Alunos: Davi Luciano, Nathan Messias e Yan Augusto

Plano GQM

**Histórico de Revisão**

| **Data** | **Versão** | **Descrição** | **Autor(es)** |
| --- | --- | --- | --- |
| **23/05/2019** | 1.0 | Elaboração do Documento | Davi Luciano |

**GQM (*Goal, Question, Metric*)**

**OBJETIVOS ESTRATÉGICOS**

1 - Qualidade do produto

Dado que o sistema visa informatizar toda a organização de produção das peças de roupas. Serve para mostrar em tempo real o andamento da produção de um modelo dentro da coleção; dividir as tarefas e alocar pessoas para a produção; Mostrar todo histórico por trás de uma peça já feita e peças em andamento. O sistema visa uma melhor usabilidade e manutenibilidade. Desta forma, o objetivo do GQM é:

| **Analisar** | **código** |
| --- | --- |
| **Com o propósito de** | melhorar |
| **Com respeito a** | Manutenibilidade do *software*  Usabilidade do *software* |
| **Sobre o ponto de vista do** | Desenvolvedor  Cliente |
| **No contexto do** | projeto GOF |

**QUESTÕES**

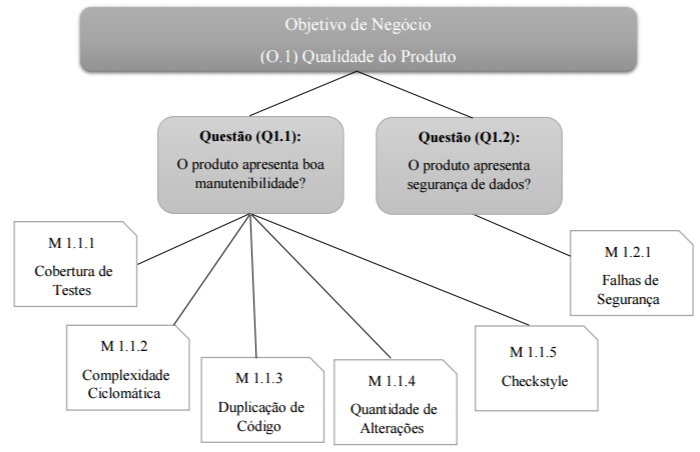
A questão anuncia a necessidade de se obter informações em uma linguagem natural, podendo-se formular uma ou mais questões para cada categoria de questões; quanto à resposta, deve estar de acordo com o objetivo.

O abstraction sheet facilita a comunicação entre a equipe de GQM e a equipe de projeto durante o processo e a revisão do plano GQM representando uma visão simplificada do plano.

|  |  |
| --- | --- |
| **Foco na qualidade**  **- Q.1.1** O produto apresenta uma boa manutenibilidade? | **Fatores de variação**  - A produtividade não atender a expectativa;  - Conhecimento da equipe limitado; |
| **Hipótese de baseline**  - 30% de cobertura de teste até a primeira *release*;  - 90% de cobertura de teste até a segunda *release*; | **Impacto das hipóteses de *baseline*** - Baixa qualidade do produto de \_softwar\_e; - Baixa manutenibilidade. |

**MÉTRICAS**

Essa seção descreve as métricas que serão aplicadas ao projeto como padrão de qualidade.  
Para cada métrica será dada uma descrição do que é a métrica, seus respectivos indicadores interpretativos, os resultados esperados para o projeto e as melhorias propostas para reverter quadros desfavoráveis, isto é, possíveis medidas que a equipe deve tomar para obter índices positivos quando estes estiverem comprometidos.



| **Métrica** | **M.1.1.1 - Cobertura de teste** |
| --- | --- |
| **Objetivo da Medição** | Garantir que o *software* não contenha erros de lógica ou digitação, assim tendo uma garantia de qualidade. |
| **Descrição** | A cobertura de teste é dada pela proporção entre linhas testadas e a quantidade total de linhas de código. A cobertura de código é importante para acompanhar o andamento dos desenvolvimento dos testes. Testes estes que garantem a qualidade e quantidades mínimas de erros de desenvolvimento. |
| **Fórmula** | Cobertura = Linhas testadas / Linhas totais |
| **Escala da Medição** | Racional |
| **Coleta** | Responsável: Equipe de gerência. Periodicidade ou Evento: A cada iteração. Ferramenta: ***SimpleCov*** |
| **Procedimentos** | Será feito o uso da ferramenta no último *commit* para obter os dados. Será mantido junto com as outras métricas em uma tabela para acompanhar o software. |
| **Análise** | Primeira *release*: “Dentro do esperado” dado por CoberturaTeste > 30% “Fora do planejado” dado por CoberturaTeste < 30%  Segunda *release*:  “Dentro do esperado” dado por CoberturaTeste > 90% “Fora do planejado” dado por CoberturaTeste < 90% |
| **Providências** | Caso a métrica esteja abaixo do esperado na primeira *release*, em qualquer uma das iterações que envolvam desenvolvimento, a equipe de gerência deve ser alertada e a coach (Monitora) deve ser procurada em caso de dificuldades. Caso a métrica esteja abaixo do esperado na segunda *release*, na primeira semana a equipe de desenvolvimento deve ser alertada apontada para possíveis materiais de ajuda. Se continuar por uma segunda semana, a equipe de gerência deve interferir, participando do desenvolvimento de testes. |

