DAS Departamento de Automação e Sistemas
CTC Centro Tecnológico
UFSC Universidade Federal de Santa Catarina

Desenvolvimento de uma API para integração de softwares sob um padrão aberto

Relatório submetido à Universidade Federal de Santa Catarina como requisito para a aprovação na disciplina

DAS 5511: Projeto de Fim de Curso

Maycon Stamboroski

Florianópolis, julho de 2014

Desenvolvimento de uma API para integração de softwares sob um padrão aberto

Maycon Stamboroski

Esta monografia foi julgada no contexto da disciplina
DAS5511: Projeto de Fim de Curso
e aprovada na sua forma final pelo
Curso de Engenharia de Controle e Automação

Prof. *Ricardo J. Rabelo*

____

Assinatura do Orientador

Agradecimentos

Resumo

Estamos caminhando em direção aos ambientes altamente integrados. Desejamos que nossos e-mails estejam nos nossos computadores, smartphones e até televisões, ao mesmo tempo, instantaneamente e sincronizados.

A indústria também requer isso para seus trabalhos, ambientes integrados geram menos custos, mais rapidez, flexibilidade na resolução de problemas e um apoio maior na tomada de decisões.

A empresa onde o PFC foi realizado atua num nicho de mercado, o de construção predial civil, que não apenas demanda, também necessita de ambientes altamente integrados. Felizmente isso já vem sendo teorizado e desenvolvido desde o começo da década de 70, porém com a evolução tecnológica agora estamos cada vez mais próximos disso.

Contudo, tudo começa com o passo inicial, no caso da AltoQi, a empresa em questão, busca integrar seus produtos a um padrão aberto da área de construção civil, o IFC, já que este é uma peça fundamental no sucesso das soluções integradas baseadas em BIM.

Mesmo atuante na área de tecnologia para a engenharia há mais de 25 anos, datando com produtos comercializados desde o final dos anos 80, os aplicativos da empresa ainda têm espaço para evoluir. Buscando a integração dos seus sistemas desenvolvidos, tanto os mais atuais, os considerados “legados” e os que virão, o primeiro passo foi dado com o conceito da criação do Projeto Qi4D.

Um projeto que busca modularizar os softwares desenvolvidos e então uni-los sob a base do IFC. Isto é contemplado através de uma refatoração parcial dos programas, focando em como seus objetos tridimensionais são concebidos. Criando junto com eles uma estrutura de metadados flexível e intercambiável, garantindo a interoperabilidade entre as soluções. Essa interoperação será dada através de uma API própria e pública que foi

desenvolvida neste projeto, contemplando as funções descritas acima e podendo ser expandida conforme as evoluções tecnológicas.

Palavras-chave: Integração de sistemas, IFC, BIM, API.

Abstract

We are heading toward highly integrated systems. We want our e-mails to be available in our computers, smartphones and televisions, at the same time, instantly and synchronized.

The industry also requires that for its work. Highly integrated environments generate less cost, more speed, flexibility on problems resolution and a larger support for decision making.

The company where the PFC was fulfilled operates in a niche market, civil building construction, which not only requires, but also demands for highly integrated environment and systems. Hopefully, that is a topic which has been under studies since the beginning of the 70's, and with the technological evolution, we are getting close to it.

Nevertheless, everything starts with the first step. AltoQi, the company in the matter, searches for the integration of its products upon an open standard in civil construction area, the IFC, a fundamental piece on the success of BIM based integrated solutions.

Despite its presence in technological solutions for civil engineering for about 25 years, with products applications dated from the late 80's, the company softwares have room for evolution. Looking for system integration of its own developed products, from the more moderns, to that called "legacy" and for the new ones to come, the Qi4D Project was created.

A project for modularization of the developed softwares, and from that, aggregated then upon the IFC assumption. This is contemplated with a partial refactoring of the applications, focused on how the tridimensional objects are designed. Adding with it, the creation of a flexible, switchable and generic metadata structure, allowing interoperation between solutions. This interoperation will occur through a new API, with public visibility, which was developed in the Qi4D Project, contemplating the functionalities described above and able to be expanded with technological evolution.

Sumário

Agradecimentos 3

Resumo 4

Abstract 6

Índice de imagens 8

Simbologia 10

Capítulo 1: Introdução. 11

Capítulo 2: A empresa. 14

Capítulo 3: O projeto Qi4D. 17

3.1: Building Information Modeling (BIM) e Industry Foundation Classes (IFC) 17

3.2: Introdução ao projeto 19

3.3: O conceito de API sob a ótica do Qi4D 20

3.4: Requisitos do projeto 21

3.4.1: Requisitos funcionais 22

3.4.2: Requisitos não funcionais 24

Capítulo 4: O Novo Visualizador 3D 28

Capítulo 5: Incorporando a API ao sistema existente 38

Capítulo 6: Resultados Obtidos 48

Capítulo 7: Conclusões e Perspectivas 54

Bibliografia: 57

Índice de imagens

Figura 1 - Primeiros produtos da empresa 14
Figura 2 - Primeira versão do Eberick 15
Figura 3 - Exemplo de arquivo IFC para um pilar 19
Figura 4 - Diagrama da Framework Qi4D 22
Figura 5 - Arquitetura do novo visualizador 29
Figura 6 - Arquitetura do novo visualizador com os produtos da empresa 31
Figura 7 - Hierarquia do desacoplamento dos módulos em camadas 33
Figura 8 - Exemplos de propriedades no QiBuilder (esquerda) e Eberick (direita) 34
Figura 9 - Grafo acíclico dos metadados 35
Figura 10 - Ilustração das branches criadas para a tarefa 37
Figura 11 - Antiga hierarquia de objetos 3D no Eberick 38
Figura 12 - Hierarquia dos objetosOpenGL no Qi4D 39
Figura 13 - Hierarquia dos conversores OpenGL no Qi4D 39
Figura 14 - Hierarquia de classes no projeto Qi4D 40
Figura 15 - Chamadas do método que busca por um conversor específico no hashmap 40
Figura 16 - Registro do conversor no hashmap ao ser criado pelo dispatcher e chamadas deste método 41
Figura 17 - Grafo de sequencia para a criação de um novo elemento a cada frame 42
Figura 18 - Grafo dos métodos que requisitam pela criação de um novo elemento convertido e as classes que o implementam 43
Figura 19 - Elementos que aceitam a visita 44
Figura 20 - Diagrama de classes dos visitor 45
Figura 21 - Diagrama de classes das factories 45
Figura 22 - Classes singletons para criação dos metadados 47
Figura 23 - Factory de metadados 47
Figura 24 - Complexidade ciclomática antes das mudanças 49

Figura 25 - Complexidade ciclomática após as alterações.................... 49
Figura 26 - Ilustração dos módulos desacoplados 50
Figura 27 - Erro de duplicação de código na geração continua de versões... 51
Figura 28 - Nenhum erro encontrado na geração continua de versões. 51
Figura 29 - Grafo de Kiviat para classe responsável pela criação dos novos elementos 3D.. 52
Figura 30 - Ferramenta de gerenciamento de projetos 53

Simbologia

A seguir:

PAC - Projeto de Aceleração ao Crescimento

PMCV - Projeto Minha Casa Minha Vida

PIB - Produto Interno Bruto

CBIC - Câmara Brasileira de Indústria e Comércio

ABNT - Associação Brasileira de Normas Técnicas

BIM - Building Information Modeling

IFC - Industry Foundation Classes

STEP - Standard for the Exchange of Product Data

API - Application Program Interface

DSM - Design Structure Matrix

MVP - Model View Presenter

ISO - Organização Internacional para Padronização

IEC - International Electrotechnical Commission

SVN – Apache Subversion Client

TDD - Test Driven Development

BDD - Behavior Driven Development

VTK - The Visualization Toolkit

OpenGL - Open Graphics Library

Capítulo 1: Introdução.

