Associação Propagadora Esdeva
Centro de Ensino Superior de Juiz de Fora – CES/JF
Curso de Engenharia de Software
Trabalho de Conclusão de Iniciação Científica – Artigo

Mineração de Repositório de Software:
Investigando a qualidade do software em projetos de código aberto

Bianca Morais Souza[1], André Luís C. Junqueira[1], Gabriel Ribeiro Testoni[1], Arthur Terra[1] e Tássio Martins Ferenzini Sirqueira[2]

[1]Bacharelado em Engenharia de Software – Centro de Ensino Superior de Juiz de Fora
Caixa Postal 36.100.000 – Juiz de Fora – MG – Brasil

[2]Centro de Ensino Superior de Juiz de Fora – Caixa Postal 36.100.000 – Juiz de Fora – MG – Brasil

{andreluis.junqueira,bibimoraiss, gabrielrtestoni}@gmail.com,
tassio@tassio.eti.br

Linha de Pesquisa: Engenharia de Software

RESUMO

Este artigo tem como principal finalidade investigar a qualidade de software em projetos de código fonte aberto, levando em consideração os padrões de qualidade e evolução impostos por Lehman e Bellady. Focado em analisar, através de métricas de software, fatores como complexidade ciclomática, número de classes, métodos e a quantidade de classes. Nesse trabalho foi verificado as principais diferenças entre 3 *releases* do *Kernel* do GNU/Linux.

Palavras-chave: Mineração de Repositórios de Software, Leis de Evolução de Software, Qualidade de Software.

1. INTRODUÇÃO

A constante mudança tecnológica gerou um efeito dramático na construção e no processo de desenvolvimento de software. Em poucas décadas, os recursos de software e hardware aumentaram exponencialmente e tornaram possíveis que cada vez mais produtos complexos fossem criados. Segundo Dijkstra (1972):

> "A maior causa da crise do software é que as máquinas tornaram-se em várias ordens de magnitude mais potentes! Em termos diretos, enquanto não havia máquinas, programar não era um problema; quando tivemos computadores fracos, isso se tornou um problema pequeno e agora que temos computadores potentes, programar tornou-se um problema gigantesco."

Dada a necessidade de aprimoramento para construir e desenvolver softwares, um desenvolvedor está fadado a garantir a qualidade do mesmo, tanto em fatores externos quanto internos. No entanto, mais complexo que mensurar a qualidade de um software construído, é como mensurar a qualidade de um software que além de construído, já está em uso e necessita de constantes mudanças para satisfazer aos seus usuários?

A utilização de métricas é fortemente utilizada para avaliar a qualidade do software, para Sommerville (2011), o objetivo a longo prazo de medição de software é usá-la no lugar de revisões para fazer julgamentos sobre a qualidade de software. Usando a medição de software, um sistema poderia, idealmente, ser avaliado usando uma variedade de métricas e, a partir dessa medição, deduzir um valor para a qualidade do sistema. Quando apropriado, as ferramentas de medição também podem realçar áreas do software que poderiam ser melhoradas.

Atualmente a área de medição de softwares na indústria vem se tornando cada vez mais essencial, e a comunidade cientifica ainda carece de informações sobre os usos atuais de medição da qualidade de projetos de código fonte aberto. Portanto, o objetivo deste artigo é investigar a qualidade de software em tais projetos, utilizando métricas e ferramentas para avaliar as releases geradas por softwares como o *Notepad*++, *Jenkins*, *Code::Blocks*, *ElasticSearch* e o *Kernel* do GNU/Linux.

Observando-se os atributos internos dos mesmos, como a complexidade ciclomática de um componente, profundidade de aninhamento condicional, comprimento de código, entre outros medidos com o uso de ferramentas que analisam o código-fonte do software, pretende-se verificar, além do já supracitado, se as leis de evolução de software propostas do Lehman estão ou não ainda capazes de quantificar a qualidade de um processo de desenvolvimento de software até os dias atuais.

2. OBJETIVOS

Depois de implementados, os softwares necessitam constantemente de mudanças para permanecerem úteis aos seus usuários (SIRQUEIRA, 2016). Tais mudanças, requerem atualizações constantes e continuo acompanhamento, identificando as fases de evolução do software e os problemas envolvidos, durante o desenvolvimento é de suma importância para prolongar o ciclo de vida do produto e, consequentemente, adiar o seu fim.

Atualmente o processo de desenvolvimento de software amadureceu e se tornou uma tarefa complexa e detalhista. Segundo Sommerville (2013), o processo de desenvolvimento de software se baseia em um conjunto de atividades relacionadas que levam a produção de um produto de software.

Tal conceito demonstra em como o processo de desenvolvimento de software amadureceu sob a perspectiva do desenvolvedor. Para Pressman (2006) no início da era da computação softwares não eram produzidos em grande escala, dessa forma o controle de sua produção era mais simples de ser feito, não existiam métodos para controlar o desenvolvimento e nem equipes para realizar um controle da produção. Não existiam muitos métodos

sistemáticos para realizar a programação, sua utilização ficava para segundo plano.

O desenvolvimento de software era feito sem administração, ou seja, sem planejamento. A necessidade de realizar modificações no software se torna cada vez mais constantes à medida que surgem novos objetivos e estes se modificam. Devido a grandes mudanças em sistemas de software, é necessário compreender como os sistemas sofrem mudanças, assim é mais fácil acompanhar sua manutenção evolutiva. Tendo o conhecimento de evolução, os riscos relacionados à perda de qualidade do software são menores.

Para Araújo e Travassos (2008), os programas, assim como as pessoas envelhecem. Não é possível deter o envelhecimento do software, porém é possível entender suas causas e assim tomar atitudes para evitar seus efeitos, reverter danos por algum tempo e assim estar preparado para o dia em que o software não será mais viável. Neste contexto, o principal objetivo deste artigo é mensurar a qualidade do software através de métricas e compara-las com a veracidade das Leis de Lehman nos dias atuais.

Definimos a ênfase da pesquisa em um projeto de código aberto, a saber, o *Kernel* do GNU/Linux. Leva-se como base as *releases* geradas em casa versão estável do software, visando demonstrar como a qualidade do software se torna vem sendo controlada ao longo de seus processos de manutenção e evolução.

3. PROBLEMA

Conforme a ABNT (2003), a qualidade de software é definida como a totalidade de características de uma entidade que afetam a capacidade de satisfazer necessidades explícitas e implícitas, quando usada sob condições específicas. Nesta mesma perspectiva, a qualidade de um software em uso, é considerada como a visão da qualidade de um sistema contendo software, sob a ótica de seu usuário final, que consiste na capacidade do software em permitir que a comunidade que o utiliza atinja metas específicas como eficácia, segurança, produtividade e satisfação. No entanto, a qualidade de um software em uso é mensurada em relação aos resultados de seu uso, e não das propriedades que compõem o próprio software.

