O gerente pode decidir quando e como melhor alternar as duplas a fim de se obter um melhor desempenho da equipe. Observe que as condutas de revisão de código e programação em pares podem ocorrer juntas, porém se a equipe não consegue dispor de horários compatíveis entre os programadores, as revisões de código podem ser planejadas junto com os testes.

Os testes de aceitação contemplam um conjunto de situações definidas pelo cliente sobre como medir o sucesso do projeto. Descrevem cenários que devem ser suportados pelo software. As características destes testes devem ser extraídas do cliente da maneira mais transparente possível, durante uma conversa com o mesmo. A equipe deve deixar o cliente à vontade para definir como quiser os casos de testes mais críticos, além de apresentar sugestões que podem ser aprovadas por ele durante a conversa. Usuários finais podem ser ouvidos para reforçar aspectos levantados pelo cliente.

A partir da conversa, um plano de testes deve ser elaborado pela equipe a fim de documentar todos os testes obtidos com o cliente. Esta tarefa é também suportada por todo um aparato ferramental para que

cada *User Story* tenha seu plano de teste bem definido. O artefato gerado deve conter a sua própria definição, os passos para executar os testes e como serão implantados, incluindo o ambiente a ser testado e a situação de teste nos quais os módulos serão inseridos.

O sucesso funcional do sistema está intimamente ligado à cobertura dos testes, nas suas unidades, nos módulos e no comportamento dos módulos submetidos aos testes de aceitação. Um módulo só deve ser considerado pronto depois de ser testado exaustivamente. Logo, os testes têm importância vital para o funcionamento do sistema. Uma *User Story* só é considerada implementada quando for submetida a todas as baterias de testes, passando com sucesso.

## Resumo

### O que é?

Trata-se de uma análise das pré-condições e pós-condições diante do contexto no qual está o módulo a ser testado. Definir e elaborar testes antes de implementar é a prática correta, pois auxilia a pensar mais no código a ser desenvolvido.

### Para que serve?

- Verificar se a funcionalidade atende ao que foi especificado;
- Documentação do plano de testes:
    - Definir um plano de testes para cada *User Story* no processo;
    - Descrição do teste de aceitação:
        - Definição do teste de aceitação;
        - Passos para executar os testes;
        - Como irão ser feitos os testes.

### Quem faz?

- Desenvolvedores fazem testes de unidade;
- A equipe junto ao cliente deve especificar os testes de aceitação e os testadores implementam estes testes.

### Quando deve ser gerado?

- O plano de testes deve ser gerado a partir da conversa com o cliente. O momento da implementação deve ser decidido no plano de testes;
- Os testes de unidade devem ser gerados antes de codificar o módulo a ser implementado, ou em paralelo com o mesmo.

### Duração

- Testes de unidade devem ser contínuos;

- Nos testes de aceitação a duração é variável de acordo com o escopo do problema.

**Programação em pares**

- Deve-se analisar antes de programar em pares as seguintes variáveis:
  - Escopo;
  - Tempo;
  - Habilidades da equipe;

# 17. Reunião De Acompanhamento

A reunião de acompanhamento deve ocorrer semanalmente, visando avaliar sistematicamente os resultados obtidos no projeto até o momento. A mesma deve ser coordenada pelo gerente, que tem fundamental importância na análise dos resultados obtidos. A dedicação e o compromisso do gerente é uma questão decisiva para o sucesso do projeto. Seu empenho e atenção podem identificar, prematuramente, falhas no andamento do projeto. Quando a descoberta de deficiências é feita antecipadamente, é possível minimizar os custos tanto para a equipe de desenvolvimento quanto para o cliente.

Toda a equipe de desenvolvimento deve participar da reunião, pois as discussões se darão em torno da produção de seus membros, assim como das situações vivenciadas por cada um deles durante a semana. Para facilitar à avaliação, é rigorosamente recomendado fazer uso do *Big Chart* e da Tabela de Alocação de Tarefas (TAT). Uma análise de riscos e possíveis mudanças inerentes ao projeto deve ser levantada durante a reunião, assim como a necessidade explícita de refatoramento de código.

***Big Chart*:** *Ver* tópico *Big Chart*, página 34.

**Tabela de Alocação de Tarefas (TAT):** Quando a reunião coincide com o término de uma iteração, a TAT deve ter seu preenchimento finalizado. Só ao final da iteração é possível preencher os campos de tempo real necessário para realizar cada uma das tarefas e o status referente às mesmas. A TAT começa a ser preenchida no planejamento da iteração. Verifique o tópico referente ao Planejamento, página 21.

**Refatoramento:** Necessidade explícita de refatoramento é identificada pelo desenvolvedor, quando o mesmo está ciente de que o código que gerou precisa passar por uma revisão de refatoramento. Essa necessidade não deve ocorrer com freqüência, pois os programadores devem tentar produzir o seu melhor código. Veja o tópico referente a Padrões de Codificação, página 25.

**Análise de riscos:** Quando riscos são identificados e analisados, deve-se buscar uma solução prudente para que o projeto não venha a falhar. De acordo com o tipo de risco encontrado, novas diretrizes para o projeto devem ser traçadas. Caso essa providência não seja suficiente, o gerente, junto aos desenvolvedores, deve questionar a viabilidade do projeto, podendo este ser abortado após o entendimento e consentimento do cliente.

**Mudança:** A identificação de possíveis mudanças deve ser abordada durante a reunião. Caso seja encontrada alguma, esta deve ser tratada com bastante cuidado. Se a mesma reflete em artefatos já gerados, a

atualização dos mesmos deve ser feita sob orientação do gerente, o qual deve ser cauteloso para que os artefatos continuem consistentes após as alterações.

Existem vários níveis de mudanças: aquelas que exigem apenas modificações de implementação, ou alocação de tarefas; e aquelas que podem mudar o planejamento, o projeto arquitetural, o esquema lógico, a priorização dos *User Stories* e até mesmo o documento de visão. Cada uma delas deve ser tratada com atenção para que a melhor solução seja obtida.

A participação efetiva do gerente é fundamental nessa decisão. Quando as mudanças trazem riscos ou mudanças de requisitos, é possível que a equipe não consiga cumprir o que tinha ficado firmado com o cliente. É aconselhável variar o escopo do prometido e não o tempo necessário para que o mesmo seja efetuado, entretanto a decisão de alteração do escopo deve ser discutida com o cliente antecipadamente. Caso a modificação ocorra no escopo de tempo, é possível que a equipe perca a credibilidade junto ao cliente, pois este quer ver resultados concretos do projeto, mesmo que em menor quantidade do que a presente no acordo.

## Resumo

### O que é?

Reunião semanal que visa recolher e analisar métricas. É aqui que se faz uso do *Big Chart* e da Tabela de Alocação de Tarefas (TAT) para examinar o andamento do projeto. A preocupação com os riscos e possíveis mudanças deve ser pauta da reunião, assim como a necessidade explícita de refatoramento.

### Para que serve?

Para fornecer ao gerente uma visão dos resultados obtidos pela equipe em uma semana de trabalho. A partir da análise feita, o gerente é capaz de identificar os pontos nos quais o projeto está caminhando bem, e aqueles onde existem falhas, podendo assim, buscar soluções para melhorar os resultados futuramente obtidos.

### Quem faz?

A reunião deve ser coordenada pelo gerente de projeto. Todos os desenvolvedores e testadores devem comparecer. A presença de todos é extremamente importante. A dedicação do gerente é decisiva com relação ao sucesso do projeto.

### Duração

Cerca de 1 hora e 30 min.

# 18. Análise De Riscos

Riscos caracterizam situações indesejáveis que podem ocorrer durante o processo, impedindo o fluxo normal do desenvolvimento do software. Técnicas como o uso de padrões de projeto, refatoramento, entre outras, têm o intuito de diminuir os riscos de fracasso em projetos de software. Nesse sentido, a análise sistemática de riscos durante o processo de desenvolvimento permite que a equipe acompanhe o projeto como um todo, aumentando a sua probabilidade de sucesso.

Alguns fatores importantes podem gerar riscos diretamente durante o processo: custo do projeto, prazo determinado para desenvolvimento e o uso de tecnologias desconhecidas por parte da equipe. Estes dois últimos são muito comuns no meio acadêmico, e podem levar um projeto ao fracasso.

A análise de riscos busca, entre outras coisas, avaliar o projeto rotineiramente e aumentar a probabilidade de sucesso do mesmo, a partir da superação dos riscos. Nessa análise, é importante que sejam propostos meios de se resolver os problemas encontrados. É necessário que exista uma metodologia de análise de riscos bem definida durante o processo de desenvolvimento. Sugere-se uma reunião da equipe para levantamento e análise de riscos pelo menos em dois momentos durante cada iteração[1]. Uma lista de riscos deve ser construída com a descrição do risco, a data em que o risco foi identificado, a prioridade[2] de superação, o responsável[3] e a solução encontrada, quando o risco tiver sido superado. Essa memória é particularmente importante quando novos problemas de mesma natureza aparecerem no processo, pois ficará mais fácil eliminá-los a partir da solução apresentada.

A partir dessa tabela de riscos, pode-se identificar quais são os maiores empecilhos encontrados durante o desenvolvimento do projeto. Atacam-se primeiro os riscos de maior prioridade e os que estão a mais tempo no processo. A relação entre prioridade e tempo deve ser a maneira mais natural de se priorizar os riscos.

A existência de uma metodologia de análise de riscos e uma rotina gerencial que privilegie tal prática faz com que os desenvolvedores estejam livres de surpresas indesejáveis quanto ao resultado final do projeto.

---

[1]No caso da proposta do YP, como a iteração é de 2 semanas, a análise é feita semanalmente, durante a reunião de acompanhamento.