| **Métrica** | **M.1.1.2 - Complexidade ciclomática (Flog)** |
| --- | --- |
| **Objetivo da Medição** | Verificar a quantidade de caminhos de execução independentes. |
| **Descrição** | Complexidade ciclomática é o número de caminhos independentes dentro do grafo de nós dentro do sistema. Cada nó é um bloco de código sequencial do sistema. De forma resumida e sucinta, complexidade ciclomática equivale ao número de desvios (ou estruturas condicionais) mais 1. Como a coleta consiste em contar o número de condicionais, a métrica também é chamada de complexidade condicional. Ela indica o número de testes que o fragmento de software precisa ter para cobrir todos caminhos linearmente independentes de execução. |
| **Fórmula** | V(G) = e - n + p Onde V(G) é a complexidade ciclomática, n = vértice, e = aresta, p = componentes conectados A média é feita, M(V(G)), dando a complexidade ciclomática média. |
| **Escala da Medição** | Racional |
| **Coleta** | Responsável: Equipe de gerência. Periodicidade ou Evento: A cada iteração.  Ferramenta: ***Rubycritic*** |
| **Procedimentos** | Será feito o uso da ferramenta no último *commit* para obter os dados. Será mantido junto com as outras métricas em uma tabela para acompanhar o \_softwar\_e. |
| **Análise** | De acordo com a ferramenta, baseado em conhecimentos empíricos:  0 a 25 - Nível aceitável  26 a 60 - Refatoração de método  61 e acima - Muito complexa (perigo) |
| **Providências** | Como a métrica de complexidade ciclomática verifica o número de caminhos independentes no código, é necessário diminuir o número de estruturas que causem isso. Loops, Condicionais e recursividades. Caso a métrica esteja acima do esperado, na primeira semana a equipe de desenvolvimento deve ser alertada e apontada para possíveis materiais de ajuda. Se continuar por uma segunda semana, a equipe de gerência de interferir, participando da manutenção do código. |

| **Métrica** | **M.1.1.3 - Duplicações de Código (Flay)** |
| --- | --- |
| **Objetivo da Medição** | Garantir o reaproveitamento de código, reduzindo duplicação de código idêntico ou semelhante. |
| **Descrição** | A análise estática também pode identificar código idêntico e semelhante. A ferramenta procura árvores de sintaxe grandes e idênticas e também usa correspondência fuzzy para detectar código que difere apenas pelos identificadores e constantes específicos usados. |
| **Fórmula** | Quantidade de sintaxes idênticas ou semelhantes presentes no código. |
| **Escala da Medição** | Racional |
| **Coleta** | Responsável: Equipe de gerência. Periodicidade ou Evento: A cada iteração.  Ferramenta: **Rubycritic** |
| **Procedimentos** | Será feito o uso da ferramenta no último commit da iteração para obter os dados. Será mantido junto com as outras métricas em uma tabela para acompanhar a evolução do software. |
| **Análise dos indicadores** | Os resultados devem apresentar índices abaixo de 25, o que indica um código com pouquíssima duplicação de código e potencialmente manutenível. |
| **Providências** | Como a métrica de conexões aferente se refere a acoplamento, então esta deve ser tratada diminuindo o número de classes que conversão com a classe em questão. Talvez isso se dê por uma falta de coesão, ou seja, informações que deveriam estar juntas estão em lugares diferentes. Caso a métrica esteja acima do esperado, na primeira semana a equipe de desenvolvimento deve ser alertada e apontada para possíveis materiais de ajuda. Se continuar por uma segunda semana, a equipe de gerência de interferir, participando da manutenção do código. |