A maior parte dos projetos de programação e desenvolvimento de software realizado em disciplinas de graduação, partem do zero, ignorando o fato de que boa parte do trabalho realizado pelos profissionais de software será sobre um software pré-existente, já em uso. Em consequência, o ensino de qualidade e manutenção de software se torna uma tarefa complexa, pois o processo torna-se mais custoso à medida que o software evolui em tamanho, complexidade e amadurecimento.

Ao longo dos anos, as métricas têm se mostrado efetivas em controlar a complexidade e, portanto, em garantir a qualidade do produto. Os dados históricos de um software são atualmente armazenados em repositórios de código fonte, tais como o GIT, e podem ser avaliados para identificar padrões e regras que influenciem na melhora de sua qualidade. A seleção correta das métricas que envolvem o software dependem exclusivamente das regras de negócio, suas necessidades e objetivos. Se usadas de forma correta, as

métricas podem quantificar o sucesso ou o fracasso, e podem auxiliar na tomada de decisões a respeito do processo de desenvolvimento. A partir do *Kernel* do GNU/Linux, a proposta é buscar identificar características que demonstram na evolução do software e seu grau de qualidade, através da mineração das informações do repositório, tendo como base os *releases* gerados.

4. QUALIDADE DE SOFTWARE

Para Sommerville (2011), os termos “garantia de qualidade” e “controle de qualidade” são amplamente usados na indústria manufatureira. A garantia de qualidade (QA, do inglês *quality assurance*) é a definição de processos e padrões que devem conduzir a produtos de alta qualidade e a introdução de processos de qualidade na fabricação. O controle de qualidade é a aplicação desses processos de qualidade visando eliminar os produtos que não atingiram o nível de qualidade.

No entanto, para a Engenharia de Software, a qualidade pode ser definida por um ponto de vista de cada indivíduo, com base em o que é esperado que ele faça, e quais as métricas que definem sua qualidade ou sua impropriedade de uso.

Traçando um paralelo, para um cliente de padaria, a qualidade do pão pode ser definida pelo seu sabor, textura da casca, entre outros. Para um padeiro, há pontos que não são vistos pelos clientes, como a fermentação, a textura da farinha, o tamanho, a espessura do miolo, entre outras variáveis determinantes. O mesmo acontece para um software, de acordo com Engholm Júnior (2010), qualidade de software está relacionada a entregar ao cliente um produto que satisfaça suas expectativas segundo os requisitos do projeto acordado inicialmente.

A qualidade necessita, além de estar em conformidade cm os requisitos funcionais acordados inicialmente, também corresponder aos requisitos não funcionais. Um software lento não irá satisfazer positivamente um usuário. Em consequência desta premissa, Boehm *et al.* (1978) sugeriram que haviam quinze atributos importantes de qualidade de software, como mostrado na Tabela 1. Esses atributos estão relacionados com a confiança, a usabilidade, a eficiência e a manutenibilidade de software.

Tabela 1 – Atributos importantes para um software.

Segurança	Compreensibilidade	Portabilidade
Proteção	Testabilidade	Usabilidade
Confiabilidade	Adaptabilidade	Reusabilidade
Resiliência	Modularidade	Eficiência
Robustez	Complexidade	Capacidade de Aprendizado

Muitas vezes atributos como confiabilidade são considerados os atributos de qualidade de importância mais prezada em um sistema, no

entanto, o desempenho também é importante, a fim de determinar o sucesso e ou fracasso da utilização do mesmo.

Portanto, a qualidade pode ser definida como uma das áreas de conhecimento da engenharia de software que tem como principal objetivo garantir a qualidade do software por meio da definição de processos de desenvolvimento. No entanto, há muitas maneiras de se definir a qualidade de um software. Há muitos métodos e ferramentas de engenharia de software que tem a finalidade de garantir, ou ao menos amenizar a obtenção da qualidade nos programas.

4.1. MÉTRICAS, MEDIDAS E MEDIÇÕES DE SOFTWARE

Pressman (2011) afirma que a medição nos permite obter entendimento do processo e do projeto de software, fornecendo um mecanismo para avaliação objetiva. Ainda segundo Pressman (2011), as métricas são medidas quantitativas que permitem aos engenheiros de software ter ideia da eficácia do processo de software e dos projetos que são produzidos. Ou seja, permitir a coleta de dados básicos de qualidade e de produtividade e utilizar os mesmos como uma ferramenta de gestão.

As métricas também podem ser utilizadas para detecção de áreas com problemas, de modo que soluções ou melhorias possam ser desenvolvidas e assim, consequentemente, o processo de software possa ser melhorado. É comum que os termos métricas, medidas e medições sejam confundidos como sinônimos. No entanto, existem diferenças mesmo que sutis entre eles, que segundo Pressman (2011) são eles:

1. Métrica: Representa uma medida quantitativa do grau que um sistema, componente ou processo possui um determinado atributo.
2. Medida: Fornece uma indicação quantitativa da extensão, quantidade, dimensão, capacidade ou tamanho de algum atributo de um processo ou produto;
3. Medição: Ato de determinar alguma medida.

Complementado a Pressman, Sommerville (2011) define os principais objetivos de uma medição em relação a uma métrica de software, onde a medição de software preocupa-se com a derivação de um valor numérico ou o perfil para um atributo de um componente de software, sistema ou processo.

Como resultado, é possível basear-se e retirar conclusões acerca da qualidade do software, ou avaliar a eficácia dos métodos, ferramentas e processos utilizados dentro da organização. A longo prazo, o principal objetivo de uma medição de software é utiliza-la para realizar julgamentos sobre a qualidade do software desenvolvido. Para isso, o sistema poderia ser avaliado com base em métricas e, a partir desta medição, determinar um valor para a qualidade do sistema.

Sendo possível, por meio das medições, mensurar e comparar numericamente propriedades abstratas como, por exemplo, acoplamento e coesão das classes de um sistema. Um exemplo de métrica inclui o tamanho de um produto em linhas de código, o índice Fog (GUNNING, 1952), que é uma

medida da legibilidade de uma passagem de texto escrito; o número de defeitos relatados em um produto de software entregue, e o número de pessoas/dia requerido para desenvolver um componente de sistema.

Portanto, um dos focos do levantamento das medidas, medições e métricas é possibilitar que os engenheiros e os gerentes de produtos reúnam um grupo de informação que sejam capazes de controlar, estimar, gerenciar e auxiliar na melhoria do projeto, proporcionando aos mesmos uma maior eficácia. Segundo Côrtes e Chiossi (2001) quando são calculadas métricas, pretende-se obter dados que irão proporcionar opções para uma melhoria.