[2]Sugere-se que as prioridades sejam classificadas nos níveis Alta, Média e Baixa.

[3]O responsável por um risco não é necessariamente o responsável pelas tarefas que o gerou. Seu papel é o de buscar formas de superar os riscos.

# Resumo

### O que é?

Riscos são situações indesejáveis que ocorrem durante o processo de desenvolvimento. A análise de riscos é uma metodologia proposta para diminuir essas situações através de uma constante verificação do estado do projeto.

### Como se faz?

Semanalmente, o gerente e a equipe de desenvolvimento fazem um levantamento dos riscos do projeto. O gerente aloca para cada risco um responsável por eliminá-lo. É dada uma prioridade para o risco e o tempo que ele está no projeto. Quando for eliminado, a solução proposta é colocada na tabela de riscos para que se mantenha um repositório de soluções.

### Para que serve?

A análise de riscos serve para maximizar regularmente a chance de sucesso do projeto.

### Quem faz?

O gerente conjuntamente com os desenvolvedores.

### Duração

Entre 15 a 20 minutos.

### Tabela proposta:

| Data de identificação | Risco | *Prioridade | Responsável | Solução |
|---|---|---|---|---|
| | | | | |

* Prioridades possíveis: Alta, Média e Baixa

# 19. *Big Chart*

Pode ser visto como uma análise quantitativa do andamento do projeto a partir do recolhimento de métricas. As métricas são definidas pelo gerente de projeto, de acordo com os pontos principais a serem analisados. Sendo assim, pode ser recolhido e avaliado o número de classes existentes, o número de linhas de código, dentre outros. Todos esses dados são exibidos, em forma de tabela ou gráfico, juntamente com os testes de aceitação que estão prontos, os testes de unidade que estão rodando e os módulos do sistema.

Ao analisarmos o *Big Chart*, não devemos esquecer que o YP trabalha com *releases* e iterações de tempo fixo, quatro semanas e duas semanas, respectivamente. Esses são dados fundamentais para a análise. Caso o *Big Chart* não revele mudanças entre uma semana e outra, ou mostre algo inesperado ou mesmo estranho[1], o gerente deve chamar a atenção da equipe, procurando respostas para a "anormalidade" ali presente. O *Big Chart* pode ser considerado um dos instrumentos mais importantes para auxiliar o gerente na análise de resultados. Quando o gerente de projeto possui a percepção aguçada diante das relações existentes entre os dados expostos no *Big Chart*, seu trabalho junto à equipe de desenvolvimento e ao cliente, é qualitativamente satisfatório.

O auxílio de ferramentas para coleta de métricas e geração automática do *Big Chart*[2] pode ser bastante útil.

[1]: Um exemplo de anormalidade que pode ser observada no *Big Chart* é o aumento do número de classes geradas e a permanência do número de classes de testes existentes. Esse fato pode levar o gerente de projeto a perceber que os desenvolvedores podem não estar dando a devida importância à geração dos testes, o que não é um bom sinal.
[2]: Verifique a lista de algumas ferramentas que podem auxiliar o desenvolvimento do projeto em:
http://www.dsc.ufcg.edu.br/~pet/easyprocess/ferramentas.htm

# 20. Exemplos

Com o intuito de explanar de forma mais clara o easYProcess, alguns exemplos, abordando todas as etapas e artefatos do processo, foram elaborados.

O primeiro exemplo diz respeito à aplicação do processo junto a uma equipe com o desenvolvimento de uma aplicação *desktop* real. O segundo representa uma aplicação WEB, desenvolvida pela primeira turma da UFCG a usar o easYProcess.

## Exemplo 1 – Sistema CRM:

O projeto diz respeito a um sistema de CRM - *Customer Relationship Management* – (Gerência de Relacionamento com o Cliente).

A equipe que irá trabalhar no projeto é composta por Milena, Cláudio, Hélio, Antão, entre outros.

O processo começa com a especificação dos papéis junto à equipe de desenvolvimento. Observe a tabela abaixo:

**Especificação de Papéis**

| Equipe | Papéis |
|---|---|
| Milena | Gerente / Desenvolvedor / Testador |
| Cláudio | Desenvolvedor / Testador |
| Hélio | Desenvolvedor / Testador |
| Antão | Cliente |
| Funcionários de uma pequena empresa | Usuários |

Observações:

- Uma tabela como esta deve ser produzida a cada *release*, refletindo o rodízio de papéis dentro da equipe;
- Cliente e Usuários possivelmente não irão mudar.

Logo depois a equipe de desenvolvimento tem a sua primeira conversa com o Cliente. Desta conversa é produzido o Documento de Visão:

**Documento de Visão**

### 1. Descrição do Projeto

**CRM** consiste em a empresa entender o comportamento do cliente. É um desafio que as empresas estão buscando alcançar por meio de uma nova estratégia de relacionamento, por processos otimizados e pelo diferencial humano nos contatos que devem agregar valor para o cliente e para a empresa, todos suportados por tecnologia de informação. Seus objetivos maiores: são conquistar, fidelizar e conhecer melhor o cliente. CRM envolve, portanto, capturar os dados do cliente ao longo de toda a empresa, consolidar todos os dados capturados interna e externamente em um banco de dados central, analisar os dados consolidados, distribuir os resultados desta análise aos vários pontos de contato com o cliente e usar essa informação ao interagir com o cliente através de qualquer ponto de contato com a empresa. A seguir temos uma estória que servirá como uma analogia para facilitar o entendimento do projeto CRM:

*"O Sr. Joaquim queria aumentar as vendas da mercearia e deixar os fregueses contentes. Como o pessoal gostava de comprar fiado, encomendou umas cadernetas, carimbando nas capas a sigla: "CRM - Caderneta de Registro Mensal". Era nelas que passou a controlar as contas dos fregueses. Logo a CRM ficou popular no bairro. "Anota aí na CRM dois quilos de tomate para a patroa", pedia a Josefina. "Minha mãe mandou perguntar quanto vai pagar de CRM este mês", dizia a Silvinha.*

*Mas a caderneta não servia apenas para cobrar os fregueses. Era a sua bola de cristal. Nela o português enxergava muito mais que o total que iria receber no final do mês. Ele identificava ciclos de comportamento do freguês, suas preferências, a associação dos produtos adquiridos e muitas outras coisas. A freguesa comprava sempre tomate? Dá-lhe campanha promovendo o macarrão e o queijo ralado. A última compra foi há muito tempo? Joaquim ligava avisando que a laranja estava em promoção. E o freguês ia buscar somente porque o Joaquim havia ligado. Até o Pepe, do açougue ao lado, criou sua própria caderneta CRM para acompanhar as preferências da freguesia. Logo Joaquim e Pepe traçavam informações de suas CRMs, para ganho mútuo. E o Manoel da padaria acabou aderindo ao sistema, assim como o João do boteco. Cada um passou a ser um agente de uma pequena rede de troca de informações. O Joaquim vendeu carvão e sal grosso para o Dr. Januário? O Pepe era logo avisado e ia preparando a carne que o doutor gostava.*

*O Manoel aumentava a receita do pão e o João do boteco colocava mais cerveja para gelar. Cada comerciante sabia prever a próxima compra, para fazer a próxima oferta e exceder a expectativa dos fregueses. Todos prosperavam e os fregueses estavam contentes."*

Um projeto deste tipo pode conter várias frentes (funcionalidades), tais como:

- Atendimento a Clientes;
- Automação e gerenciamento da força de vendas (permitir que vendedores obtenham acesso a dados dos clientes);
- Desenvolvimento e gerenciamento de campanhas e relacionamento com clientes;

O projeto como um todo deve ser segmentado em projetos menores com uma estratégia de implantação gradativa, constante e, de preferência, rápida, e deve seguir a seguinte ordem: Estratégia + Processos + Tecnologia. Em resumo, CRM trata-se de um sistema para aumentar a interatividade entre o consumidor e a empresa, de maneira que a empresa possa manter o cliente em uma relação lucrativa de longo prazo, em que todos ficam satisfeitos. Não é uma visão focada na tecnologia, embora seja a tecnologia que torne essa abordagem interessante. Ele deve ser utilizado quando se deseja:

- Atrair o cliente certo, ou seja, para:
  - Identificar os clientes mais valiosos da empresa;
  - Identificar o que a empresa irá lucrar com eles no que diz respeito a bens e serviços.
- Aprender a reter os clientes, ou seja, para:
  - Aprender por que os clientes desistem da compra e como trazê-los de volta;
  - Analisar o que os concorrentes estão fazendo para atrair seus clientes mais valiosos;
  - Criar condições para que a gerência da empresa monitore o nível de desistência dos clientes.

## 2. Requisitos Funcionais

1. **Cadastro de Clientes** - O dono do estabelecimento cadastra os clientes.
2. **Registrar Vendas** - O dono do estabelecimento cadastra as vendas efetuadas, associando ao cliente que efetuou a compra.
3. **Consultar Promoções** - O dono do estabelecimento entra com as informações do cliente para verificar em quais as promoções que o cliente está inserido.

4. **Definir Promoção** - O dono do estabelecimento escolhe uma categoria de clientes (Ouro, Prata, Bronze e Lata) e define as regras para promoções dentro de cada categoria.
5. **Cadastro de Produtos** - O dono do estabelecimento cadastra os produtos do estabelecimento.
6. **Cadastro de Funcionários** - O dono do estabelecimento entra com as informações dos funcionários do estabelecimento.
7. **Cadastro de Usuários do Sistema** - O dono do estabelecimento cadastra *login* e senha para cada usuário do sistema.
8. **Criar Eventos** - O dono do estabelecimento entra com as informações de um evento que estará associado a um cliente.
9. **Cadastro de Categorias** - O dono do estabelecimento cadastra informações a respeito das categorias de clientes que ele deseja criar para o seu estabelecimento.
10. **Consultar Eventos** - O dono do estabelecimento escolhe um tipo de evento e o sistema emite uma listagem dos eventos.
11. **Emitir Relatórios** - O dono do estabelecimento escolhe um dos relatórios e o sistema gera o relatório.