O setor de construção civil está em franca expansão no Brasil. Os programas governamentais como o Projeto de Aceleração ao Crescimento (PAC) e o Projeto Minha Casa Minha Vida (PMCV) impulsionaram o setor nos últimos anos, aliado a isso, há o investimento em infraestrutura devido aos eventos grandes mundiais sediados pelo país, como a Copa do Mundo em 2014 e as Olimpíadas em 2016.

Em matéria publicada em dezembro de 2013 [1], a revista Exame apontou um crescimento previsto para o setor de 2,8% em 2014, seguindo o crescimento do PIB brasileiro. Porém estudos publicados pela Câmara Brasileira de Indústria e Comércio (CBIC) entre 2009 e 2013 [2] mostram que para o setor continuar crescendo e a economia continuar competitiva, é necessário aumentar a produtividade.

Esses estudos também avaliaram onde encontram-se os gargalos dos projetos, enfatizando problemas com prazos, custos, integração com órgãos e legislações, entre outros. A empresa AltoQi atua no ramo oferecendo softwares para projetos de edificações, integrando além do projeto estrutural, o projeto hidrossanitário, elétrico, gás, projeto de instalações hidráulicas de combate à incêndio, entre outas soluções, todas baseadas nas normas brasileiras gerenciadas pela ABNT. O uso destes softwares além de aumentar a produtividade, diminuindo o tempo de projeto, também oferece uma base de segurança pela sua integração com a norma e possibilidade de otimização no uso de materiais.

O conceito de BIM (*Building Information Modeling)* já é utilizado desde 1987 na criação de "Edificações Virtuais" [3], e sobre estes princípios os softwares da empresa são desenvolvidos. O Comitê Norte Americano para Padrões de Projetos e Modelos Informacionais em Construções define o BIM como: "Uma representação digital das funcionalidades e características físicas de uma construção. O BIM é um recurso de conhecimento compartilhado que busca fornecer uma base confiável de informações de uma construção,

auxiliando a tomada de decisões durante o seu ciclo de vida.” Porém há uma carência na integração das funcionalidades fornecidas pelos softwares da empresa. Para solucionar este problema, apoia-se na solução de formatos de arquivo IFC (Industry Foundation Classes).

O IFC foi desenvolvido pela buildingSMART usando os conceitos e especificações da STEP (Standard for the Exchange of Product Data), a fim de criar um modelo de dados consistente para representação de uma edificação, sendo um “Padrão Aberto” para troca de informações de diferentes tipos de software [4].

Para uma compreensão melhor da importância dos modelos IFCs, propõe-se um caso de estudo chamado “A Torre de Babel”, um edifício residencial multi-familiar. Para este projeto supõe-se que o projeto arquitetural será desenhado no Revit (da empresa AutoCAD), o projeto estrutural modelado no Eberick (da empresa AltoQi) e os projetos hidráulico e elétrico no software QiBuilder (ambos também da AltoQi).

O dono do edifício cria o projeto e disponibiliza-o para os profissionais referentes a cada área da construção. Supõe-se que o projeto base será o arquitetural, o projetista exporta o modelo criado no formato IFC e o associa à construção específica. Os outros profissionais agora podem acessar o arquivo IFC e trabalhando paralelamente, criar seus modelos específicos para o projeto, vinculando-os à mesma construção.

Após todos os projetos concluídos, o dono da construção importa o modelo IFC agregado de todos os projetos, podendo visualizar o modelo tridimensional total da solução. Para os produtos da AltoQi participarem deste novo cenário do mercado de arquitetura e construção civil, o pré-requisito é o sucesso do projeto Qi4D, para que com a integração obtida, seja possível levar os sistemas da empresa a um novo patamar de colaboração.

O engenheiro de controle e automação é uma peça fundamental no projeto de sistemas interoperáveis, como é o caso. A integração de sistemas está intrinsicamente ligada ao desenvolvimento de software, não só pelo fato de muitas vezes ser necessário “colocar a mão na massa” e desenvolver linhas de

código, mas também pelo fato que a visão de arquitetura do sistema e conceitos de arquitetura de software são necessários. Somam-se a isso os requisitos de qualidade de mercado, onde os sistemas precisam operar rapidamente e sem erros.

Capítulo 2: A empresa.

Conhecer um pouco a história da empresa e seus produtos criados é fundamental para entender a importância do projeto desenvolvido. A AltoQi é líder em software para projetos de edificações no Brasil. Existente desde 1989, iniciou suas atividades com o lançamento e comercialização do software PROVIGA, e entre os anos 1991 a 1993 completou seu leque de soluções com os softwares PROPILAR, PROLAJE e PROINFRA. Esses sistemas eram baseados em DOS e buscavam suprir a demanda por automatização do conhecimento já existente na época.

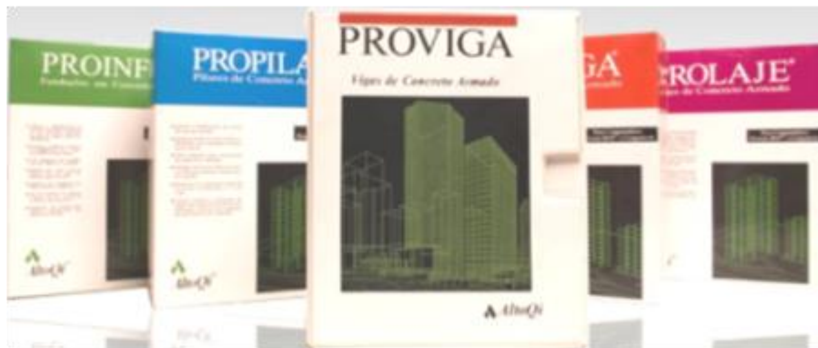


Figura 1 - Primeiros produtos da empresa

Nos anos seguintes, uma empresa estrangeira entrou no mercado brasileiro, oferecendo estas soluções em um software único e com interface gráfica. Consequentemente a AltoQi se viu obrigada a reestruturar-se, para não perder mercado. Entre os anos de 1993 a 1996 a empresa concentrou seus esforços no desenvolvimento de um sistema, integrando os módulos já existentes em um único produto de projetos estruturais e sob uma camada de interface gráfica em CAD. Isso culminou para que, em 1996, fosse lançado o software AltoQi Eberick, o carro chefe da empresa até hoje e responsável pela sua retomada de mercado. O Eberick continua em desenvolvimento, recebendo atualizações e novas funcionalidades regularmente.