4.2.1. ORIGEM DAS MÉTRICAS, MEDIDAS E MEDIÇÕES

Na década de 1970, fez-se necessário a aplicação de cálculos a fim de quantificar indicadores sobre os processos de desenvolvimento de software. Moller (1993) indicou quatro tendências deste tipo de tecnologia, sendo elas:

1. Medida de Complexidade de Código: Desenvolvido na década de 70, onde as métricas de códigos eram simples de se obter desde que seu cálculo fosse realizado pelo próprio código de forma automatizada.
2. Estimativa de Custo: Também desenvolvida na década de 70, onde estima-se o trabalho e o tempo gasto para se desenvolver um software, considerando fatores como quantidade de linhas de código necessárias para a implementação do sistema.
3. Garantia da Qualidade do Software: Está técnica foi aprimorada entre as décadas de 70 e 80, onde se procura dar ênfase nas informações faltantes durante o ciclo de vida do software.
4. Processo de Desenvolvimento de Software: Este último tornou-se grande e mais complexo com o decorrer do tempo, levando em consideração que inicialmente a necessidade de controlar os processos era emergencial. Procura dar ênfase no gerenciamento de controle e de recursos.

Os desenvolvedores, a partir do aparecimento destas tendências propostas por Moller em 1993, iniciaram a utilização das métricas e medições com o propósito de melhorar não só o processo de desenvolvimento de software, mas também como método de auxílio de tomadas de decisões no âmbito organizacional.

4.2.2. IMPORTÂNCIA DAS MÉTRICAS, MEDIDAS E MEDIÇÕES

De acordo com Demarco (1989) não se pode controlar o que não se pode medir. E complementando a tal premissa, Jacobson (1992) define que um método necessário para controlar o desenvolvimento de um software, bem como sua evolução é utilizando as métricas. As métricas podem mensurar etapa a etapa do desenvolvimento e definir importantes e variados aspectos do produto a ser desenvolvido, sendo um deles, o sucesso e/ou fracasso do desenvolvimento do projeto de software.

As medidas são de suma importância para analisar a qualidade e a produtividade de todo o processo de desenvolvimento e manutenção, bem

como de um produto de software já construído e em constante evolução. Segundo Pressman (2011), o software é medido por vários motivos, sendo eles:

1. Indicar a qualidade do produto;
2. Avaliar a produtividade das pessoas que produzem o produto;
3. Avaliar os benefícios em termos de produtividade e qualidade derivados de novos métodos e ferramentas de software;
4. Formar uma linha básica de estimativas;
5. Ajudar a justificar os pedidos de novas ferramentas ou até mesmo de treinamentos adicionais.

A correta mensuração do software pode auxiliar em tomadas de decisão importantes no decorrer do processo de desenvolvimento do projeto, bem como na evolução do software. Essa premissa vem sendo afirmada e reafirmada desde a década de 90 até a atualidade.

Conforme Fernandes (1995), a gestão de projetos e de produtos de software somente atinge determinado nível de eficácia e exatidão se houver métricas e medidas que possibilitem gerenciar através de fatos e, somente então, será possível gerenciar os aspectos econômicos do software, que geralmente são negligenciados em organizações de desenvolvimento. Segundo Cardoso (1999), a realidade da informática atual aponta para o sério problema da má qualidade nos produtos desenvolvidos.

A única preocupação das equipes de desenvolvimento seria o cumprimento dos prazos. Com esta atitude, é entregue ao cliente apenas as funções básicas do seu sistema, com os prováveis defeitos, que após seriam novamente avaliados pelos desenvolvedores e muitas vezes o cliente teria que pagar adicionalmente um produto que já havia sido concluído.

Já conforme Kruchten et al. (2012), os desenvolvedores de software e gerentes de empresas frequentemente discordam sobre decisões importantes de como investir os escassos recursos em projetos de desenvolvimento, especialmente quanto aos aspectos internos de qualidade, os quais são cruciais para a sustentabilidade do produto, mas invisíveis para a gestão e os clientes, além de não gerarem receita a curto prazo. Esses aspectos incluem o código, a qualidade do projeto e a documentação.

Portanto, considera-se que se os conceitos de engenharia de software para o desenvolvimento do produto fossem devidamente adotados, tornaria-se possível medir a qualidade do processo e/ou projeto de forma mais eficaz. Comparar tais conceitos com metas preestabelecidas, fundamentar a melhoria contínua e a evolução do software e verificar tendências auxiliaria de forma positiva no processo de desenvolvimento do produto. De acordo com Fernandes (1995), para a correta adoção desses conceitos é requerido que:

1. Um processo de software definido em termos de políticas de desenvolvimento, procedimentos, padrões, métodos, técnicas e ferramentas;
2. Medições relativas a atributos do projeto, como prazos, recursos, custo e esforço;

3. Medições relativas a atributos do processo, como cobertura de testes, nível de detecção e remoção de defeitos, nível de complexidade do projeto, produtividade;
4. Medições relativas a atributos do produto, como confiabilidade, nível de detecção de defeitos, índice de manutenção, tamanho do software;
5. Medições relativas à satisfação do cliente
6. Sistema de garantia da qualidade, incluindo gerência de configuração, planos de qualidade, inspeção formal de software, técnicas de testes de sistemas e assim sucessivamente.

Portanto, segundo Fernandes (1995), a principal importância da mensuração do software é fornecer aos desenvolvedores informações que sejam pertinentes ao gerenciamento e planejamento do processo de desenvolvimento ou evolução, tornando então possível o controle do trabalho com maior exatidão e tornando o seu conceito em relação ao sistema mais confiável, do ponto de vista do cliente. Tal realidade, aponta para 12 necessidades importantes que levam a utilização das medições, sendo elas que:

1. As estimativas de prazos, recursos, esforço e custo são realizadas com base no julgamento pessoal do gerente de projeto;
2. A estimativa do tamanho do software não é realizada;
3. A produtividade da equipe de desenvolvimento não é mensurada;
4. A qualidade dos produtos intermediários não é mensurada;
5. A qualidade do produto final não é medida;
6. O aperfeiçoamento da qualidade do produto ao longo de sua vida útil não é medido;
7. Os fatores que impactam a produtividade e a qualidade não são determinados;
8. A qualidade do planejamento dos projetos não é medida;
9. Os custos de não conformidade ou da má qualidade não são medidos;
10. A capacidade de detecção de defeitos introduzidos durante o processo não é medida;
11. Não há ações sistematizadas no sentido de aperfeiçoar continuamente o processo de desenvolvimento e de gestão do software;
12. Não há avaliação sistemática da satisfação dos usuários (clientes).

Tais premissas levam a consideração de que as medidas, métricas e medições se tornam um importante fator no processo de evolução e melhoria contínua do software, considerando que as supracitadas são consideradas como as bases da engenharia de software empírica.

Para Endres e Rombach (2003), a engenharia de software empírica visa pesquisar as experiências em sistemas de software e a coleta de dados e informações sobre projetos reais, que são utilizadas para formar e validar hipóteses sobre os métodos e técnicas utilizados no desenvolvimento do processo e/ou projeto. Onde, se é possível confiar no valor dos métodos e das

técnicas de engenharia se e, somente se, pudermos fornecer evidências concretas de que os mesmos oferecem os benefícios pelos quais foram construídos.

Para Sommerville (2011), mesmo quando se é possível realizar medições objetivas e tirar conclusões acerca das mesmas, isto pode não ser suficiente para convencer na tomada de decisões. Em vez disso, as tomadas de decisões são influenciadas, muitas vezes, por fatores meramente subjetivos, como a novidade ou a extensão em que as técnicas são de interesse para os profissionais.