### 3. Requisitos não funcionais

1. **Interface** – Gráfica tipo *desktop* de um sistema. O usuário poderá utilizar o mouse e o teclado para interagir com a interface do produto. A interface deverá ser de fácil utilização, com o objetivo de facilitar a vida do dono do estabelecimento.
2. **Volume de Utilização** - Sem restrição;
3. **Segurança** - Haverá necessidade de *login* por parte dos donos do estabelecimento para utilizar o sistema;
4. **Qualidade / Robustez** - A integridade dos dados é garantida pelo SGBD;
5. **Hardware e Software** - Qualquer máquina que tenha acesso à Internet através de *browser* que permitirá a navegação;

### 4. Perfil do usuário

Proprietário de qualquer tipo de estabelecimento comercial, que queira manter um histórico das compras dos clientes. Funcionários dos proprietários devem ser capazes de interagir com a aplicação, dessa forma, o sistema deve ter uma interface para um público que não é usuário experiente de sistemas informatizados.

Terminado o documento de visão, podemos começar a fase de Inicialização:

## Fase de Inicialização

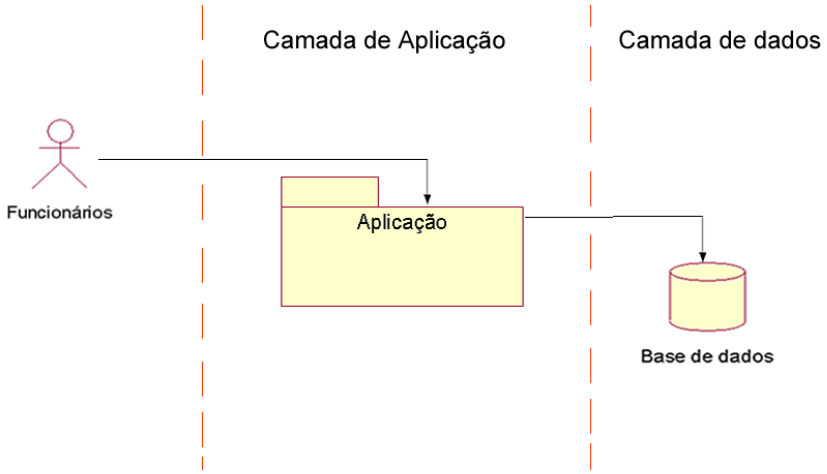
Esta fase consiste de 3 tarefas: definição das *User Stories*, elaboração de um Projeto Arquitetural e do Modelo Lógico de Dados, caso haja um Banco de Dados envolvido.

Veja os exemplos abaixo:

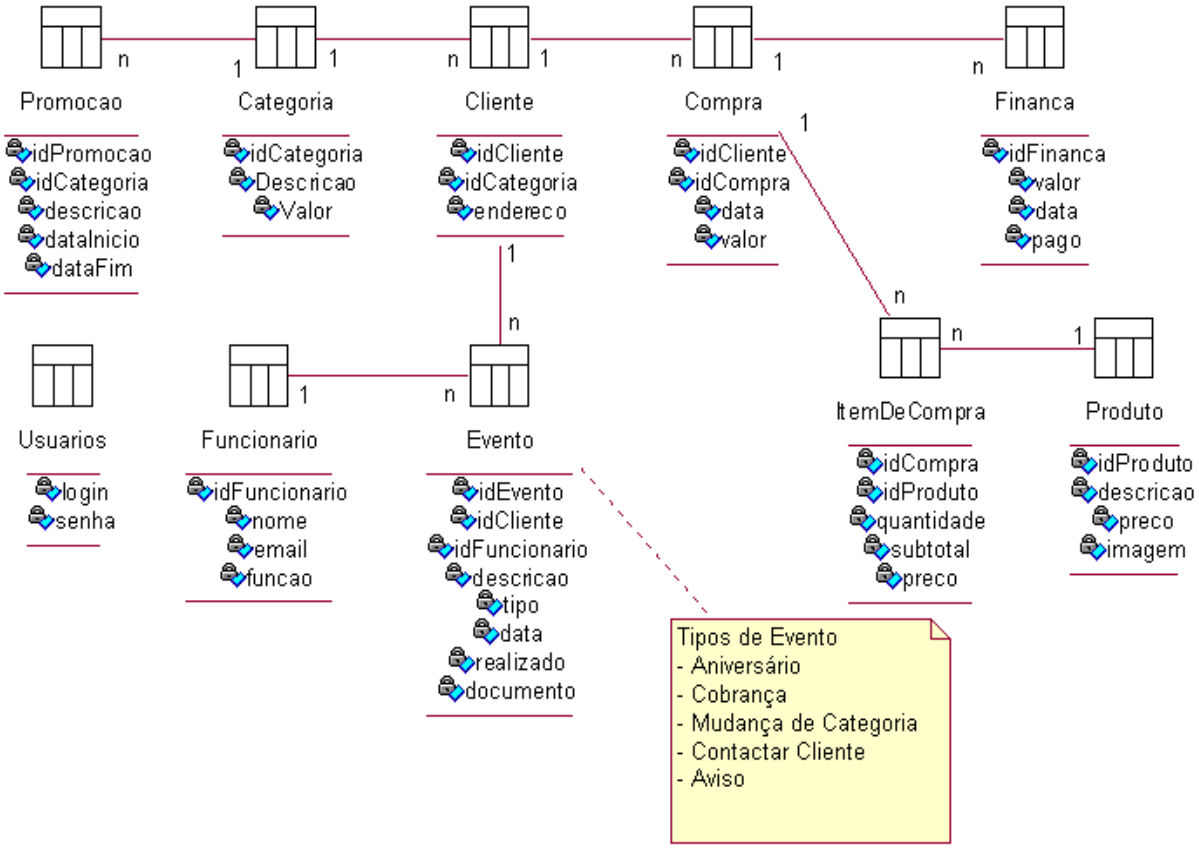
**Lista de *User Stories***

| *User Stories* | Tempo Estimado (Horas) |
|---|---|
| US1 – Cadastro dos clientes, incluindo dados profissionais e pessoais. | 8 |
| US2 – Cadastro dos produtos que serão comercializados, descrevendo as particularidades de cada produto. | 6 |
| US3 – Cadastro de categorias. Estas categorias dizem respeito às possíveis classificações a que um cliente pode pertencer. | 4 |
| US4 – Fazer registro das vendas efetuadas, incluindo informações dos produtos vendidos e do cliente. | 8 |
| US5 – Implementar mecanismo de busca para produtos (por nome, por código, por preço) e clientes (por nome, por categoria, por CPF). | 4 |

**Projeto Arquitetural**



**Modelo Lógico de Dados**




Concluídas as tarefas acima descritas, pode ser iniciada a fase de Planejamento do processo.

Antes de iniciar o planejamento é aconselhável construir uma matriz de competências e dedicação. Logo depois podemos construir os planos de *release* e iteração. É interessante observar que o plano de iteração é composto da TAT (Tabela de Alocação de Tarefas) e da especificação dos Testes de Aceitação.

Veja os exemplos abaixo.

**Matriz de Competências / Dedicação**

| Equipe | Competências | Tempo de Dedicação (Horas/semana) |
|---|---|---|
| Milena | - HTML<br>- JSP/ Servlet<br>- SQL<br>- Oracle<br>- Java | 8 |
| Cláudio | - JSP/ Servlet<br>- SQL<br>- Oracle<br>- Java | 6 |
| Hélio | - HTML<br>- Java<br>- Facilidade de comunicação | 6 |

Observação.:

- Ao final do semestre letivo é interessante que seja gerada uma nova Matriz de Competências para que, comparando-a com a matriz inicial, possa-se saber quais competências foram adquiridas ao longo do desenvolvimento do projeto.

**Plano de *Release***

| ***Release* 1: 05/05 a 30/05** | | |
|---|---|---|
| **Iterações** | **Período** | ***User Stories*** |
| Iteração 1 | 05/05 a 16/05 | US1,US2 |
| Iteração 2 | 19/05 a 30/05 | US3,US4 |

# Plano de Iteração

## TAT – Tabela de alocação de tarefas

| **TAT – Iteração 1** | | | | | |
|---|---|---|---|---|---|
| **Tarefa** | **Descrição** | **Responsável** | **Estimativa de Tempo (Horas)** | **Tempo real (Horas)** | **Status** |
| T1.1 | Formulário de cadastro de cliente | Milena | 1 | | |
| T1.2 | Implementar funcionalidade de cadastro do cliente | Cláudio | 5 | | |
| T1.3 | Tratamento de mensagens de erro | Milena | 2 | | |
| T2.1 | Formulário para cadastro de produto | Hélio | 1 | | |
| T2.2 | Implementar funcionalidade de cadastro do produto | Hélio | 3 | | |
| T2.3 | Tratamento de mensagens de erro | Milena | 2 | | |
| Observe que as tarefas acima, da *User Story* 1 e 2, somam 10 horas e 8 horas respectivamente. Diferente da estimativa feita inicialmente na definição das *User Stories*. Este tipo de erro de estimativa pode ocorrer no começo sendo que, com a prática, os desenvolvedores irão melhorar sua capacidade de estimar o tempo. | | | | | |

Observações:

- A notação T1.2 indica que esta é a tarefa 2 da *User Story* 1.
- Outros exemplos: T1.1 – tarefa 1 da *User Story* 1.
  T2.1 – tarefa 1 da *User Story* 2.
- Possíveis valores para o campo status: C – concluída
  D – em desenvolvimento
  A – abortada

## Definição dos Testes de Aceitação

*User Story* 1
TA1.1 – O preenchimento do formulário de cadastro de um cliente deve ser rápido, com alguns campos com valores padrão.
TA1.2 – A confirmação de cadastro (efetuação do cadastro) não deve demorar mais do que 5 segundos.