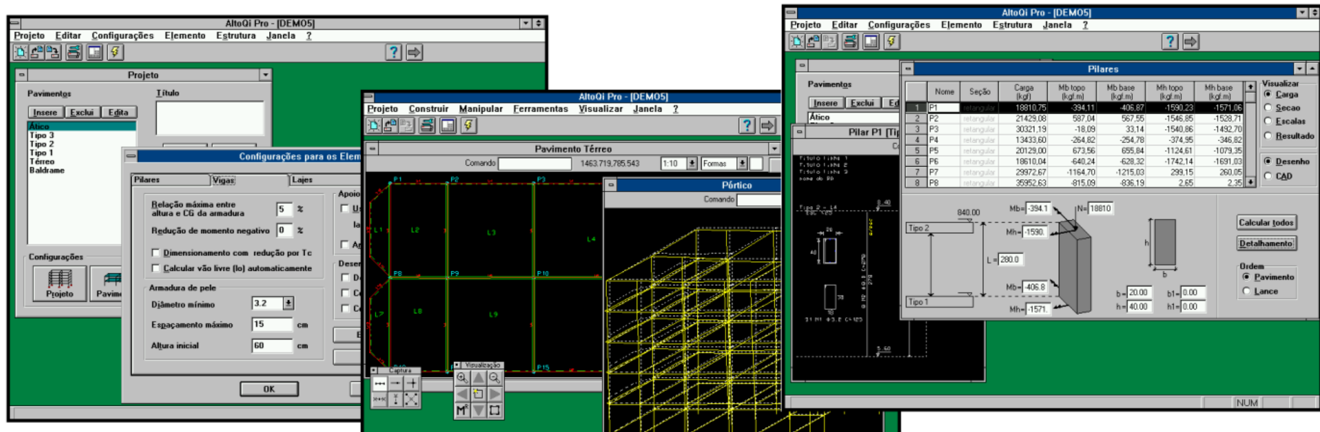


Figura 2 - Primeira versão do Eberick

Entre o final da década de noventa e começo dos anos dois mil, a empresa completou o seu hall de soluções com mais dois novos produtos, o AltoQi Hydros, que engloba projetos hidráulicos, sanitários, de incêndio e gás, e o AltoQi Lumine, contemplando projetos elétricos, telefônicos e de cabeamento estruturado. Esses softwares são maduros e estão em um estágio onde todos os recursos solicitados pelos usuários já estão desenvolvidos, posicionando-se assim acima dos seus concorrentes.

Por volta de 2005, algumas más escolhas no desenvolvimento do Eberick começaram a cobrar seu preço, como descrito abaixo:

"No final de 93, quando iniciamos o desenvolvimento do Eberick, a ferramenta escolhida foi o Borland C++. Esta oferecia uma framework para desenvolvimento em Windows chamada OWL, que era fundamentalmente uma hierarquia de classes que faziam uma "casca" sobre a API do Windows Baseamos então todo o software sobre essa framework, utilizando intensivamente as classes, tipos e estruturas de dados que eram oferecidas. Não prestamos atenção ao fato de que aquilo era uma framework para interface, e que poderia mudar a qualquer momento. No início de 95 já sentimos os primeiros prejuízos: a nova versão do Borland C++, 4.0 na época, trazia a framework totalmente modificada, baseada em templates. Tivemos então um enorme trabalho reescrevendo o programa para a nova abordagem. Mas continuamos utilizando a framework como se esta fosse realmente parte da linguagem. Em 97 o Borland C++ chegou à sua versão 5.02 e algum tempo depois se concluiu que esta seria a última. Ao mesmo tempo em que Eberick

começava a amadurecer como produto, sua ferramenta de desenvolvimento chegava à aposentadoria. E hoje (2005!), o Eberick continua sendo escrito sobre a mesma ferramenta. O grande problema disso é que será impossível acompanhar as novas tecnologias (.net, Windows 64 bits, etc.) com uma ferramenta descontinuada há oito anos." (trecho de "Para onde estamos indo", por Adriano Coser [5]).

Com esse problema necessitando de resolução, a empresa resolveu por adotar uma abordagem do tipo "*The great refactoring in the sky*". Assim iniciou-se o projeto do QiCAD, que consiste numa framework própria para ambientes CAD, assim fornecendo uma base para a futura reestruturação e integração dos produtos existentes e dos novos a serem criados. A evolução desse projeto é o QiBuilder, um programa novo, desenvolvido sobre o conceito nativo de integração, onde os recursos criados são naturalmente parte de um mesmo ambiente computacional. O QiBuilder é desenvolvido com a ferramenta C++Builder da Borland e pode-se agregar novas funcionalidades ao programa, como a interface Ribbon da Qt.

Porém fazem cerca de dez anos que começou essa corrida para que o QiBuilder alcance os softwares já existentes, além disso, essa arquitetura adotada também começou a mostrar suas deficiências nesse tempo. Além disso, o time original do desenvolvimento do projeto não se encontra mais na empresa, e provavelmente muitas das intenções originais de código já se desviaram. Isso fez com que se começasse a adotar uma política de refatoração contínua nos softwares, obtendo um controle na qualidade do código criado.

Recentemente, o mercado começou a exigir que os softwares CADs fossem compatíveis com o modelo BIM. Alinhando-se com a visão da empresa e a ideia de melhorar a integração dos softwares, decidiu-se por iniciar o projeto Qi4D. Esse projeto busca aumentar a interoperabilidade entre os softwares existentes dentro da empresa e garantir que serão compatíveis com o padrão IFC.

Capítulo 3: O projeto Qi4D.

3.1: Building Information Modeling (BIM) e Industry Foundation Classes (IFC)

O conceito de edificações virtuais redireciona o modelo 3D como chave de integração CAD/CAM/CAE e planejamento. De acordo com Charles Eastmam [6], o BIM é um modelo estrutural de informação digital, tridimensional onde constam os vários objetos que compõem um edifício, capturando a sua forma, comportamento e relação nas várias partes do edifício, sendo possível indexar todo um conjunto de dados a um determinado elemento. Deste modo, envolvem-se tecnologias e processos para o desenvolvimento de uma prática de projeto integrada, no qual os participantes unam seus esforços na construção de um modelo único de edifício[7].

Com o advento de modelos 3D na indústria da arquitetura e construção civil e posterior incremento de funcionalidades nestes softwares, definiu-se o conceito do BIM 4D. Este modelo possibilita a visualização do projeto da construção, planejamento do CPM, gerenciamento da cadeia de suprimentos, gerencia de custos e riscos, mantendo a interoperabilidade com o CAD 3D, para prover uma simulação do edifício virtual. O papel do BIM 4D é adicionar uma nova dimensão aos modelos CAD 3D, juntando informações aos elementos tridimensionais do modelo [8].

Para o sucesso do BIM e a obtenção de um modelo único com toda a informação do projeto, é imprescindível a integração de softwares para a colaboração, coordenação e gerenciamento de informações, isto é alcançável através de um modelo padronizado distribuído, neste caso, o IFC.

O IFC é fruto dos esforços da *BuildingSMART Alliance* na criação de uma padrão que permita a interoperabilidade entre programas da área de BIM, facilitando a transferência de dados representativos de partes de edificações e

suas relações. O formato teve sua primeira especificação em 1997 e encontra-se na versão IFC4, sendo está aceita também como norma ISO.

O modelo de dados IFC é um modelo de dados baseado em definição de classes representando objetos. Ele é baseado em três entidades principais [17]: o *IfcRoot*, *ProxyObjects* e as *PropertySets*. O *IfcRoot* funciona de forma análoga à uma interface abstrata em relação à linguagens computacionais orientadas a objetos. Assim, as entidades individuais dentro do IFC são baseadas no *IfcRoot*, sendo elas objetos, propriedades e relações entre estes. Os objetos para o IFC são as geometrias dos elementos, a ideia do modelo 3D sólido. As propriedades representam os conceitos como: materiais, resistência ao cisalhamento, revestimento, entre ouras características contextuais. As relações são o que fazem o acoplamento entre objetos diferentes ou propriedades e objetos, definidas como classificações abstratas.