Como consequência, podemos definir que para alcançar os objetivos, os aspectos relevantes ao processo de desenvolvimento devem ser observados e medidos. Utilizando dados coletados, modelos, *releases*, relacionamentos entre projetos/processos e recursos de software que podem auxiliar a algumas teorias e expectativas a serem confirmadas ou refutadas.

Tais questões evidenciam a importância das medidas, métricas e medições, pelas quais podem se tornar um fator essencial para o sucesso no gerenciamento de projetos e na melhoria dos processos.

7. MELHORIA E EVOLUÇÃO CONTÍNUA DE SOFTWARE

O processo de desenvolvimento usado para criar um sistema de software influencia a qualidade desse sistema. Por isso, muitas vezes os desenvolvedores são levados a acreditam que melhorar o processo de desenvolvimento de software acarreta melhorias na qualidade de software.

De acordo com (PFLEEGER, 2004) a vida útil de um sistema não acaba com a realização de sua entrega. Um software, durante o seu ciclo de vida, necessita de evolução contínua, podem surgir erros e falhas, mudanças nas regras de negócios da empresa, novos requisitos, entre outros. A evolução de software é importante porque as organizações são muito dependentes dos sistemas que investiram e que utilizam.

De acordo com Hamborg (2004), as Leis de Lehman começaram a ser formuladas no início dos anos 70 através da análise do processo de programação da IBM. Neste período foram formuladas as três primeiras leis, e na década de 80, foram adicionadas mais duas leis. Na década de 90 foram apresentadas mais três leis, formando assim um total de 8 que correspondem as leis de Lehman adotadas atualmente.

Adotar técnicas de evolução de software pode ser de grande importância em meio às transformações que o software está sujeito a sofrer. Segundo Lehman apud Araújo e Travassos (2008), as Leis de Evolução de Software descrevem como um sistema se comporta ao longo de sucessivas versões. Dessa forma as futuras mudanças são de certa forma, controladas.

Como as modificações estão presentes na vida de um software, é preciso nos preparar para as próximas e tentar reverter através das técnicas de evolução de software, o envelhecimento do software, ou seja, adiar o dia em que este já não poderá ser mais viável.

Segundo Sommerville (2011), o trabalho de Lehman e Belady (1985) alegam que essas leis são suscetíveis de serem verdadeiras para todos os tipos de sistemas de software de grandes organizações e que nesses sistemas, os requisitos estão mudando para refletir as necessidades dos negócios. Novos releases do sistema são essenciais para o sistema agregar valor ao negócio. A seguir, na Tabela 2, é possível verificar as Leis de Evolução de Software.

Tabela 2 – Leis de Evolução de Software

Lei	Descrição
1- Mudança Continua	Um programa usado em um ambiente do mundo real deve necessariamente mudar, ou se torna progressivamente menos útil nesse ambiente.
2 - Aumento da Complexidade	Como um programa em evolução muda, sua estrutura tende a tornar-se mais complexa. Recursos extras devem ser dedicados a preservar e simplicar a estrutura.
3 - Evolução de Programa de Grande Porte	A evolução de programa é um processo de autorregulação. Atributos de sistema como tamanho, tempo entre releases e número de erros relatados são aproximadamente invariáveis para cada release do sistema.
4 - Estabilidade Organizacional	Ao longo da vida de um programa, sua taxa de desenvolvimento é aproximadamente constante e independente dos recursos destinados ao desenvolvimento do sistema.
5 - Conservação da Familiaridade	Durante a vigência de um sistema, a mudança incremental em cada release é aproximadamente constante.
6 - Crescimento Contínuo	A funcionalidade oferecida pelos sistemas tem de aumentar continuamente para manter a satisfação do usuário.
7 - Declínio da Qualidade	A qualidade dos sistemas cairá, a menos que eles sejam modificados para refletir mudanças em seu ambiente operacional.
8 - Sistema de Feedback	Os processos de evolução incorporam sistemas de feedback multiagentes, multiloop, e você deve tratá-los como sistemas de feedback para alcançar significativa melhoria do produto.

A primeira lei se relaciona a mudança contínua, para Sommerville (2011) a primeira lei afirma que a manutenção do sistema é um processo inevitável. Portanto, o software em uso, ou irá passar por mudanças contínuas, ou irá, de forma progressiva se tornando menos útil. A segunda lei versa sobre o aumento da complexidade. Ao passo que um sistema passa por manutenções,

sua estrutura vai aos poucos sendo degradada e sua complexidade consequentemente irá aumentar.

Sommerville (2011) considera necessárias as manutenções preventivas, a fim de aprimorar continuamente a estrutura do software, não se resumindo apenas a adicionar funcionalidades. Consequentemente, haverá custos adicionais para a realização de tal manutenção. A terceira lei indica que a dinâmica da estrutura definida inicialmente para o sistema influenciará nas mudanças, podendo em alguns casos até restringir modificações em um sistema. Conforme o sistema vai evoluindo, o mesmo ficará mais complexo e difícil de ser compreendido, dificultando a sua manutenibilidade.

Para Sommerville (2011) as pequenas mudanças graduais são mais viáveis, pois evitam a redução de confiabilidade do sistema. Com a realização de grandes mudanças, há uma tendência de surgirem muitos defeitos. A quarta lei de Lehman é relativa à estabilidade organizacional.

Lehman e Bellady (1985) consideram que, em grandes projetos, a ocorrência de mudanças de recursos ou de pessoal não é um fator determinante na evolução do sistema a longo prazo. Na quinta lei, Lehman relaciona os incrementos do sistema juntamente com o surgimento de defeitos. Ao adicionar uma nova funcionalidade no sistema, também são adicionados proporcionalmente e de forma inevitável novos defeitos.

Para Sommerville (2011) é necessário a cada *release* de sistema um novo *release* apenas para correção de defeitos. Além disso, ao se realizar grandes incrementos de funcionalidade, a longo prazo acarretará na necessidade de se haver um orçamento prevendo correção de defeitos.

As demais leis foram adicionadas posteriormente, em outro trabalho de Lehman. A sexta e sétima leis são análogas e, essencialmente, dizem que os usuários de um determinado sistema estão cada vez mais insatisfeitos com o mesmo, a não ser que o software seja evoluído e que novas funcionalidades sejam adicionadas com o tempo. A última lei reflete o mais recente trabalho com processos de feedback, embora ainda não esteja claro como isso pode ser aplicado em desenvolvimento prático de software.

As observações de Lehman possuem um valor a agregar aos processos de desenvolvimento de software e devem ser levadas em consideração ao se planejar um processo de manutenção. Através das leis e da classificação que Lehman impõe sobre os sistemas, provê-se um vocabulário amplamente aceito para discutir a natureza das mudanças dos softwares, sejam elas quais forem.

Com base nestas mesmas leis, é possível projetar sistemas para serem mais flexíveis, realizar o planejamento das manutenções, além de entender e controlar de uma melhor forma o desenvolvimento do software, e não apenas reagir aos problemas que ocorrem durante o processo.