| **Métrica** | **M.1.1.4 - Quantidade de Alterações (Churn)** |
| --- | --- |
| **Objetivo da Medição** | Monitorar alterações de arquivos de origem. |
| **Descrição** | Contagem do número de vezes que uma classe foi modificada no seu histórico de controle de versão. |
| **Fórmula** | Número de alterações no arquivo. |
| **Escala da Medição** | Racional |
| **Coleta** | Responsável: Equipe de gerência. Periodicidade ou Evento: A cada iteração.  Ferramenta: ***Rubycritic*** |
| **Procedimentos** | Será feito o uso da ferramenta no último *commit* para obter os dados. Será mantido junto com as outras métricas numa tabela para acompanhar o *software*. |
| **Análise** | Verificada no indicador I.1 |

| **Métrica** | **M.1.1.5 - Checkstyle** |
| --- | --- |
| **Objetivo da Medição** | Garantir a manutenibilidade do código. |
| **Descrição** | Analisador estático de código para checar se o código fonte está de acordo com as regras de codificação na linguagem especificada. Auxilia nas boas práticas de programação na qual melhora-se a qualidade do código, reusabilidade, clareza, entre outros fatores. |
| **Fórmula** | Não se aplica |
| **Escala da Medição** | Racional |
| **Coleta** | Responsável: Equipe de gerência. Periodicidade ou Evento: A cada iteração. Ferramenta: ***Rubocop*** |
| **Procedimentos** | Será feito o uso da ferramenta no último *commit* para obter os dados. Será mantido junto com as outras métricas numa tabela para acompanhar o *software*. |
| **Análise dos indicadores** | Indicação de trechos não condizentes com a convenção da linguagem. |
| **Providências** | Refatoração do código com o objetivo de seguir as regras e a convenção da linguagem. |

| **Métrica** | **M.1.2.1 - Falhas de Segurança** |
| --- | --- |
| **Objetivo da Medição** | Garantir a segurança do software. Contribuir para redução de vulnerabilidades em sistemas complexos. |
| **Descrição** | Esta medição atuará sobre a vulnerabilidade de código. A ferramenta auxilia a analisar esteticamente o código do aplicativo Rails para encontrar problemas de segurança em qualquer estágio de desenvolvimento. |
| **Fórmula** | Não se aplica |
| **Escala da Medição** | Racional |
| **Coleta** | Responsável: Equipe de gerência. Periodicidade ou Evento: A cada iteração.  Ferramenta: ***Brakeman*** |
| **Procedimentos** | Será feito o uso da ferramenta no último *commit* para obter os dados. Será apresentado um relatório de todos os problemas de segurança encontrados pela ferramenta. |
| **Análise e indicadores** | A ferramenta *Brakeman* oferece as seguintes diretrizes:  **Alto**: Muito provável que a entrada do usuário seja usada de forma insegura.  **Médio**:Uso inseguro de uma variável, podendo ou não ser entrada do usuário.  **Fraca**:Normalmente significa que a entrada do usuário foi indiretamente usada de uma maneira potencialmente insegura.   O considerado ideal para este projeto é o indicador **Alta**. |
| **Providências** | Mitigação de vulnerabilidades. |

**INDICADORES**

| **Indicador** | **I.1 - Turbulência** |
| --- | --- |
| **Objetivo** | Verificar partes de código que necessitam de refatoração. |
| **Descrição** | A relação de *Churn* e Complexidade nos apresenta os arquivos que estão gerando mais dificuldades para o projeto e para os desenvolvedores. |
| **Fórmula** | *Churn* x Complexidade |
| **Escala da Medição** | Racional |
| **Coleta** | Responsável: Equipe de gerência. Periodicidade ou Evento: A cada iteração.  Ferramenta: *\*Rubycritic* |
| **Procedimentos** | Será feito o uso da ferramenta no último *commit* para obter os dados. Será apresentado um relatório de todos os problemas de segurança encontrados pela ferramenta. |
| **Análise e indicadores** | A ferramenta oferece um gráfico, e a partir da análise dessa figura, utiliza-se as seguintes diretrizes:  **Superior direito** - Classes com alta complexidade e alto *churn*. Essas são boas prioridades para a refatoração pois seus problemas de manutenção estão afetando os desenvolvedores em uma base regular. **Superior esquerdo** - Classes com alta complexidade e baixo *churn*. Possui menor prioridade de refatoração, porém possui código com necessidade de depuração. **Inferior esquerdo** - Classes com baixo *churn* e baixa complexidade. O melhor tipo de resultado a ter. A maior parte do código deve estar neste quadrante. **Inferior direito** - Classes com baixa complexidade e alto *churn*. Pode ser justificada por refatorações constantes, não há muito o que refatorar neste caso. |
| **Providências** | Refatoração do código. |