*User Story* 2
TA2.1 – O preenchimento do formulário de cadastro de um produto deve ser rápido, com alguns campos com valores padrão.
TA2.2 – O acesso ao cadastro de produtos só deve ser possível depois de uma autenticação.
TA2.3 – A confirmação de cadastro (efetuação do cadastro) não deve demorar mais que 5 segundos.

Durante o andamento das iterações, reuniões de acompanhamento estarão acontecendo a fim de melhor gerenciar o processo e o desenvolvimento do produto.

Nas reuniões de acompanhamento, o *Big Chart* e a Análise de Riscos devem ser abordadas.

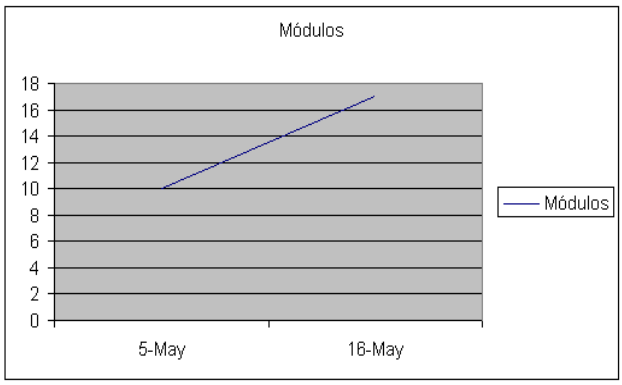
**Tabela de Riscos**

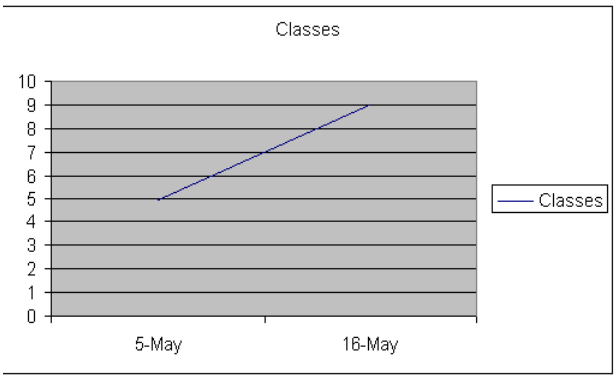
| Data de Identificação | Risco | Prioridade | Responsável | Solução |
|---|---|---|---|---|
| 07/05 | Aprender a manipular o BD Interbase | Alta | Hélio | Estudo dirigido de um tutorial |
| 20/05 | Concluir as tarefas da iteração, devido a não definição dos testes de aceitação junto ao cliente | Alta | Cláudio | Contactar o cliente urgentemente |

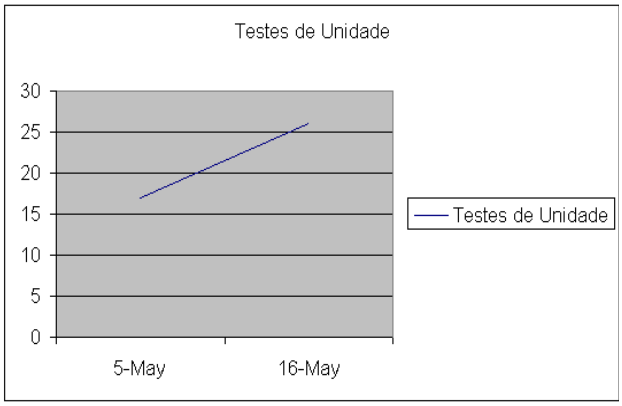
Prioridades possíveis: Alta, média e baixa

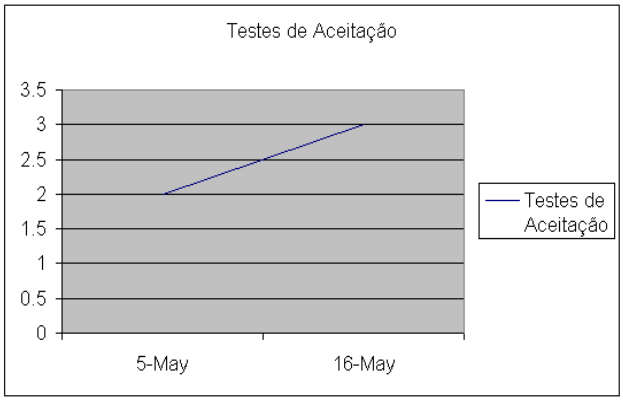
***Big Chart* da 1ª Iteração**

| Data | Módulos | Classes | Testes de Unidade | Testes de Aceitação | Observações |
|---|---|---|---|---|---|
| 09/05 | 10 | 5 | 17 | 2 | |
| 16/05 | 15 | 9 | 32 | 3 | |




É importante lembrar que na reunião de acompanhamento que coincide com o fim da iteração a TAT deve ter seu preenchimento finalizado.

### Fechamento da Iteração 1

TAT – Tabela de alocação de tarefas

| **TAT – Iteração 1** | | | | | |
|---|---|---|---|---|---|
| **Tarefa** | **Descrição** | **Responsável** | **Estimativa de Tempo (Horas)** | **Tempo real (Horas)** | **Status** |
| T1.1 | Formulário de cadastro | Milena | 1 | 1 | C |
| T1.2 | Implementar funcionalidade de cadastro do cliente | Cláudio | 5 | 7 | C |
| T1.3 | Tratamento de mensagens de erro | Milena | 2 | 2 | C |
| T2.1 | Formulário para cadastro | Hélio | 1 | 1 | C |
| T2.2 | Implementar funcionalidade de cadastro do produto | Helio | 3 | 4 | D |
| T2.3 | Tratamento de mensagens de erro | Milena | 2 | 1 | D |

## Exemplo 2 - Projeto XMan

O segundo exemplo de uso do easYProcess aborda um projeto WEB. Trata-se do XMan, projeto desenvolvido pela primeira turma de LES (Laboratório de Engenharia de Software) a usar o YP.

## Definição de papéis

A primeira etapa do processo pode ser executada até mesmo antes de se ter um projeto definido. Trata-se da definição de papéis. Uma vez que se tem uma equipe pronta, deve-se atribuir os papéis de forma que, independente do projeto a ser desenvolvido, cada integrante terá consciência de suas responsabilidades.

A tabela abaixo representa de forma simples e objetiva as pessoas envolvidas no projeto e seus respectivos papéis:

| **Equipe** | **Papéis** |
|---|---|
| Ana Emília Victor Barbosa | Gerente/Desenvolvedor/Testador |
| Ana Isabella Muniz | Desenvolvedor/Testador |
| Cleidson Barreto | Desenvolvedor/Testador |
| Loreno Feitosa Oliveira | Desenvolvedor/Testador |
| Esther, Francilene, Roberta | Cliente |
| Esther, Marcelo, Roberta | Usuários |

**Observação:** Cliente e Usuários dificilmente irão mudar.

Note que uma pessoa pode assumir mais de um papel simultaneamente ou em diferentes momentos. É de fundamental importância que os papéis de cliente e usuário estejam bem definidos e que o gerente seja assumido apenas por um membro da equipe em um determinado tempo.

## Documento de Visão

Realizada a primeira conversa com o cliente, a equipe deve possuir uma idéia bem geral sobre o projeto, o que torna possível elaborar o documento de visão, que pode ser visto a seguir:

### 1. Descrição do Projeto

Desenvolver uma ferramenta para auxílio ao planejamento/gerenciamento de um projeto que utilize XP ou alguma de suas variantes. Esta ferramenta é destinada aos desenvolvedores e gerentes do projeto de software e deve dispor de:

**1.** Funcionalidades para os Desenvolvedores:

- Formulários para inclusão de user stories e tasks - Os campos serão definidos por cada projeto. O software deve ser capaz de fazer o gerenciamento de vários projetos ao mesmo tempo.
- Um repositório de BUGS - Funcionalidades úteis para os desenvolvedores e gerentes.
- Geração de gráficos de acompanhamento (data X %done) a partir das tasks (ou User Story). OBS: Estes gráficos seriam disponibilizados via web e todos os desenvolvedores teriam acesso a estas informações (se estão atrasados, adiantados ou de acordo com o planejado).

**2.** Quantidade de esforço em cada tarefa: quantidade real X quantidade estimada.

O sistema deve suportar os seguintes conceitos de XP:

- Releases, formada por um conjunto de iterações;
- Iteração, formada por um conjunto de User Stories;
- User Story formada por um conjunto de Tasks;
- Task, unidade de trabalho.

Cada um destes conceitos deve conter as seguintes informações específicas:

- Releases: nome, descrição, data de início e data de fim, TAG NAME desta release no CVS. Além disso, é interessante uma tabela contendo todas as iterações que a compõem.
- Iteração: nome, descrição, data início e data de fim, custo em horas estimadas, custo em horas realmente trabalhadas (que corresponde à soma dos custos da User Stories).
- User Story: identificador único, nome, data de criação, responsável, barra de progresso (porcentagem feita), descrição, prioridade, ID da User Story da qual depende o início da User Story em questão, tipo - nova funcionalidade, melhoria em uma funcionalidade existente, teste de aceitação relacionado a ela.
- Task: identificador único, nome, tipo, barra de progresso, estimativa de tempo em horas, quem aceitou a task (dar a possibilidade de cadastrar 2 pessoas neste campo - o pair), natureza da task - adicionada (uma nova tarefa) e planejada (definida desde o início). Uma lista de acompanhamento com os seguintes

campos: data-hora de início do trabalho, data-hora de fim do trabalho, descrição da atividade, autor, par.