8. MINERAÇÃO DE REPOSITÓRIOS DE SOFTWARE

Segundo Xie e Hassan (2009) os repositórios de software são ricos em dados sobre o projeto de software e contêm informações que podem direcionar a tomada de decisão. O campo da Mineração de Repositórios de Software

(Mining Software Repositories - MSR) analisa e cruza dados ricos disponíveis em repositórios de software para descobrir informações interessantes e recursos sobre sistemas de software.

Ao longo dos últimos tempos, a mineração e os repositórios de software receberam maior atenção dos pesquisadores. Os dados provenientes do desenvolvimento de software podem ser obtidos de diferentes fontes de dados. Segundo Xie e Hassan (2011), é possível realizar diferentes análises com os dados coletados e aplicar diferentes algoritmos de mineração, para explicar diferentes tarefas do processo de desenvolvimento do software.

A mineração de dados na engenharia de software pode auxiliar no entendimento das atividades do processo de desenvolvimento, bem como permitir a construção de conhecimento para planejar, entender e predizer aspectos relacionados a um projeto ou processo.

Com base nestas afirmativas, foi-se analisado o repositório de software do *Kernel* do GNU/Linux, através da ferramenta Imagix 4D[1]. Nele, foram-se analisadas se as métricas em relação as Leis de Lehman e se ainda estão conformidade nos dias atuais, considerando métricas de desenvolvimento de software como embasamento dos resultados.

8.1 PROJETO DE CÓDIGO ABERTO: KERNEL DO LINUX

Embora o Linux seja possivelmente o sistema operacional de software livre mais popular, seu histórico é, na verdade, bastante breve, considerando a cronologia dos sistemas operacionais.

Nos primórdios da computação, os programadores desenvolviam diretamente sobre o hardware, na linguagem dele. A ausência de um sistema operacional significava que apenas um aplicativo (e um usuário) por vez poderia utilizar o dispositivo grande e dispendioso. Os primeiros sistemas operacionais foram desenvolvidos nos anos 1950, para fornecer uma experiência mais simples de desenvolvimento.

Os primeiros computadores pessoais surgiram por volta de 1980, lançado pela IBM, em parceria com a Microsoft. Numa negociação extremamente perspicaz, Bill Gates garantiu que toda máquina da IBM utilizasse seu sistema operacional, o DOS, retendo, todavia, seu direito de comercialização.

Desde então, o software passou a ser cada vez mais caro e, por se tratar de um software comercial, seu código fonte não era divulgado. A expressão "software livre" é creditada a Richard Stallman, que em 1983, ele fundaria o Projeto GNU, com o objetivo de criar um sistema operacional completo e totalmente baseado em software livre. O projeto GNU foi responsável por criar todos os utilitários que foram integrados aos *kernel* desenvolvido pelo Linus Torvalds em 1991.

---

[1] Imagix 4D – Disponível em: < https://www.imagix.com/index.html >

Por conta disto, o *Kernel* do GNU/Linux foi considerado como um dos repositórios de software para ser investigado sua qualidade, justamente por ser considerado um software livre. Atualmente, os dados provenientes das *releases* geradas pelo Linus Torvalds estão disponíveis para o público[2] na Linux Foundation, que tem como principal finalidade distribuir gratuitamente o kernel do Linux e outros softwares de código aberto ao público.

8.2 FERRAMENTA DE APOIO: IMAGIX 4D

A ferramenta Imagix 4D é uma poderosa facilitadora no âmbito da engenharia reversa e da análise de código fonte. Segundo Imagix (2019) a ferramenta tem como principal objetivo proporcionar a compreensão de códigos nas linguagens C, C++ e Java, sejam eles sistemas legados ou de código fonte aberto.

Visa também a aceleração do desenvolvimento, testes, reutilização e manutenção. Pois permite verificar rapidamente ou sistematicamente o software em qualquer nível de sua arquitetura, visualizando detalhes e sua construção, classes, métodos e funções, de forma rápida e precisa.

A ferramenta se mostrou muito útil na análise de software de código aberto, para avaliar a qualidade do código. A análise estática utilizando os dados abrangentes coletados através de repositórios de software possibilitou o levantamento de resultados para este artigo. Sendo possível considerar que as métricas geradas pelos releases auxiliariam a verificar a garantia de qualidade e dos esforços de compreensão do programa.

A partir das métricas e dos relatórios de análise de qualidade, foi-se possível gerar os resultados desta pesquisa. Durante o levantamento dessa análise foi disponibilizado por parte da empresa uma licença temporário de uso do software Imagix 4D.

9. ANÁLISE

Foram analisados nesse estudo 3 *releases* do *Kernel* do GNU/Linux, sendo elas a saber, a 4.2, 5.0 e 5.1. Algumas das métricas coletadas em cada *release* encontram-se disponíveis no anexo deste documento.

Com base nos conjuntos de dados observados, foi possível determinar que algumas das leis de evolução de software se aplicavam ao sistema, sendo elas as leis 1, 3, 4, 5 e 8.

Essa validade pode ser explicada por uma série de fatores, tais como o lançamento regular de novas versão, o controle do repositório principal pelo Linus Torvalds, o controle da qualidade pela própria comunidade que também da seu *feedback*.

Observando os dados, também verificou-se que entre a maior parte dos arquivos, não existe variação significativa no código, tornando as métricas colhidas quase imutáveis.

---

[2] *Kernel* Linux – Dispnível em: < https://github.com/torvalds/linux >

A validade dessas métricas depende de novas observações, considerando um conjunto mais de dados.

10. CONSIDERAÇÕES

A partir dos repositórios pré-existentes de software, buscar e identificar características que demostram a evolução ou declínio na qualidade do software, através da mineração das informações do repositório, tendo como base as *releases* geradas é um trabalho árduo e que requer maior esforço da comunidade visto os ganhos que podemos obter com base no histórico.

Com base na MRS, é possível analisar os efeitos das refatorações, do tamanho do projeto, e da experiência dos desenvolvedores, entre outros fatores.

Com base nas 3 *releases* estudadas do *Kernel* do GNU/Linux, verificou-se que algumas das leis não foram observadas, necessitando de mais investigação para valida-la ou refuta-la, sendo elas as leis 2, 6 e 7. As demais leis se mostraram válidas ao estudo.

Esse estudo é o passo inicial da pesquisa que pretende aumentar o número de observações do *Kernel* e analisar outros softwares comerciais para fomentar melhor os resultados.

ABSTRACT

The main purpose of this paper is to investigate software quality in open source projects, taking into consideration the quality and evolution standards imposed by Lehman and Bellady. Focused on analyzing, through software metrics, factors such as cyclomatic complexity, number of classes, methods and number of classes. In this work we verified the main differences between 3 releases of the GNU / Linux Kernel.

REFERÊNCIAS

ABNT, ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **Normas ABNT de Engenharia de software** – Qualidade de produto – Parte 1: Modelo de qualidade. NBR ISO/IEC 9126-1. Rio de Janeiro, 2003.