As outras duas entidades, *ProxyObjects* e *PropertySets* são responsáveis por estender o uso do padrão. Elas funcionam de forma análoga aos objetos e propriedades, respectivamente, definidas no IFC, mas através destas, são possíveis definir novos elementos que não existem no modelo. Desta forma, pode-se definir atributos e objetos que são importantes dentro de um contexto, empresa ou localidade, sem precisar fugir do padrão.

```
/* Pilares - Secção e características  */
   #425= IFCRECTANGLEPROFILEDEF (.AREA.,'20x30',$,0.20,0.30);
      #426= IFCEXTRUDEDAREASOLID(#425,#191,#13,3.0);
         #427= IFCSHAPEREPRESENTATION(#202,'Body','SweptSolid',(#426));
            #428= IFCPRESENTATIONLAYERASSIGNMENT('Columns Profile','No
                                                description',(#427),$);
            #429= IFCPRODUCTDEFINITIONSHAPE($,$,(#427));
   #430= IFCPROPERTYSINGLEVALUE ('MassPerLength',$,IFCMASSPERLENGTHMEASURE(25)
                               ,$);
   #431= IFCPROPERTYSINGLEVALUE ('CrossSectionArea',$,IFCAREAMEASURE(0.06),$);
   #432= IFCPROPERTYSINGLEVALUE ('MomentOfInertiaY',$,
                                IFCMOMENTOFINERTIAMEASURE(0.0002),$);
   #433= IFCPROPERTYSINGLEVALUE ('MomentOfInertiaZ',$,
                               IFCMOMENTOFINERTIAMEASURE(0.00045),$);
   #434= IFCPROPERTYSINGLEVALUE ('TorsionalSectionModulus',$,
                               IFCSECTIONMODULUSMEASURE(0.622),$);
      #435= IFCPROFILEPROPERTIES ('Pset_ProfileMechanical',$,(#430,#431,#432,
                                #433,#434),#425);
   #436= IFCCARTESIANPOINT ((0.0,0.0,0.0));
      #437= IFCAXIS2PLACEMENT3D (#436,#13,#9);
         #438= IFCLOCALPLACEMENT (#192,#437);
               #439= IFCCOLUMN ('GUID',#5,'Column 1',$,$,#438,#429,$,
                              .COLUMN.);
                  #440= IFCRELCONNECTSSTRUCTURALELEMENT ('GUID',#5,$,$,(#301),
                                                       #439);
   #441= IFCCARTESIANPOINT ((5.5,0.,0.));
      #442= IFCAXIS2PLACEMENT3D (#441,#13,#9);
         #443= IFCLOCALPLACEMENT (#192,#442);
               #444= IFCCOLUMN ('GUID',#5,'Column 2',$,$,#443,#429,$,
                              .COLUMN.);
                  #445= IFCRELCONNECTSSTRUCTURALELEMENT ('GUID',#5,$,$,(#311),
                                                       #444);
   #446= IFCCARTESIANPOINT ((0.,5.5,0.));
      #447= IFCAXIS2PLACEMENT3D (#446,#13,#9);
         #448= IFCLOCALPLACEMENT (#192,#447);
               #449= IFCCOLUMN ('GUID',#5,'Column 3',$,$,#448,#429,$,
                               .COLUMN.);
                  #450= IFCRELCONNECTSSTRUCTURALELEMENT ('GUID',#5,$,$,(#321),
                                                       #449);
   #451= IFCCARTESIANPOINT ((5.5,5.5,0.));
      #452= IFCAXIS2PLACEMENT3D (#451,#13,#9);
         #453= IFCLOCALPLACEMENT (#192,#452);
               #454= IFCCOLUMN ('GUID',#5,'Column 4',$,$,#453,#429,$,
                              .COLUMN.);
                  #455= IFCRELCONNECTSSTRUCTURALELEMENT ('GUID',#5,$,$,(#331),
                                                       #454);

                  #456= IFCRELCONTAINEDINSPATIALSTRUCTURE ('GUID',#5,$,$,(#439
                                                         ,#444,#449,#454),
                                                         #196);
```

Figura 3 - Exemplo de arquivo IFC para um pilar

3.2: Introdução ao projeto

Como relatado nos tópicos anteriores, o conceito de BIM se apoia no ideal de interoperabilidade, parametrização e colaboração entre diferentes sistemas. Isso pode ser alcançado utilizando diversas estratégias, como exploradas na disciplina de Integração de Sistemas Corporativos nesta universidade. Atualmente os softwares da empresa obtém certo nível de interoperabilidade usando o conceito de troca de arquivos, porém alguns dados e funcionalidades são perdidos nesse processo, além de ser um processo lento, com maior custo de armazenamento e propenso a retrabalho.

Para ampliar essa interoperabilidade, não apenas entre os softwares desenvolvidos dentro da empresa, mas também com outras soluções do

mercado, optou-se por aplicar novos conceitos nos produtos. O primeiro deles é a abordagem por interação de sistemas, obtida através de uma nova API criada, sendo ela uma interface de integração. A segunda solução é adotar a troca de informações através de um padrão aberto, que para este caso, está sendo baseado no formato IFC. Somando-se aos dois, após o fim do projeto, espera-se possível adotar a abordagem Top-Down de interoperabilidade para e empresa, utilizando e expandindo os conceitos obtidos no projeto Qi4D para os outros projetos.

Esta ótica de integração e os requisitos do mercado para que isto ocorra, abriu a possibilidade para a criação de um novo projeto para fornecer essas soluções e oferecer uma maior interoperabilidade entre os programas existentes da empresa. O Projeto Qi4D visa a criação de um novo sistema elaborado para substituir as janelas 3D existentes no Eberick e no QiBuilder. Para isso, as novas funcionalidades e pacotes serão encapsulados por APIs abstratas, e os softwares existentes serão refatorados para sua utilização.

3.3: O conceito de API sob a ótica do Qi4D

API é a sigla para *Application Programming Interface*, ou interface programável da aplicação. Uma API é um conjunto de regras e especificações particular que um programa deve seguir para acessar e fazer uso de serviços e recursos provenientes de outro software em particular que implementa esta API. Ela serve como uma interface, uma ponte de ligação, entre diferentes programas, facilitando a interação entre eles, de maneira similar como uma interface humano-máquina facilita a comunicação entre humanos e computadores[9].

Uma API foi desenvolvida para fornecer uma estratégia de integração global para o projeto Qi4D. Ela foi desenvolvida para criar uma estrutura genérica de informações e dados, que pode ser reutilizável em várias situações. A API, a principio, foi concebida para generalizar o modelo 3D e suas caraterísticas (metadados), poder ler e gerar arquivos no padrão IFC e garantir a persistência desses dados.

Assim, esforçou-se para modificar os produtos da empresa para tornarem-se condizentes com as regras e especificações da API. Desta forma, se novas aplicações seguirem as mesmas regras, elas funcionarão prontamente entre os diversos softwares da empresa.

Como exemplo podemos citar os motores gráficos utilizados, VTK para o QiBuilder e OpenGL para o Eberick. Ao adequar-se os respectivos softwares para a API, pode-se intercambiar estas ferramentas de desenho 3D e elas funcionaram sem grandes problemas nos dois softwares. O mesmo será possível caso deseje-se mudar como a persistência de dados é feita, para um banco de dados NoSQL, hipoteticamente. Além disso, novas aplicações desenvolvidas sob a API funcionaram para ambos os programas e pode-se terceirizar o desenvolvimento sem abrir o código fonte do programa, apenas apresentando este conjunto de regras e documentação adicional explicando-as.