**2. Requisitos Funcionais**

**1.** TaskManager:

- Oferecer todas as operações de manutenção para os conceitos que ela reconhece.
- Oferecer um pool de User Stories alocáveis, isto é, que ainda não pertencem a nenhuma iteração, permitindo assim a movimentação de uma User Story de um lugar para outro (de uma iteração para outra).
- Permitir a criação de times de desenvolvimento e a associação destes aos projetos.
- Oferecer controle de acesso para preservar informações do projeto.
- Pode ser integrada a um ambiente que nos forneça integração contínua e testes. Deve fornecer a métrica de cobertura de testes.
- Permitir a geração de métricas próprias do XP como, por exemplo, team velocity (tempo que levou para desenvolver uma User Story).

**2.** BuilDog

- Fazer manutenção no código para retirar bug's.
- Pode ser integrada a um ambiente de gerenciamento de projetos.

**3. Requisitos não funcionais**

**1.** TaskManager:

- Interface Web
- Aceitável que o CVS esteja na mesma máquina que o servidor Web, por causa da ferramenta BuilDog;
- É admissível que se tenha restrições sobre a construção do build.xml;
- Fácil internacionalização com a língua inglesa;
- Conectado ao Banco de Dados MySQL;
- Integração com ferramentas externas.

**2.** BuilDog:

- Interface Web
- Aceitável que o CVS esteja na mesma máquina que o servidor Web, por causa da ferramenta BuilDog;
- É admissível que se tenha restrições sobre a construção do build.xml;
- Integração com ferramentas externas.

**4. Perfil do usuário**

Esta ferramenta é destinada aos desenvolvedores e gerentes do projeto de software usando processos da família XP (YP, XP1,...).

A equipe avaliou as informações contidas no documento de visão e julgou que o desenvolvimento do projeto é viável. Irá aceitar desenvolver o software. O próximo passo agora é mostrar o documento para o cliente.

O cliente poderá aprovar ou não o documento de visão. Aprovar significa concordar que a equipe realmente entendeu a sua idéia e conseguiu traduzi-la neste documento. Dessa forma, o documento de

visão passa a servir como um contrato entre cliente e equipe, servindo como referência nas horas de dúvida.

## Inicialização

A primeira etapa da fase de inicialização consiste em definir as *User Stories*. Se o cliente não estiver presente, a equipe não poderá prosseguir com o projeto. Quem irá definir as *User Stories* além do cliente? Quem pode definir as funcionalidades do sistema na ausência do cliente? Ninguém! A equipe está desenvolvendo software para o cliente. Portanto, só este pode dizer o que o produto deve fazer, até porque o produto será dele!

Ao definir uma *User Story*, o cliente deve garantir que o que foi pedido por ele foi feito realmente como ele esperava. Para isso ele define testes de aceitação, que representam condições para que ele se convença de suas expectativas.

Eis uma lista de *User Stories* que pôde ser identificada neste momento inicial:

| User Stories | Tempo Estimado (Horas) |
|---|---|
| **US1** – Estudar as ferramentas taskManager e XPlanner para verificar qual deverá ser extendida. | 6 |
| **US2** – Implementar as funcionalidades de edição e deleção de tarefas. | 22 |
| **US3** – Implementar os testes das classes do pacote beans existente no TaskManager em paralelo a implementação de novas funcionalidades. | 12 |
| **US4** – Estudar a ferramenta BuilDog, verificando as funcionalidades existentes. | 16 |
| **US11** – Adição de novos conceitos (iteração, release, projeto) no XMan. | 16 |
| **US12** – Implementar as funcionalidades de edição e deleção de User Story. | 10 |
| **US13** – Implementar as funcionalidades de edição e deleção de Iteração. | 10 |
| **US14** – Implementar as funcionalidades de edição e deleção de Release. | 10 |
| **US15** – Implementar as funcionalidades de edição e deleção de Task(com os novos conceitos adicionados | 10 |
| **US16** – Implementar as funcionalidades de edição e deleção de Project | 10 |
| **US17** – Implementar as funcionalidades de edição e deleção de um Usuário | 10 |

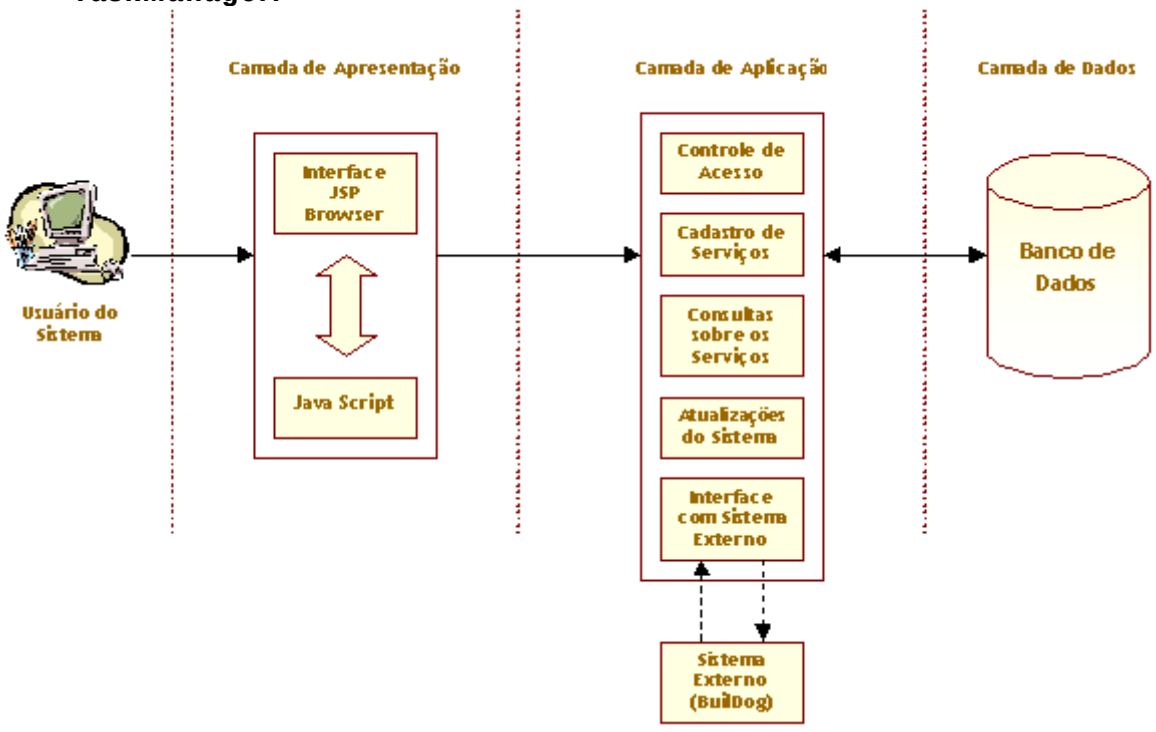
| | |
|---|---|
| **US18** – Reconfigurar o ambiente do administrador | 18 |
| **US19** –Implementar as funcionalidades de poder dar start e stop em uma Task | 14 |
| **US20** – Adição do conceito de time (equipe) no TaskManager, trabalhar em pares em uma Task. | 8 |
| **US21** –Criar uma versão formatada para impressão de um Projeto, de uma Release, de uma Iteração, de uma User Story e de uma Tarefa. | 20 |
| **US22** – Implementar o gerenciamento de User Stories não alocadas. | 6 |
| **US23** – Definir o controle de acesso para os usuários do sistema (visibilidade). | 8 |
| **US24** – Elaboração do Manual do produto. | 2 |

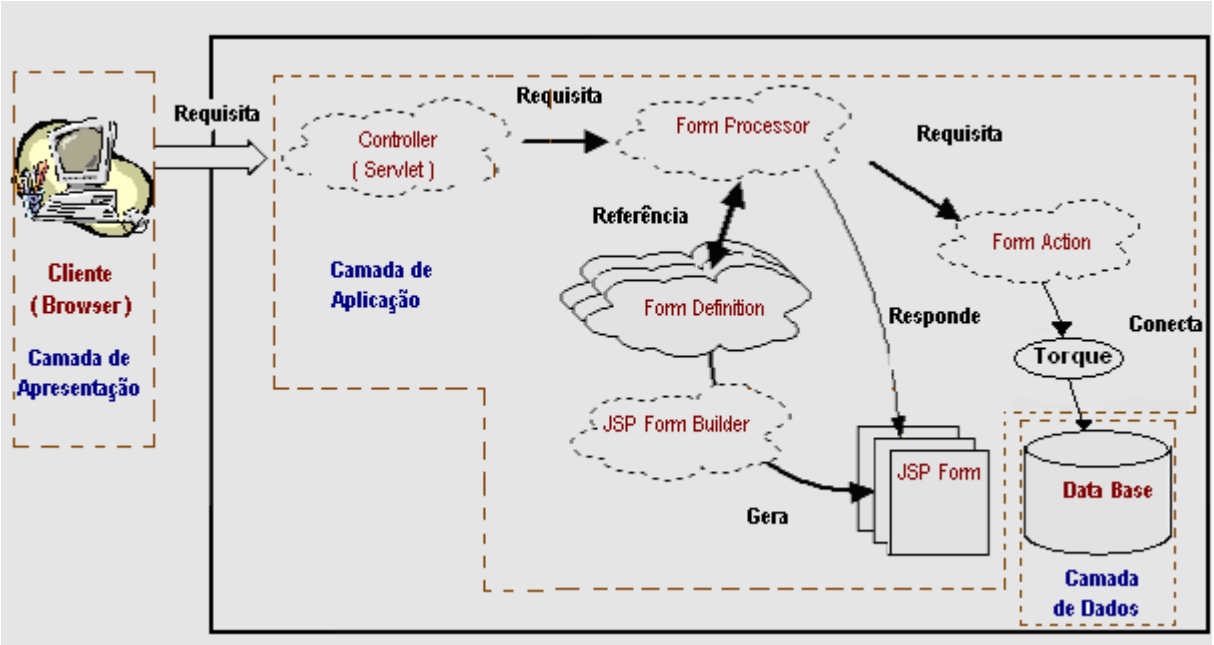
Nota-se que os testes de aceitação não foram definidos ainda. Isto é natural, pois, neste momento, o cliente está preocupado apenas em definir as funcionalidades do produto. Mais tarde, as *User Stories* serão priorizadas e alocadas nas iterações. Ao alocar uma US na próxima iteração (durante o plano de iteração), os testes de aceitação devem ser definidos. Se não forem, a equipe jamais poderá considerar a US como concluída.