ARAÚJO, Marco Antônio Pereira; TRAVASSOS, Guilherme Horta. **A System Dynamics Model based on Cause and Effect Diagram to Observe Object-Oriented Software Decay**. Technical Report ES-720/08, 2008.

BOEHM, B. W.; BROWN, J. R.; KASPAR, H.; LIPOW, M.; MacLEOD, G.; MERRIT, M. **Characteristics of Software Quality**. Amsterdam: North-Holland, 1978.

CARDOSO, Eduardo José. **Métricas para programação orientada a objetos**. 1999. 45 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

CÔRTES, Mario L.; CHIOSSI, Thelma C. S. **Modelos de qualidade de software**. Campinas: Editora da UNICAMP, 2001.

DEMARCO, Tom. **Controle de projetos de software: gerenciamento, avaliação, estimativa**. Rio de Janeiro: Campus, 1989.

ENDRES, Albert; ROMBACH, H. Dieter. **A handbook of software and systems engineering: Empirical observations, laws, and theories**. Pearson Education, 2003.

ENGHOLM JR, Hélio. **Engenharia de software na prática**. Novatec Editora, 2010.

FERNANDES, Aguinaldo Aragon. **Gerência de software através de métricas: garantindo a qualidade do projeto, processo e produto**. Atlas, 1995.

GUNNING, R. **Techniques of Clear Writing**. Nova York: McGraw-Hill, 1962.

HAMBORG, Kai-Christoph. **Fragebögen zur Bestimmung der ergonomischen Qualität von Software**. Tagungsband UP04, 2004.

IMAGIX. **Visualization and dependency analysis of source code**. *In*: Visualization and dependency analysis of source code. [*S. l.*]: Imagix 4D, 2019. Disponível em: https://www.imagix.com/products/source-code-analysis.html. Acesso em: 25 nov. 2019.

JACOBSON, Ivar et al. **Object oriented software engineering: a use case driven approach**. Wokingham: Addison Wesley, 1992.

KRUCHTEN, Philippe; NORD, Robert L.; OZKAYA, Ipek. **Technical debt: From metaphor to theory and practice**. Ieee software, v. 29, n. 6, p. 18-21, 2012.

LEHMAN, Manny M.; BELADY, Laszlo A. **Program evolution: processes of software change**. Academic Press Professional, Inc., 1985.

MOLLER, K. H, PAULISH, D. J. **Software metrics: a practitionerïs guide to improved product development.** Los Alamitos: IEEE, 1993. 257p.

PFLEEGER, Shari Lawrence. **Engenharia de software: teoria e prática**. Prentice Hall, 2004.

PRESSMAN, R. **Engenharia de Software**. 7 ed. McGraw-Hill, 2011.

SIRQUEIRA, Tassio Ferenzini Martins et al. **Code Smell Analyzer: A Tool To Teaching Support Of Refactoring Techniques Source Code**. IEEE Latin America Transactions, v. 14, n. 2, p. 877-884, 2016.

SOMMERVILLE, Ian. **Engenharia de Software**. 9. ed. São Paulo: Pearson Addison Wesley, 2011.

XIE, T.; HASSAN, Ahmed E. **Mining Software Engineering Data**. Presented at the 59 31st International Conference on Software Engineering (ICSE 2009), Tutorials, Vancouver, Canada, 2009.

XIE, T.; HASSAN, Ahmed E. **Mining Software Engineering Data**. Presented in the 33rd International Conference on Software Engineering (ICSE 2011), Technical Briefing, Honolulu, Hawaii, 2011.

ANEXO

Linux 4.2
Summary
Project: demo_c_cpp
location: c:/Program Files/Imagix/data/demo_oomrm
database: 14 Dec 2010 (17:05)
data src: Dialog Based (C/C++)
descript:
document: 16 Oct 2019 (18:17)

File Summary

Top 100 Files

Files Sorted by Size, Bytes

Top 100 Files

Files File Name	Sorted Size (Bytes)	by Number Lines	Size, Number Stmts	Bytes Number Decls	Maint Index	Average Cyclo
ucos_ii.h	79382	1895	1033	320	-	-
oomrm_c.c	23998	433	166	4	126	1.10
cpu.h	18833	361	107	41	-	-
encodedmotor.cpp	18399	419	240	17	124	2.44
camerac328.cpp	18020	545	325	4	113	6.40
types.h	16319	449	163	44	-	-
cameragb.cpp	15600	531	119	3	113	2.10
tpu_regs.h	13262	387	277	8	-	-
shared.h	12166	376	25	7	-	-
uart.cpp	12127	293	225	24	133	2.38
primitive_driver.h	11605	198	112	1	-	-
status.h	11518	378	56	13	-	-
odcmotor.h	11332	200	7	1	-	-
persistent.h	10808	253	269	127	-	-
os_cfg.h	10714	144	78	76	-	-
os_cfg.h	10714	144	1	0	-	-
oschedule.h	10656	280	55	2	-	-
hardware.h	10593	289	164	6	-	-
dcmotor.cpp	10192	292	139	1	136	2.25
spi.cpp	9760	273	106	3	124	2.71
sci.cpp	9557	365	170	10	132	1.60

schedule.cpp	9400	261	102	15	128	2.13
tpu.cpp	9058	273	107	3	127	1.75
cpu_def.h	8364	133	11	11	-	-
fixedpoint.c	7634	415	163	0	95	5.33
object_list.h	6686	144	76	2	-	-
schedule2.cpp	6577	185	94	10	138	2.70
ouart.h	6574	138	79	9	-	-
status.cpp	6526	213	93	0	131	2.45
ocameragb.h	6365	139	61	1	-	-
sim_regs.h	6032	344	233	2	-	-
encoder.cpp	5856	165	109	0	141	1.83
os_cpu.h	5771	124	37	15	-	-
fixedpoint.h	5751	203	65	40	-	-
ovbr.h	5555	112	21	1	-	-
ocamerac328.h	5494	121	68	3	-	-
pwm.cpp	5443	151	74	0	141	1.20
bencodedmotor.h	5370	114	90	1	-	-
uart_rtos_handler.c	5224	105	52	10	99	10.00
qsm_regs.h	5077	263	172	2	-	-
ttsemic.cpp	5050	180	84	8	129	2.38
sci_rtos_handler.c	5016	147	91	7	79	26.00
osonar.h	4849	95	29	2	-	-
uart_handler.c	4798	159	49	10	109	1.00
os.h	4700	110	130	1	-	1.00
Ehandler.c	4653	114	48	12	137	10.00
oencodedmotor.h	4630	98	7	1	-	-
ottsemic.h	4502	131	38	2	-	-
oomrm.h	4500	112	32	15	-	-
vbr.cpp	4391	124	31	4	141	1.60
gbcam.h	4246	142	33	9	-	-
oconsole.h	4202	89	28	1	-	-
oprocessor.h	4166	103	27	1	-	-
sci_default_handler.c	4084	116	59	5	89	19.00
otpu.h	4033	110	43	1	-	-
uart_default_handler.c	3929	82	32	10	112	3.00
led.cpp	3677	125	50	0	133	1.00
opwm.h	3647	74	7	1	-	-
bdcmotor.h	3568	82	58	1	-	-
osci.h	3520	105	84	3	-	-
compssdv.cpp	3475	93	27	0	140	1.25
oencoder.h	3360	84	7	1	-	-