3.4: Requisitos do projeto

O framework Qi4D será compartilhado pelas aplicações existentes e as novas que virão a serem criadas. Suas principais funcionalidades são a representação do modelo tridimensional (modelo geométrico) e a transferência dos seguintes tipos de informação entre as aplicações: dados dos elementos, dados da construção, informações de concreto, dados colaborativos e colisões entre modelos.

Todas as informações dos modelos 3D serão compartilhadas por todos os softwares e a transferência de dados podendo ser feita, primeiramente, através da troca de arquivos. A adesão dos programas à nova framework possibilita a extensão dessa troca de informações para novos ambientes, como web e mobile.

O integrador Qi4D é a aplicação responsável por unir os diferentes projetos em um único modelo 3D. Além disso, a framework pode ser utilizada para o desenvolvimento de várias outras aplicações, por exemplo, há a possiblidade de se adicionar informações temporais ao projeto, permitindo a outra aplicação ser responsável pelo planejamento e gerenciamento da construção.

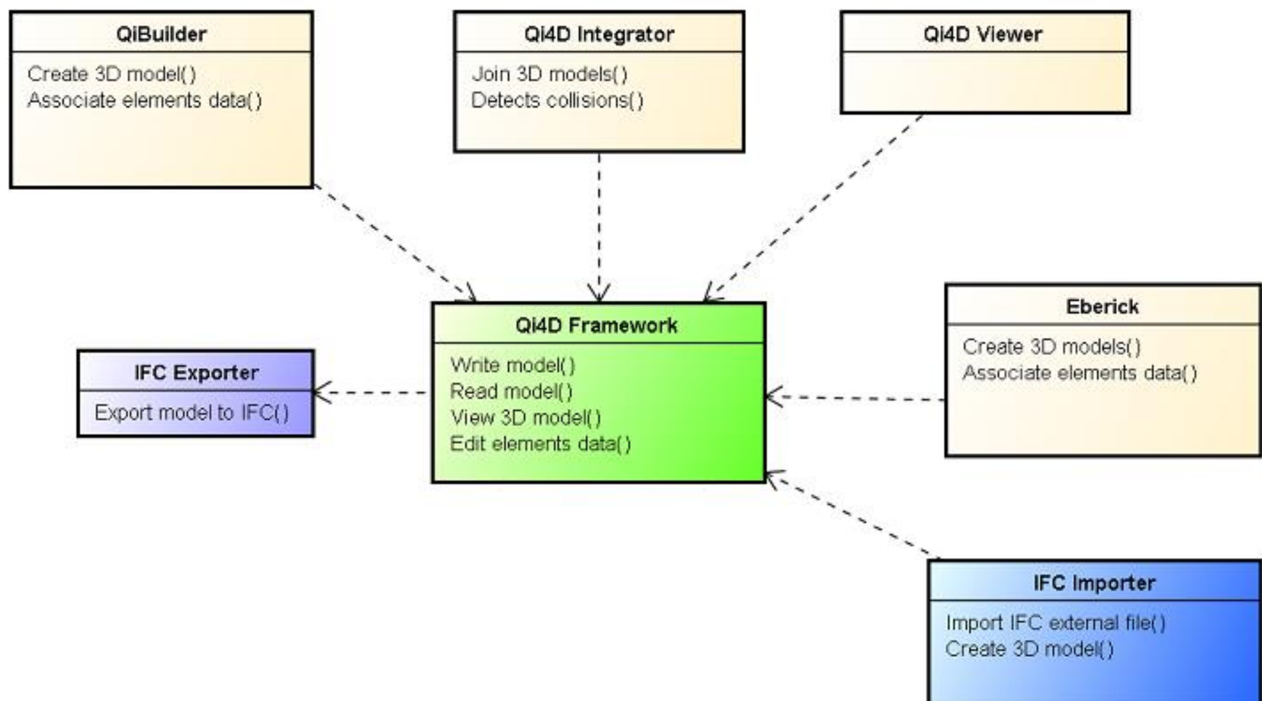


Figura 4 - Diagrama da Framework Qi4D

Para melhor entendimento dos requisitos do projeto, eles podem ser divididos em dois subconjuntos: requisitos funcionais, que representam o comportamento e características desejadas pelo sistema e requisitos não funcionais, que definem a forma que o software será produzido, padrões de qualidade, recursos que devem ser utilizados, entre outros.

3.4.1: Requisitos funcionais

Para o sucesso e viabilidade do projeto, é necessário que as seguintes funcionalidades estejam presentes nos programas englobados pela refatoração e interfaceados pela framework Qi4D:

- Gravar e ler modelos do servidor: Como exemplificado na introdução, deverá ser possível o acesso do usuário à plataforma web, para que seja possível obter e enviar o modelo geométrico e demais dados de elementos da construção, concreto e outros dados colaborativos, com as limitações de acesso definidas pelo administrador do projeto;
- Visualizar modelos 3D: Ao receber os modelos geométricos gerados pelo Eberick e QiBuilder, em seus formatos internos, a ferramenta deve gerar primitivas tridimensionais e renderizá-las para fornecer uma visualização, além de ferramentas básicas para isto;
- Exportar IFC: A ferramenta deve gerar um arquivo no formato IFC que inclua o modelo geométrico, os dados dos elementos dos projetos e instalações. Este arquivo deve ser lido por outros programas BIM de forma eficiente;
- Importar Objetos 3D: O software QiBuilder apresenta um cadastro de peças, visto que este é uma ferramenta mais abrangente, trabalhando com modelos hidráulicos e elétricos de edificações. Deseja-se que no caso de alguns tipo de peças em formatos mais elaborados e com geometria fixa elaborados em outros softwares (como o AutoCad), seja possível a substituição de elementos 3d gerados via código por estes, consequentemente atualizando o cadastro de peças e substituindo o objeto anterior pelo novo na visualização do 3D;
- Editar dados dos elementos: Através da seleção de um ou mais objetos (como um grupo), dentro do visualizador 3D, deve ser possível alterar seus dados de construção e de concreto.
- Visualizar colisões: As colisões de elementos e modelos, como, por exemplo, uma tubulação atravessando um pilar, devem ser identificadas na Edificação através de uma ferramenta para isto. Estes dados então devem ser acessados via uma lista ou através da visualização do modelo tridimensional.

Gravar backup do projeto no servidor: Ao autor do projeto deve ser possível criar uma cópia deste, seja no Eberick ou Qibuilder, e armazená-lo num servidor local ou remoto. Esta cópia também poderá ser restaurada pelo usuário, devidamente identificada e sem erros.

3.4.2: Requisitos não funcionais

Por ser um projeto novo, com uma arquitetura própria e por ser uma API que eventualmente pode ser disponibilizada publicamente, as funcionalidades desenvolvidas seguem um rigoroso protocolo para garantir sua qualidade.

Os requisitos básicos são: o código tem que ser desenvolvido sob o paradigma de orientação a objetos, ser utilizada a linguagem C++ seguindo o padrão C++03 definido pela ISO/IEC 14882:2003 e as aplicações binárias geradas devem ser nativamente compatíveis com 64-bits.

Como os outros projetos da empresa, este também segue as convenções de formatação da empresa, descrito em documentos internos. Há alguns anos a empresa também vem adotando a ideologia do *Clean Code* [10] descrita no livro "Clean Code: A Handbook of Agile Software Craftsmanship", de Robert C. Martin. As principais premissas são: simplicidade, ausência de duplicação, facilidade de leitura e elegância. Estas características podem ser obtidas seguindo alguns parâmetros, descritos a seguir:

- Limitação no número de parâmetros de uma função, tentando mantê-los menores ou iguais a três;
- Manter os métodos com menos de 20 linhas;
- Evitar o uso de comentários, apenas em situações muito específicas;
- O uso de comentários deve ser substituído por nomes de métodos e atributos que descrevem completamente sua funcionalidade e objetivo;
- Minimizar ao máximo a duplicação de código.