Com o tempo, novas *User Stories* podem aparecer e algumas existentes podem ser modificadas ou até mesmo canceladas pelo cliente.

O próximo passo consiste em elaborar o projeto arquitetural, que pode ser visto logo abaixo:

**TaskManager:**






**Organização Geral:**

A arquitetura do sistema está organizada de forma que se tem uma interface com o usuário, a aplicação e uma forma de persistência. Para descrever o sistema é mais adequado uma arquitetura de 3 camadas que provê escalabilidade, facilidade para suporte além de permitir que utilizemos o browser para oferecer uma interface que é integrada ao desktop.

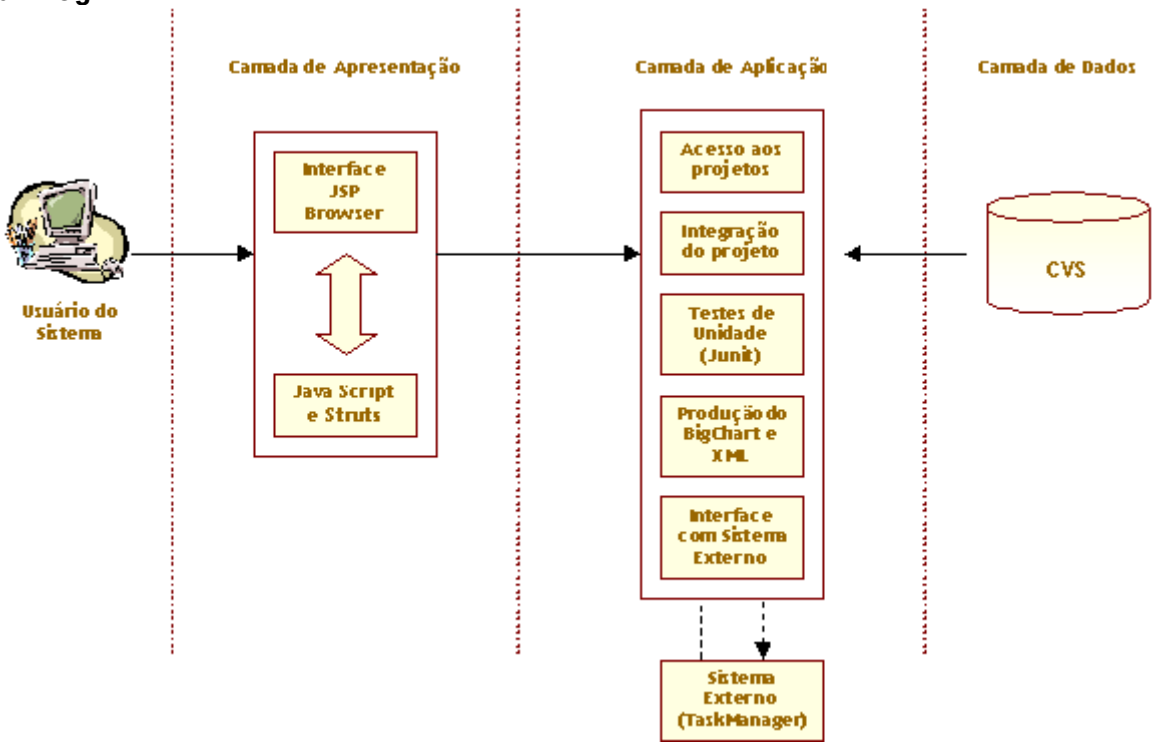
As informações que a estarão na página serão geradas e exibidas de forma dinâmica através da utilização da tecnologia JSP. A lógica da aplicação será construída utilizando a linguagem de programação JAVA, a forma de persistência será o SGBD MySQL que será acessado através da tecnologia JDBC.

A lógica de aplicação do sistema consiste no acompanhamento de todos os membros de uma equipe de desenvolvimento, através da definição de User Stories, estas sendo divididas em tarefas com toda a equipe de desenvolvimento de software, teremos com este sistema uma maneira mais simples de estimar o tempo de desenvolvimento de tarefas junto à equipe, facilitando o trabalho do gerente responsável por designar papéis.

Tipo de Interface: Entre cada módulo existirão interfaces que ajudarão a fazer o desacoplamento entre os módulos uma vez que o acesso ao banco de dados estará disponível apenas através dessas interfaces.

Forma de persistência: Escolhemos como forma de persistência o SGBD MySQL pois além de gratuito ele permite a manipulação/armazenamento de arquivos, inclusive com busca textual.

**BuilDog:**




**Organização Geral:**

A arquitetura do sistema está organizada de forma que se tem uma interface com o usuário, a aplicação e um reservatório (CVS) que será utilizado para capturar os projetos. Para descrever o nosso sistema é mais adequado uma arquitetura de 3 camadas que provê escalabilidade, facilidade para suporte além de permitir que utilizemos o browser para oferecer uma interface que é integrada ao desktop.

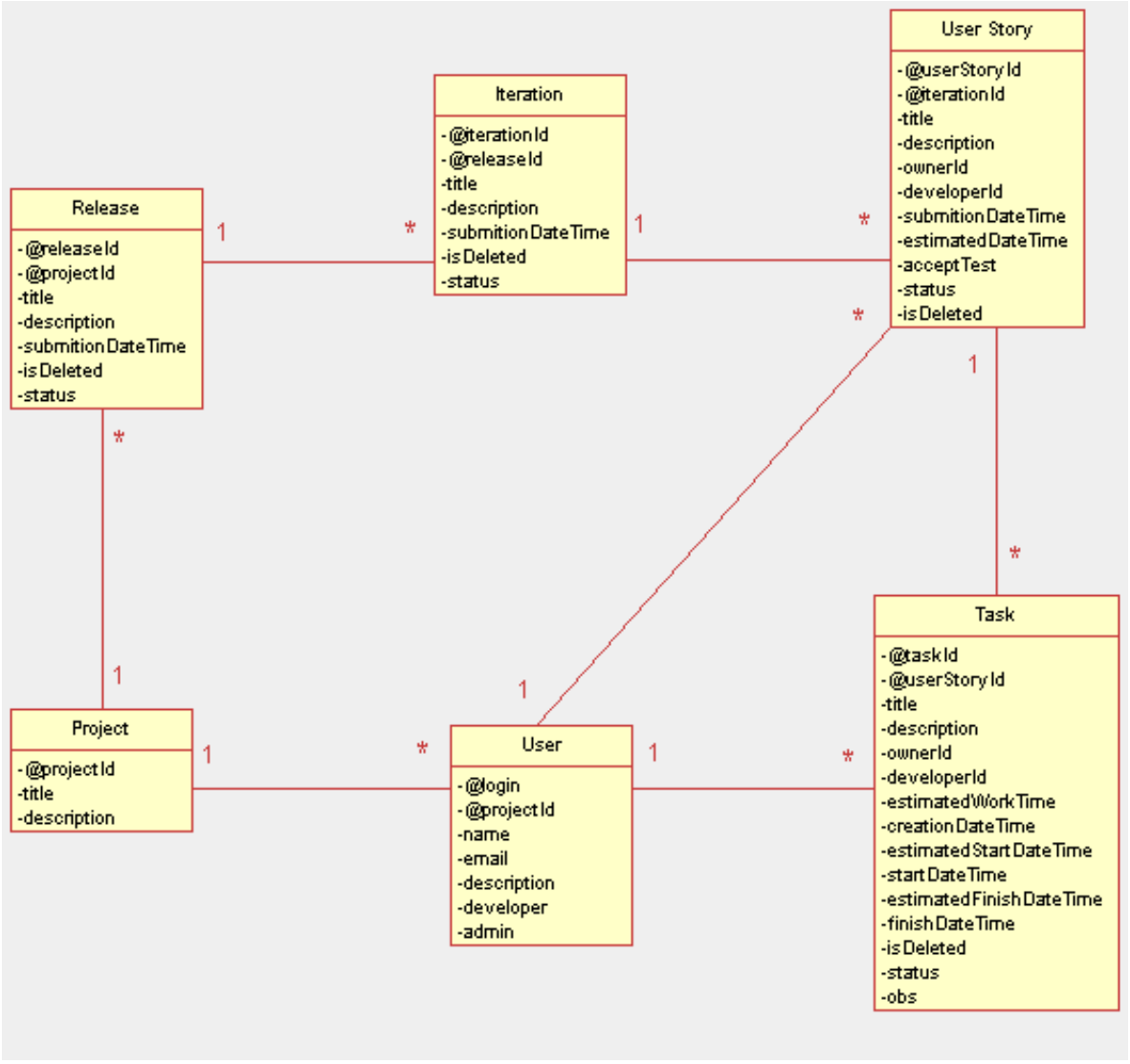
As informações que estarão na página serão geradas e exibidas de forma dinâmica através da utilização da tecnologia JSP e Struts. A lógica da aplicação será construída utilizando a linguagem de programação JAVA e o software auxiliar ANT, o reservatório que conterá os projetos necessários à execução da aplicação é o CVS.