sim_reg.h	3351	64	58	55	-		-
ocompass.h	3271	72	21	1	-		-
critical.h	3151	71	10	4	-		-
sine.c	3112	79	38	1		132	4.33
ir.cpp	3077	115	55	0		128	3.50
bpwm.h	3067	62	28	1	-		-
processor.cpp	3044	104	49	1		138	2.33
sonardv.cpp	3029	85	32	0		141	1.50
lcd.cpp	3012	114	67	0		129	1.57
eb500.cpp	3002	98	48	1		137	2.17
bencoder.h	2989	66	45	3	-		-
qsm.cpp	2967	67	29	2		138	1.33
oir.h	2840	81	22	1	-		-
types.h	2644	62	23	9	-		-
timer.cpp	2624	92	1	0	-		-
oeb500.h	2560	68	34	1	-		-
bdio1.h	2535	77	34	2	-		-
bled.h	2355	62	27	1	-		-
do1.cpp	2292	62	25	0		125	3.50
oa2d.h	2287	59	17	1	-		-
ospi.h	2249	56	29	1	-		-
oeeprom_spi.h	2226	77	22	1	-		-
osharpgp2d12.h	2164	58	11	1	-		-
image_normalize8.cpp	2013	65	38	2		108	6.00
ocamera.h	2008	66	26	9	-		1.00
di1.cpp	1981	55	17	0		130	2.00
sharpgp2d12.cpp	1981	49	16	1		140	1.67
ram.h	1951	47	19	13	-		-
odio2.h	1921	69	37	1	-		-
app_cfg.h	1911	54	20	18	-		-
blcd.h	1907	55	29	3	-		-
a2d.cpp	1896	59	24	0		151	1.20
rx_settings.h	1859	49	22	5	-		-
checksum.c	1777	48	22	1		117	5.00
odio4.h	1768	56	37	1	-		-
ooreg.h	1754	36	13	11	-		-
hexcode.c	1714	69	40	0		103	19.00
getfield.c	1711	42	23	0		117	9.00

Linux 5.0
Summary
Project: demo_c_cpp

location: c:/Program Files/Imagix/data/demo_oomrm
database: 14 Dec 2010 (17:05)
data src: Dialog Based (C/C++)
descript:
document: 16 Oct 2019 (18:18)

Top 100 Files

Files File Name	Sorted Size (Bytes)	by Number Lines	Size, Number Stmts	Bytes Number Decls	Maint Index	Average Cyclo
ucos_ii.h	79382	1895	1033	320	-	-
oomrm_c.c	23998	433	166	4	126	1.10
cpu.h	18833	361	107	41	-	-
encodedmotor.cpp	18399	419	240	17	124	2.44
camerac328.cpp	18020	545	325	4	113	6.40
types.h	16319	449	163	44	-	-
cameragb.cpp	15600	531	119	3	113	2.10
tpu_regs.h	13262	387	277	8	-	-
shared.h	12166	376	25	7	-	-
uart.cpp	12127	293	225	24	133	2.38
primitive_driver.h	11605	198	112	1	-	-
status.h	11518	378	56	13	-	-
odcmotor.h	11332	200	7	1	-	-
persistent.h	10808	253	269	127	-	-
os_cfg.h	10714	144	78	76	-	-
os_cfg.h	10714	144	1	0	-	-
oschedule.h	10656	280	55	2	-	-
hardware.h	10593	289	164	6	-	-
dcmotor.cpp	10192	292	139	1	136	2.25
spi.cpp	9760	273	106	3	124	2.71
sci.cpp	9557	365	170	10	132	1.60
schedule.cpp	9400	261	102	15	128	2.13
tpu.cpp	9058	273	107	3	127	1.75
cpu_def.h	8364	133	11	11	-	-
fixedpoint.c	7634	415	163	0	95	5.33
object_list.h	6686	144	76	2	-	-
schedule2.cpp	6577	185	94	10	138	2.70
ouart.h	6574	138	79	9	-	-
status.cpp	6526	213	93	0	131	2.45
ocameragb.h	6365	139	61	1	-	-

sim_regs.h	6032	344	233	2	-	-
encoder.cpp	5856	165	109	0	141	1.83
os_cpu.h	5771	124	37	15	-	-
fixedpoint.h	5751	203	65	40	-	-
ovbr.h	5555	112	21	1	-	-
ocamerac328.h	5494	121	68	3	-	-
pwm.cpp	5443	151	74	0	141	1.20
bencodedmotor.h	5370	114	90	1	-	-
uart_rtos_handler.c	5224	105	52	10	99	10.00
qsm_regs.h	5077	263	172	2	-	-
ttsemic.cpp	5050	180	84	8	129	2.38
sci_rtos_handler.c	5016	147	91	7	79	26.00
osonar.h	4849	95	29	2	-	-
uart_handler.c	4798	159	49	10	109	1.00
os.h	4700	110	130	1	-	1.00
Ehandler.c	4653	114	48	12	137	10.00
oencodedmotor.h	4630	98	7	1	-	-
ottsemic.h	4502	131	38	2	-	-
oomrm.h	4500	112	32	15	-	-
vbr.cpp	4391	124	31	4	141	1.60
gbcam.h	4246	142	33	9	-	-
oconsole.h	4202	89	28	1	-	-
oprocessor.h	4166	103	27	1	-	-
sci_default_handler.c	4084	116	59	5	89	19.00
otpu.h	4033	110	43	1	-	-
uart_default_handler.c	3929	82	32	10	112	3.00
led.cpp	3677	125	50	0	133	1.00
opwm.h	3647	74	7	1	-	-
bdcmotor.h	3568	82	58	1	-	-
osci.h	3520	105	84	3	-	-
compssdv.cpp	3475	93	27	0	140	1.25
oencoder.h	3360	84	7	1	-	-
sim_reg.h	3351	64	58	55	-	-
ocompass.h	3271	72	21	1	-	-
critical.h	3151	71	10	4	-	-
sine.c	3112	79	38	1	132	4.33
ir.cpp	3077	115	55	0	128	3.50
bpwm.h	3067	62	28	1	-	-
processor.cpp	3044	104	49	1	138	2.33
sonardv.cpp	3029	85	32	0	141	1.50
lcd.cpp	3012	114	67	0	129	1.57