A arquitetura do projeto segue os parâmetros dos produtos definidos pela AltoQi, de forma geral, o sistema deve ser representado na forma de uma

DSM (Design Structure Matrix) na forma triangular inferior [11], onde os pacotes são distribuídos em camadas (layers) e cada pacote só pode acessar aqueles que estão abaixo da sua camada. Além disso, cada pacote deve ser isolado de outro através de APIs abstratas.

Também segue-se o padrão MVP (Model – View – Presenter), para o desenho da interface de usuário. Este modelo facilita a utilização de testes unitários automatizados, descritos logo adiante, e melhora a apresentação da lógica do modelo. Seguindo esse padrão, cada visualização é divida em três conceitos: modelo de dados (definidos pela camada *model*), a operação/ lógica de controle (gerenciados pela camada *view*) e a interface direta com o usuário (*presenter*) [12].

Para evitar problemas referentes a bibliotecas externas (como o exemplo do ocorrido com o Eberick e o compilador Borland), as APIs e suas implementações devem utilizar apenas tipos simples (*int, bool, double,* etc) ou primitivas mais complexas definidas na *QiPrimitives*, uma biblioteca própria da empresa que encapsula as funcionalidades necessárias vindas de outras bibliotecas. Esta diretriz busca garantir a portabilidade e independência do código.

Outra métrica que começou a ser utiliza no projeto Qi4D é o da Complexidade Ciclomática (ou complexidade condicional). Este conceito, descrito por Thomas J. McCabe em 1976 [13], define quantidade de decisões lógicas tomadas pelo código fonte de uma função, isto é, o número de caminhos independentes que um programa, ou método, pode tomar durante a sua execução. Vários estudos indicam que o valor da complexidade ciclomática está correlatado com o número de potenciais bugs no código analisado. Além disso, limitar este valor favorece a legibilidade, reusabilidade, manutenção e modularidade do código fonte. O SEI (Software Engineering Institute) oferece no seu guia de referência “C4 Technololy Guide” [14], uma tabela com os seguintes valores para a complexidade ciclomática:

- 1 – 10: método simples, sem muitos riscos;
- 11 – 20: método com complexidade média, riscos moderados;
- 21 – 50: método complexo, risco elevado;

- Maior que 50: método instável, risco elevadíssimo.

Para o código desenvolvido na empresa, leva-se em conta os critérios abaixo durante a escrita do código:

- Complexidade máxima por método: menor ou igual a 10;
- Complexidade média por método: menor ou igual a 2;
- Complexidade máxima por arquivo ou classe: menor ou igual a 100;
- Complexidade média por arquivo e classe: menor ou igual a 10;
- Nível de nidificação: menor ou igual a 4.

Quando há a necessidade da criação de uma nova estrutura de dados ou quando forem refatorados módulos muito grandes de código, procura-se adotar os *design patterns* mais propensos para cada situação. *Desing patterns*, ou padrões de projeto, são descrições de problemas conhecidos e as essências das soluções destes, para que estas possam ser configuradas de acordo com a necessidade do usuário. Ian Somerville [15] descreve-os como "*The pattern is not a detailed specification. Rather, you can think of it as a description of accumulated wisdom and experience, a well-tried solution to a common problem."*

Os requisitos de qualidade descritos acima são do tipo de análise estática, ou seja, acontecem durante a escrita do código. Em geral, a validação desses requisitos acontece na máquina do desenvolvedor através de programas específicos para cada uso (por exemplo, o Cpp Check) e através da revisão de código, obrigatória ser aprovada antes do profissional poder fazer seu "commit" no SVN (sistema de controle de versões). Por fim, ocorre também a validação destes requisitos e métricas através do servidor de *builds* contínuos, o Jenkins, que é integrado com o SVN.

Somando à qualidade dos softwares desenvolvidos pela empresa, uma parte importante da "cadeia de produção" são as análises dinâmicas do programa. Essas análises só podem ser feitas com o programa em execução.

Sob esse escopo, a primeira análise realizada é para vazamentos de memória. Vazamentos de memória é um aspecto recorrente em C++, ocorrem

quando um bloco de memória é dinamicamente alocado e nunca liberado, podendo acabar com a memória disponível do sistema ou levar a erros durante a execução do programa.

Em seguida parte-se para os testes. Para o projeto específico do Qi4D buscou-se agregar a ideia de testes unitários, TDD (Test Driven Development) e sua evolução, o BDD (Behavior Driven Development) [16]. Idealmente, cada função deve ter um teste específico para si, validando seus dados de entrada/saída. O BDD expande esse conceito, incorporando ferramentas externas ao desenvolvimento e fazendo que o código passe por testes de comportamento do sistema. Desse modo, garante-se que as funcionalidades criadas estejam de acordo com o esperado pelo “cliente” e que, ao exercitar o comportamento do sistema, também se está exercitando seus métodos privados e unitários, garantindo a integridade da solução.

Por fim, cada build gerada pelo servidor é validada por um “testador”, que avalia o impacto da solução no programa, a existência de novos bugs e os requisitos de performance, que são o tempo de resposta e o uso de memória. Ao fim disso, a última versão gerada no dia passa por uma bateria de testes automatizados com o programa em execução.

Capítulo 4: O Novo Visualizador 3D

O desenvolvimento desta monografia se deu paralelamente ao “Release 3” do projeto Qi4D. Está é a terceira parte das implementações previstas e é focada na criação do novo visualizador 3D, vinculado à nova API, para o Eberick e a prototipagem dos novos metadados associados a essa visualização.

O padrão IFC define a classificação de suas entidades em três categorias [17]: objetos, propriedades e relações. Para o projeto Qi4D, as relações (que podem ser entre objeto e propriedade ou entre objetos) e as propriedades são modeladas como Metadados, associados a um objeto.

Já os objetos do IFC estão intrinsicamente associados com sua geometria no espaço tridimensional, ou seja, ao ler ou gerar dados no padrão IFC, esperam-se objetos modelados em 3D. O framework Qi4D tem uma modelagem genérica de objetos que suporta as informações do IFC, porém o mesmo não ocorre para as aplicações existentes na empresa. Para isso, um novo modulo de visualização tridimensional precisou ser projetado, e os softwares da empresa precisaram de uma grande refatoração para seu uso. Para o caso do Eberick, esta separação e interfaceamento para um novo módulo é bastante complicado, pois ele é um sistema legado e altamente acoplado.

Este novo visualizador deve ser independe da plataforma e aplicação utilizada, fornecendo uma interface genérica para os vários produtos da empresa. A imagem abaixo ilustra a arquitetura pretendida do visualizador.