A lógica de aplicação de nosso sistema consiste num software responsável por escutar atualizações de projetos devidamente especificados pelo usuário, contidos num reservatório (CVS), fazer a integração automaticamente da ultima versão deste projeto usando a ferramenta ANT, executar os teste do mesmo e gerar um relatório XML e um Big Chart sobre as execuções anteriores.

Tipo de Interface: Entre cada módulo existirão interfaces que ajudarão a fazer o desacoplamento entre os módulos uma vez que o acesso ao banco de dados estará disponível apenas através dessas interfaces.

É importante confirmar se os requisitos não funcionais estão coerentes com a arquitetura. A arquitetura jamais poderá conter alguma informação contraditória com os requisitos não funcionais.

Vejamos agora como fica o modelo lógico de dados:




Vale lembrar que um modelo bem estruturado evita mudanças, as quais acarretam trabalho demorado.

## Planejamento

A primeira tarefa a se fazer na fase de planejamento deve ser a construção de uma matriz de competências. Tal matriz poderá ajudar a alocação de pessoas para as tarefas. Segue abaixo a matriz gerada pela equipe do projeto XMan:

| Equipe | Competências | Tempo de Dedicação (Horas/semana) |
|---|---|---|
| Ana Emília V. Barbosa | Java - SQL | 10 |
| Ana Isabella Muniz | Java - SQL | 10 |
| Cleidson Barreto | Java - SQL - Linux | 12 |
| Loreno Feitosa | Java - SQL - Linux | 10 |

Pode ser interessante atualizar esta tabela de tempos em tempos, ou no final do projeto. Serve para identificar o aprendizado que o projeto proporcionou aos membros da equipe.

Feito isso, pode-se dar início ao planejamento de *releases*. As tabelas abaixo mostram o resultado final do planejamento de três *releases*.

Release 1:

| Iteração | Período | User Stories |
|---|---|---|
| 01 | 02/06/2003 - 6/06/2003 | US1, US2 |
| 02 | 16/06/2003 - 30/06/2003 | US2, US3, US4 |

Release 2:

| Iteração | Período* | User Stories |
|---|---|---|
| 03 | 01/09/2003 - 22/09/2003 | US4, US5, US6, US7, US8, US9, US10, US11, US12, US13, US14 |
| 04 | 22/09/2003 - 13/10/2003 | US12, US13, US14, US15, US16, US17, US18, US19 |

Release 3:

| Iteração | Período** | User Stories |
|---|---|---|
| 05 | 13/10/2003 - 27/10/2003 | US12, US13, US14, US15, US17, US18, US19, US20, US21, US22, US23, US24 |

***** Em virtude da greve dos professores e funcionários da UFCG, durante o período de 08/07/2003 - 29/08/2003, as datas da 2º e 3º Releases anteriormente previstas, foram alteradas para as novas datas descritas acima.

****** Com a redução do período letivo houve o cancelamento da 6ª iteração, com isto ocorreu o reajuste das datas anteriormente estabelecidas.

O fluxo seguido foi o seguinte: primeiro foi feito o planejamento do primeiro *release*. Logo após foi feito o planejamento da primeira iteração e passou-se pela fase de implementação, repetindo estes passos para cada iteração do release. Só depois disso é que foi feito o planejamento do

segundo *release*. A idéia segue a mesma até que se alcance o planejamento do terceiro *release*, e assim por diante...

Vejamos agora como foi feito o planejamento da primeira iteração:

**TAT – Tabela de Alocação de Tarefas:**

| **TAT – Iteração 1** | | | | | |
|---|---|---|---|---|---|
| **Tarefa** | **Descrição** | **Responsável** | **Estimativa de Tempo (Horas)** | **Tempo Real (Horas)** | **Status** |
| T1.1 | Estudar a ferramenta TaskManager para verificar qual deverá ser estendida. | Ana Emília, Ana Isabella, Loreno e Cleidson | 3 | | |
| T1.2 | Estudar a ferramentas XPlanner para verificar qual deverá ser estendida. | Ana Emília, Ana Isabella, Loreno e Cleidson | 3 | | |
| T2.1 | Documentar o código do TaskManager (superficialmente). | Ana Emília e Ana Isabella | 2 | | |
| T2.2 | Configurar o tomcat. | Loreno | 1,5 | | |
| T2.3 | Configurar o Banco de Dados MySQL | Loreno | 1,5 | | |
| T2.4 | Listar todas as funcionalidades existentes no TaskManager. | Ana Emília e Ana Isabella | 1 | | |
| T2.5 | Estudar (código) para todas as funcionalidades do TaskManager. | Ana Emília, Ana Isabella, Loreno e Cleidson | 2 | | |
| T2.6 | Verificar as funcionalidade que realmente estão funcionando no taskManager. | Ana Emília, Ana Isabella, Loreno e Cleidson | 2 | | |
| T2.7 | Implementar a funcionalidade de edição de tarefas. | Ana Emília, Ana Isabella, Loreno e Cleidson | 3 | | |
| T2.8 | Implementar a funcionalidade de deleção de tarefas. | Ana Emília, Ana Isabella, Loreno e Cleidson | 3 | | |

**Definição dos Testes de Aceitação:**

**User Story 1:**

**TA** - Verificar juntamente com os clientes, como cada ferramenta funciona e as tecnologias empregadas em cada uma.

**User Story 2**

**TA1** – Conferir se a edição de uma tarefa está sendo efetuada com sucesso.
**TA2** – Conferir se a deleção de uma tarefa está sendo efetuada com sucesso.

Com a ajuda da TAT, é possível dividir as *User Stories* em tarefas e alocá-las entre os membros da equipe. Caso não existam testes de aceitação para as US, o cliente terá que defini-los.

O andamento do projeto deve ser acompanhado pelo gerente. Este deve conduzir as reuniões de acompanhamento, coordenando o planejamento de *releases* e iteração. Além disso, é preciso que o mesmo analise o progresso do projeto. Para fazer isso, é preciso que se faça uma análise de riscos. É bastante importante que o gerente disponha de métricas, para que os riscos sejam facilmente identificados e o progresso do projeto seja mais bem analisado.

É, portanto, imprescindível que se mantenha uma lista de riscos e um *Big Chart*, que deve ser gerado pelo menos a cada semana. Eis os exemplos destes artefatos:

**Análise de riscos:**

Primeira semana:

| Data de Identificação | Descrição | Grau | Responsável | Status | Providência/Solução |
|---|---|---|---|---|---|
| 02/06 | Uso da tecnologia desconhecida Tomcat | Médio | Todos | Vigente | Estudar o assunto, considerando o tempo necessário no plano de iteração. |
| 05/06 | Uso do que da ferramenta TaskManager existente | Alto | Todos | Vigente | Estudar o assunto, considerando o tempo necessário no plano de iteração. |

Prioridades possíveis: Alta, média e baixa.

Segunda semana:

| Data de Identificação | Descrição | Grau | Responsável | Status | Providência / Solução |
|---|---|---|---|---|---|
| 02/06 | Uso da tecnologia desconhecida tomcat | Médio | Todos | Superado | Estudar o assunto, considerando o tempo necessário na plano de iteração. |
| 05/06 | Uso da ferramenta taskManager existente | Alto | Todos | Superado | Estudar o assunto, considerando o tempo necessário na plano de iteração. |
| 15/06 | Integração com a ferramenta BuilDog | Médio | Todos | Vigente | Reservar tempo para entendimento do código do BuilDog |

Prioridades possíveis: Alta, média e baixa

Terceira Semana:

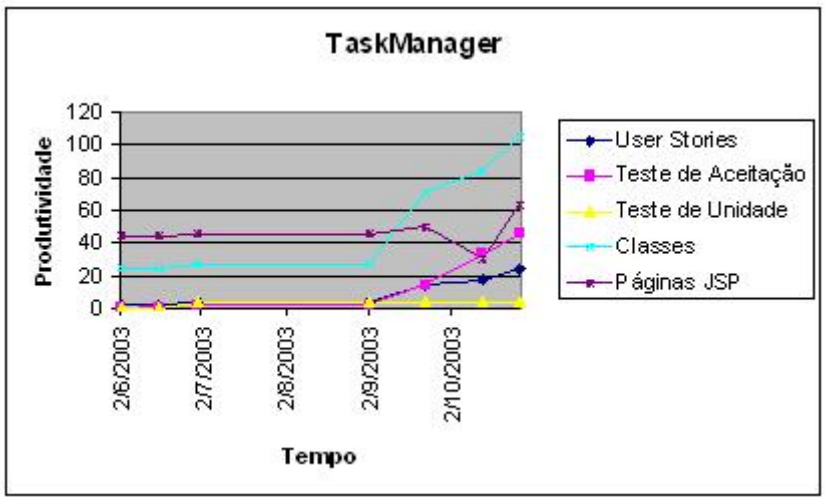
| Data de Identificação | Descrição | Grau | Responsável | Status | Providência / Solução |
|---|---|---|---|---|---|
| 02/06 | Uso da tecnologia desconhecida Tomcat | Médio | Todos | Superado | Estudar o assunto, considerando o tempo necessário na plano de iteração. |
| 05/06 | Uso do que da ferramenta TaskManager existente | Alto | Todos | Superado | Estudar o assunto, considerando o tempo necessário na plano de iteração. |
| 15/06 | Integração com a ferramenta BuilDog | Médio | Todos | Superado | Reservar tempo para entendimento do código do BuilDog |
| 20/06 | Disponibilidade de máquinas no laboratório – PCT | Alto | Todos | Vigente | Utilizar outro laboratório; Utilizar o PCT em horário de pouco movimento |
| 20/06 | Uso da tecnologia desconhecida – Torque | Alto | Todos | Vigente | Estudar o assunto, considerando o tempo necessário no plano de iteração. |
| 25/06 | Redução do período das iterações 05 e 06 devido à greve | Médio | Todos | Vigente | Rever escopo do planejado e conversar com o cliente a respeito. |

Prioridades possíveis: Alta, média, baixa

Para cada risco é escalada uma pessoa responsável por mitigá-lo. Esta pessoa deve definir uma providência a ser tomada. Quando o risco for completamente eliminado, é aconselhado que seja mantido em um histórico, ressaltando qual foi a solução usada para o eliminá-lo.