eb500.cpp	3002	98	48	1	137	2.17
bencoder.h	2989	66	45	3	-	-
qsm.cpp	2967	67	29	2	138	1.33
oir.h	2840	81	22	1	-	-
types.h	2644	62	23	9	-	-
timer.cpp	2624	92	1	0	-	-
oeb500.h	2560	68	34	1	-	-
bdio1.h	2535	77	34	2	-	-
bled.h	2355	62	27	1	-	-
do1.cpp	2292	62	25	0	125	3.50
oa2d.h	2287	59	17	1	-	-
ospi.h	2249	56	29	1	-	-
oeeprom_spi.h	2226	77	22	1	-	-
osharpgp2d12.h	2164	58	11	1	-	-
image_normalize8.cpp	2013	65	38	2	108	6.00
ocamera.h	2008	66	26	9	-	1.00
di1.cpp	1981	55	17	0	130	2.00
sharpgp2d12.cpp	1981	49	16	1	140	1.67
ram.h	1951	47	19	13	-	-
odio2.h	1921	69	37	1	-	-
app_cfg.h	1911	54	20	18	-	-
blcd.h	1907	55	29	3	-	-
a2d.cpp	1896	59	24	0	151	1.20
rx_settings.h	1859	49	22	5	-	-
checksum.c	1777	48	22	1	117	5.00
odio4.h	1768	56	37	1	-	-
ooreg.h	1754	36	13	11	-	-
hexcode.c	1714	69	40	0	103	19.00
getfield.c	1711	42	23	0	117	9.00

Linux 5.1

Summary

Project: demo_c_cpp
location: c:/Program Files/Imagix/data/demo_oomrm
database: 14 Dec 2010 (17:05)
data src: Dialog Based (C/C++)
descript:
document: 16 Oct 2019 (18:15)

File Summary

Top 100 Files

Files Sorted by Size, Bytes

File Name	Size (Bytes)	Number Lines	Number Stmts	Number Decls	Maint Index	Average Cyclo
ucos_ii.h	79382	1895	1033	320	-	-
oomrm_c.c	23998	433	166	4	126	1.10
cpu.h	18833	361	107	41	-	-
encodedmotor.cpp	18399	419	240	17	124	2.44
camerac328.cpp	18020	545	325	4	113	6.40
types.h	16319	449	163	44	-	-
cameragb.cpp	15600	531	119	3	113	2.10
tpu_regs.h	13262	387	277	8	-	-
shared.h	12166	376	25	7	-	-
uart.cpp	12127	293	225	24	133	2.38
primitive_driver.h	11605	198	112	1	-	-
status.h	11518	378	56	13	-	-
odcmotor.h	11332	200	7	1	-	-
persistent.h	10808	253	269	127	-	-
os_cfg.h	10714	144	78	76	-	-
os_cfg.h	10714	144	1	0	-	-
oschedule.h	10656	280	55	2	-	-
hardware.h	10593	289	164	6	-	-
dcmotor.cpp	10192	292	139	1	136	2.25
spi.cpp	9760	273	106	3	124	2.71
sci.cpp	9557	365	170	10	132	1.60
schedule.cpp	9400	261	102	15	128	2.13
tpu.cpp	9058	273	107	3	127	1.75
cpu_def.h	8364	133	11	11	-	-
fixedpoint.c	7634	415	163	0	95	5.33
object_list.h	6686	144	76	2	-	-
schedule2.cpp	6577	185	94	10	138	2.70
ouart.h	6574	138	79	9	-	-
status.cpp	6526	213	93	0	131	2.45
ocameragb.h	6365	139	61	1	-	-
sim_regs.h	6032	344	233	2	-	-
encoder.cpp	5856	165	109	0	141	1.83
os_cpu.h	5771	124	37	15	-	-
fixedpoint.h	5751	203	65	40	-	-
ovbr.h	5555	112	21	1	-	-
ocamerac328.h	5494	121	68	3	-	-
pwm.cpp	5443	151	74	0	141	1.20
bencodedmotor.h	5370	114	90	1	-	-

uart_rtos_handler.c	5224	105	52	10	99	10.00
qsm_regs.h	5077	263	172	2	-	-
ttsemic.cpp	5050	180	84	8	129	2.38
sci_rtos_handler.c	5016	147	91	7	79	26.00
osonar.h	4849	95	29	2	-	-
uart_handler.c	4798	159	49	10	109	1.00
os.h	4700	110	130	1	-	1.00
Ehandler.c	4653	114	48	12	137	10.00
oencodedmotor.h	4630	98	7	1	-	-
ottsemic.h	4502	131	38	2	-	-
oomrm.h	4500	112	32	15	-	-
vbr.cpp	4391	124	31	4	141	1.60
gbcam.h	4246	142	33	9	-	-
oconsole.h	4202	89	28	1	-	-
oprocessor.h	4166	103	27	1	-	-
sci_default_handler.c	4084	116	59	5	89	19.00
otpu.h	4033	110	43	1	-	-
uart_default_handler.c	3929	82	32	10	112	3.00
led.cpp	3677	125	50	0	133	1.00
opwm.h	3647	74	7	1	-	-
bdcmotor.h	3568	82	58	1	-	-
osci.h	3520	105	84	3	-	-
compssdv.cpp	3475	93	27	0	140	1.25
oencoder.h	3360	84	7	1	-	-
sim_reg.h	3351	64	58	55	-	-
ocompass.h	3271	72	21	1	-	-
critical.h	3151	71	10	4	-	-
sine.c	3112	79	38	1	132	4.33
ir.cpp	3077	115	55	0	128	3.50
bpwm.h	3067	62	28	1	-	-
processor.cpp	3044	104	49	1	138	2.33
sonardv.cpp	3029	85	32	0	141	1.50
lcd.cpp	3012	114	67	0	129	1.57
eb500.cpp	3002	98	48	1	137	2.17
bencoder.h	2989	66	45	3	-	-
qsm.cpp	2967	67	29	2	138	1.33
oir.h	2840	81	22	1	-	-
types.h	2644	62	23	9	-	-
timer.cpp	2624	92	1	0	-	-
oeb500.h	2560	68	34	1	-	-
bdio1.h	2535	77	34	2	-	-

bled.h	2355	62	27	1	-		-
do1.cpp	2292	62	25	0		125	3.50
oa2d.h	2287	59	17	1	-		-
ospi.h	2249	56	29	1	-		-
oeeprom_spi.h	2226	77	22	1	-		-
osharpgp2d12.h	2164	58	11	1	-		-
image_normalize8.cpp	2013	65	38	2		108	6.00
ocamera.h	2008	66	26	9	-		1.00
di1.cpp	1981	55	17	0		130	2.00
sharpgp2d12.cpp	1981	49	16	1		140	1.67
ram.h	1951	47	19	13	-		-
odio2.h	1921	69	37	1	-		-
app_cfg.h	1911	54	20	18	-		-
blcd.h	1907	55	29	3	-		-
a2d.cpp	1896	59	24	0		151	1.20
rx_settings.h	1859	49	22	5	-		-
checksum.c	1777	48	22	1		117	5.00
odio4.h	1768	56	37	1	-		-
ooreg.h	1754	36	13	11	-		-
hexcode.c	1714	69	40	0		103	19.00
getfield.c	1711	42	23	0		117	9.00