**Big Chart:**

| | Data/ User Story | User Stories | Teste de Aceitação | Teste de Unidade | Classes | Páginas JSP/ HTML | Observações |
|---|---|---|---|---|---|---|---|
| 1ª Iter | 02/06/2003 | 0 | 0 | 0 | 25 | 44 | Estudo e preparação do ambiente de trabalho. As classes e páginas apresentadas fazem parte do TaskManager. |
| | 16/06/2003 | 2 | 3 | 0 | 25 | 44 | |
| 2ª Iter | 16/06/2003 | 2 | 1 | 0 | 25 | 44 | |
| | 30/06/2003 | 3 | 4 | 4 | 27 | 46 | |
| 3ª Iter | 01/09/2003 | 3 | 4 | 4 | 27 | 46 | |
| | 22/09/2003 | 11 | 15 | 4 | 70 | 50 | |
| 4ª Iter | 23/09/2003 | 11 | 15 | 4 | 70 | 50 | Redução no número de JSP's devido à reformulação. O número de testes continua o mesmo devido a problemas com a ferramenta de geração de testes para JSP. |
| | 13/10/2003 | 18 | 33 | 4 | 84 | 30 | |
| 5ª Iter | 13/10/2003 | 18 | 33 | 4 | 84 | 30 | |
| | 27/10/2003 | 24 | 46 | 4 | 105 | 63 | |




Este é um *Big Chart* referente à quinta iteração. Note que o mesmo é cumulativo, o que permite acompanhar o progresso do projeto desde o início com maior clareza.

# 21. Ferramentas Livres

### Integração Contínua

| Ferramenta | Descrição |
|---|---|
| **Bacon** | Ferramentas para construção automática que têm como principal função o compartilhamento das informações do estado do projeto por parte dos seus integrantes e geração automática do *Big Chart*. Ambas foram desenvolvidas por alunos do curso de Ciência da Computação da UFCG. |
| **Buildog** | |
| **CruiseControl** | Ferramentas com o mesmo objetivo de construção automática. |
| **Gump** | |
| **AntHill OS** | |

### Desenvolvimento

| Ferramenta | Descrição |
|---|---|
| **Ant** | Ferramenta utilizada para a automação de tarefas, tais como compilação de programas ou geração de javadocs. |
| **Eclipse** | Ambiente integrado de desenvolvimento em Java, com suporte a CVS e Ant. |
| **Java** | Linguagem de programação orientada a objetos desenvolvida pela Sun, com extensões para aplicações cliente/servidor e para dispositivos móveis. |

### Testes

| Ferramenta | Descrição |
|---|---|
| **Junit** | Framework para implementação de testes para programas escritos em Java. |
| **Cactus** | Framework para testes de aplicações web. |

### Controle de Versões

| Ferramenta | Descrição |
|---|---|
| **CVS** | Gerencia alterações concorrentes sobre os arquivos de um projeto. |
| **WinCVS** | Cliente CVS com interface gráfica. |

### Ferramentas da Rational

| Ferramenta | Descrição |
|---|---|
| **ClearCase** | Ferramenta para controle de versões dos artefatos produzidos em ferramentas da suíte da Rational. |
| **RequisitePro** | Gerencia os requisitos do projeto e as mudanças ocorridas nestes, de forma a evitar incoerências com os outros artefatos. |

| | |
|---|---|
| **Rose** | Ferramenta CASE para construção de diagramas UML, com grande integração com o RequisitePro. |
| **TestManager** | Ferramenta para o planejamento e a elaboração de testes. |

# 22. Glossário

**[1] Análise de riscos -** Tarefa que permite, depois de identificados, analisar os riscos e buscar uma solução para os mesmos, assim como alocar responsáveis para solucioná-los.

**[2] Arquitetura -** Estrutura e organização lógica de funcionamento de um sistema computacional. A Arquitetura do sistema é a sua organização em grandes módulos.

**[3] Artefato -** Um artefato é um documento gerado durante o processo. Este documento pode ser um arquivo de código-fonte, uma parte da documentação do sistema, um projeto arquitetural, etc. Alguns artefatos são mantidos durante todo o projeto, enquanto outros apenas durante uma certa etapa do projeto.

**[4] *Big Chart* -** Artefato que permite a análise do andamento do projeto a partir do recolhimento de métricas. Deve ser produzido pelo gerente.

**[5] Escopo -** Abrangência do domínio do problema.

**[6] Glossário -** Artefato gerado no intuito de esclarecer termos não familiares à equipe de desenvolvimento ou ao próprio cliente. Possibilita uma melhor comunicação entre os envolvidos no projeto, evitando dúvidas no entendimento de aspectos do problema.

**[7] Iteração -** Período de tempo para implementação de tarefas. Várias Iterações compõem um *release*.

**[8] Modelo Lógico de Dados -** Modelo que representa a estrutura lógica, relativa ao banco de dados de um sistema.

**[9] Mudanças de requisitos -** São mudanças em características do sistema que afetam diretamente o desenvolvimento do mesmo. Podem acontecer devido a modificações de negócio ou por decisão do cliente. É muito importante que a equipe saiba como lidar com esse tipo de mudanças, uma vez que elas podem representar um grande risco para o sucesso do projeto.

**[10] Padrões de projeto -** Descrição das melhores práticas e bons *designs* de programação para prover soluções mais robustas a problemas já conhecidos.

**[11] Papel -** Um papel constitui um conjunto de responsabilidades. Determinam quais serão os comportamentos dos membros da equipe.

**[12] Processo de negócio -** Compreende um conjunto de atividades realizadas na empresa, associadas às informações que manipula, utilizando os recursos (técnicas, métodos, ferramentas, sistemas de informação, recursos financeiros, etc) e a organização da empresa. Normalmente está direcionado a um determinado mercado/cliente, com fornecedores bem definidos.

**[13] Produto -** Objeto concreto obtido ao final do processo de desenvolvimento, representando o que foi proposto pela equipe inicialmente. Inclui aspectos tais como manual de instruções, help on-line e programa de instalação (ou guia de instalação). Deve satisfazer os requisitos propostos pelo cliente e usuário.

**[14] Projeto Arquitetural -** Artefato cujo objetivo é descrever o funcionamento do sistema a ser desenvolvido num nível alto de abstração. Deve explicitar como as partes do sistema interagem entre si.

**[15] Propriedade Coletiva de Código -** É a propriedade de o código não pertencer a um só desenvolvedor, e sim a todos. Através da atividade de revisão de código, idéias dos vários desenvolvedores da equipe estarão no código produzido, reforçando esta prática.

**[16] Refatoramento -** Prática de reestruturar um código que já está funcionando sem alterar funcionalidade.

**[17] *Release* -** Período de tempo para implementação de uma versão funcional do sistema. É dividido em iterações.

**[18] Reunião de acompanhamento -** Reunião semanal que visa recolher e analisar métricas e riscos, bem como o andamento geral do projeto.

**[19] Revisão de código -** Consiste da revisão interna por parte de um testador, do código produzido por um desenvolvedor, desde que testador e desenvolvedor que gerou o código sejam pessoas diferentes. Prima pela qualidade, capacidade e clareza do código.

**[20] Status -** Situação ou estado de alguma tarefa em uma determinada iteração.

**[21] Tarefas -** Unidade atômica de atividades a serem realizadas. *User Stories* são quebradas em tarefas.

**[22] TAT -** Tabela de Alocação de Tarefas. Faz parte do plano de iteração. Lista todas as tarefas da iteração, especificando tempo estimado de desenvolvimento, responsável pela tarefa, tempo real de desenvolvimento e status. Construída no início da iteração e finalizada ao final da mesma.

**[23] Testes -** Verificação automática de código, abordando se a funcionalidade atende ao especificado anteriormente. É uma análise de pré-condições e pós-condições dentro de um determinado ambiente de configuração.

**[24] Testes de aceitação -** Testes de aceitação são testes funcionais executáveis definidos pelo cliente a partir de uma conversa. É testada toda uma funcionalidade definida pelo cliente. No YP deve se ter um ou mais testes de aceitação para cada *User Story*.

**[25] Testes de Unidade -** São testes para classes e métodos individuais. São sempre automatizados. Em um projeto devem sempre estar rodando 100% para todas as classes testadas.

**[26] *User Story* -** Descrição de uma funcionalidade do sistema pelo cliente através de uma ou duas frases. Toda *User Story* deve possuir pelo menos um teste de aceitação, também definido pelo cliente e que determina uma condição para que a mesma seja considerada como concluída. Os testes de aceitação só precisam ser definidos no plano de iteração.

# 23. Referências

**[1]** E. Gamma, R. Helm, R. Johnson y J. Vlissides, "*Design* Patterns Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

**[2]** http://www.refactoring.com/catalog/index.html

**[3]** http://www.agilemodelling.com

**[4]** http://www.extremeprogramming.org

**[5]** http://www.rational.com

**[6]** http://www.therationaledge.com