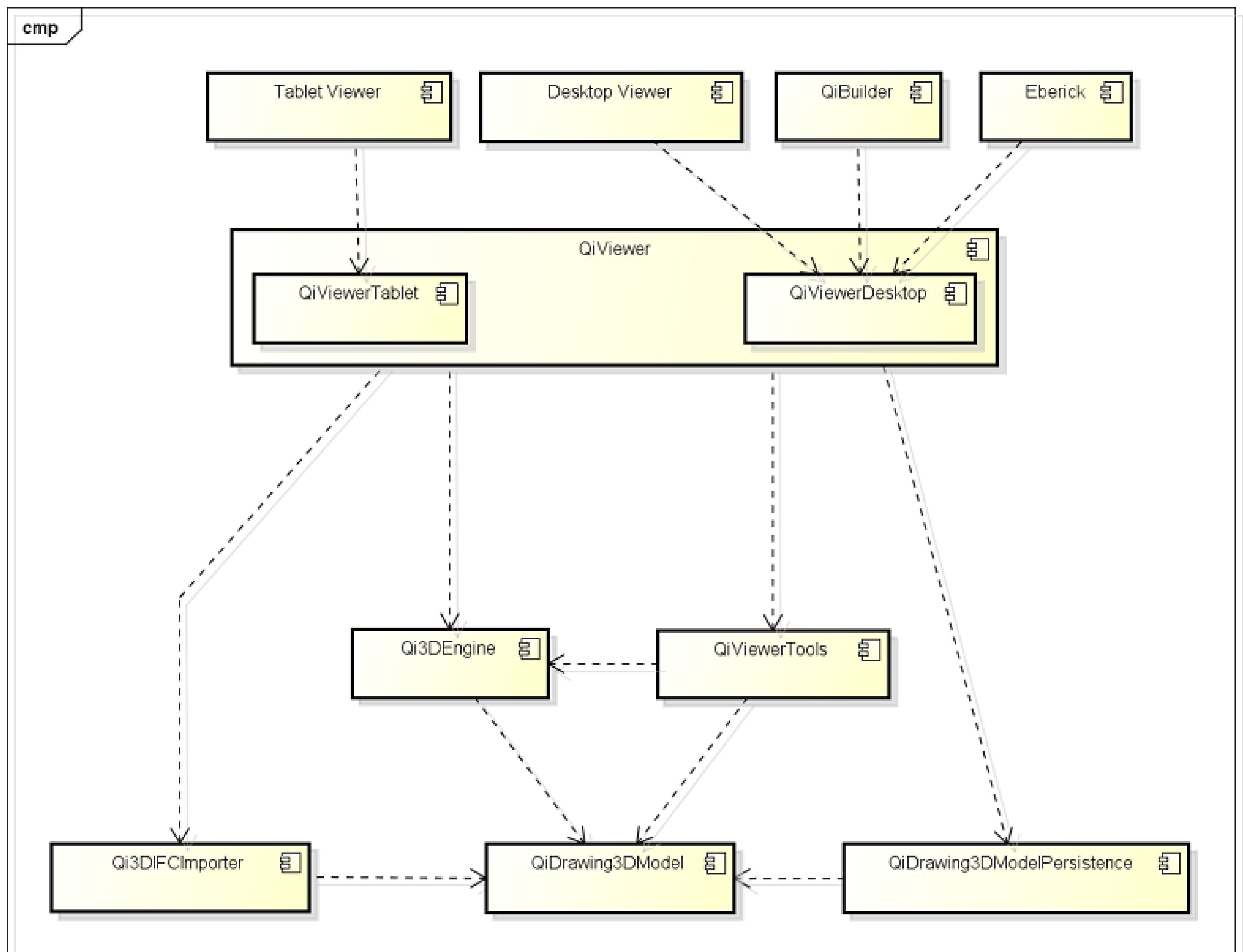


Figura 5 - Arquitetura do novo visualizador

Uma explicação rápida de cada módulo:

- Tablet Viewer: A aplicação visualizadora para plataformas mobile. É construída agregando vários módulos existentes;
- Desktop Viewer: É a aplicação para visualização do modelo em desktops (computadores de mesa e notebooks), também é construída agregando módulos e funcionalidades já existentes em outros pacotes;
- QiBuilder: É o programa “QiBuilder” já existente da empresa;
- Eberick: É o programa “Eberick” já existente da empresa;

- QiViewerTablet: É o código da interface gráfica para usuários (GUI) mobile;
- QiViewerDesktop: É o código da interface gráfica para usuários (GUI) da aplicação em desktop;
- QiViewerTools: É o módulo responsável por gerenciar as ferramentas utilizadas pelos visualizadores;
- Qi3DEngine: É o módulo de visualização 3D. É responsável por receber informações dos modelos 3D e apresenta-los na tela num ambiente tridimensional virtual. Deve ser possível utiliza-lo pelas aplicações visualizadoras mencionadas e diretamente pelos programas existentes na empresa. Sua única dependência deve ser do módulo *QiDrawing3DModel*;
- QiDrawing3DModel: É a definição do modelo que representa as entidades 3D utilizados pela Engine;
- Qi3DIFCImporter: Módulo responsável por converter as informações dos modelos 3D vindos dos arquivos IFC em modelos no formato *Qi3DDrawingModel*;
- Qi3DDrawingModelPersistence: Módulo responsável pela persistência dos dados dos modelos de elementos 3D.

Como explicado no início deste capítulo, um dos objetivos do projeto é a utilização do visualizado 3D nos programas já existentes na empresa, isso deve ocorrer através da adição e utilização dos módulos *Qi3DEngine* e *QiDrawing3DModel* nas aplicações. A figura a seguir define a arquitetura desta implementação.

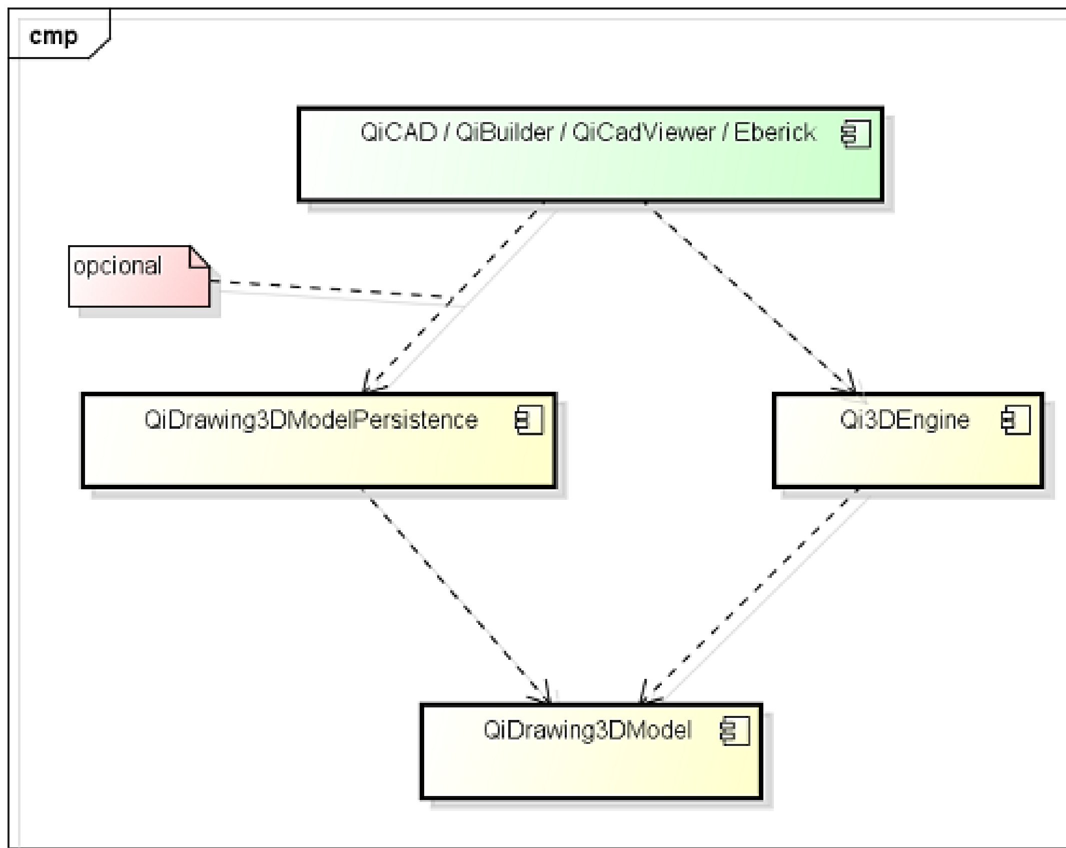


Figura 6 - Arquitetura do novo visualizador com os produtos da empresa

Para o projeto dos visualizadores 3D, foram especificados como requisitos funcionais mínimos, independente da interface dos programas, os seguintes tópicos:

- Exibir os elementos contidos no modelo em perspectiva;
- Para cada elemento, ser possível alterar a sua cor, transparência e visibilidade (opção de esconder ou mostrar no modelo);
- Operações básicas de manipulação: aproximação e afastamento da câmera, rotação, deslocamento (*pan*) do modelo, rotação dinâmica e diferenças do comportamento da câmera quando dentro ou fora da edificação;
- Filtros para visualização de mais de um tipo de projeto, pavimento ou elemento;

- Inserção de planos de corte nos eixos X, Y e Z, positivos ou negativos;

Como dito anteriormente, para a utilização do novo visualizador 3D nos programas existentes foi necessária uma grande alteração no código-fonte das aplicações. Tanto no caso do QiBuilder quanto no Eberick, as primitivas 3D e as *engines* utilizadas estavam “misturadas” com outras partes de código, não respeitando a hierarquia desejada. Ressaltando que para o QiBuilder o motor gráfico 3D atualmente utilizado é o *VTK* e para o Eberick o *OpenGL.*

A refatoração dos dois programas utilizou uma metodologia semelhante. O primeiro passo é descobrir quais elementos de cada motor gráfico são realmente utilizados e após isso, entender como a transferência de informações para desenho dos elementos é realizada. O próximo passo é criar uma nova hierarquia de representação dos dados, que seja independente do programa ao qual está associada, porém mantendo a mesma funcionalidade já existente, para que não se insira novos erros nos softwares existentes.

Essa nova hierarquia criada, deve servir como ponte entre a *engine* existente e os modelos da biblioteca Qi3DModel e para validar sua independência, deve ser apta a compilar em mais de uma IDE existente. Depois dessa validação, modifica-se a forma de desenho dos programas existentes, fazendo que obrigatoriamente sejam utilizados os elementos dessa nova arquitetura.

A etapa seguinte é substituir a forma como os programas criam suas primitivas 3D, pelos novos modelos vindos da Qi3DModel, associá-los com os novos *Metadados* e criar os filtros para os novos modelos. Dessa forma já se tem a aplicação parcialmente desacoplada dos motores gráficos utilizados anteriormente. A parte final é substituir o motor gráfico pela *Qi3DEngine* e fazê-la se comunicar com a API de desenho respectiva.

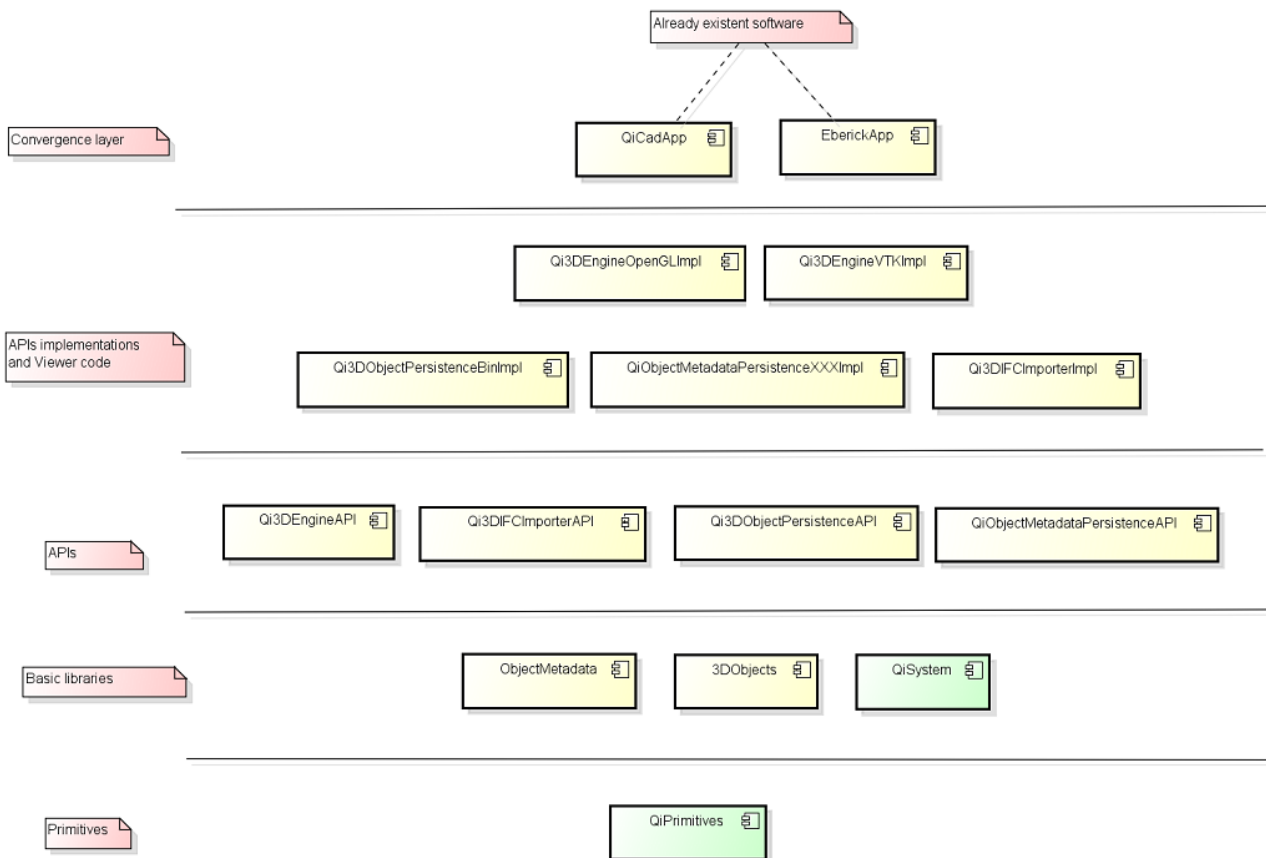


Figura 7 - Hierarquia do desacoplamento dos módulos em camadas

Como já frisado, estes novos objetos tridimensionais estão associados à *Metadados* que também foram desenvolvidos durante o projeto. A solução buscada para este caso também tenta oferecer uma forma flexível e de baixo acoplamento para resolver o problema.

Nas aplicações existentes, cada elemento dentro do modelo tridimensional tem uma série de propriedades, como: nome, tipo, pavimento associado, nome do projeto e etc. Estas informações são utilizadas no visualizador 3D para gerenciar objetos, por exemplo, mostrando somente aqueles pertencentes a um pavimento, ou alterando a cor de um grupo.

Para o Qi4D, especificou-se que os objetos podem ter qualquer tipo de propriedades, estas podendo ser tipos simples ou estruturas complexas, onde as estruturas complexas têm sub-propriedades. Esta estratégia é aplicada utilizando-se de um grafo acíclico para representação das propriedades.

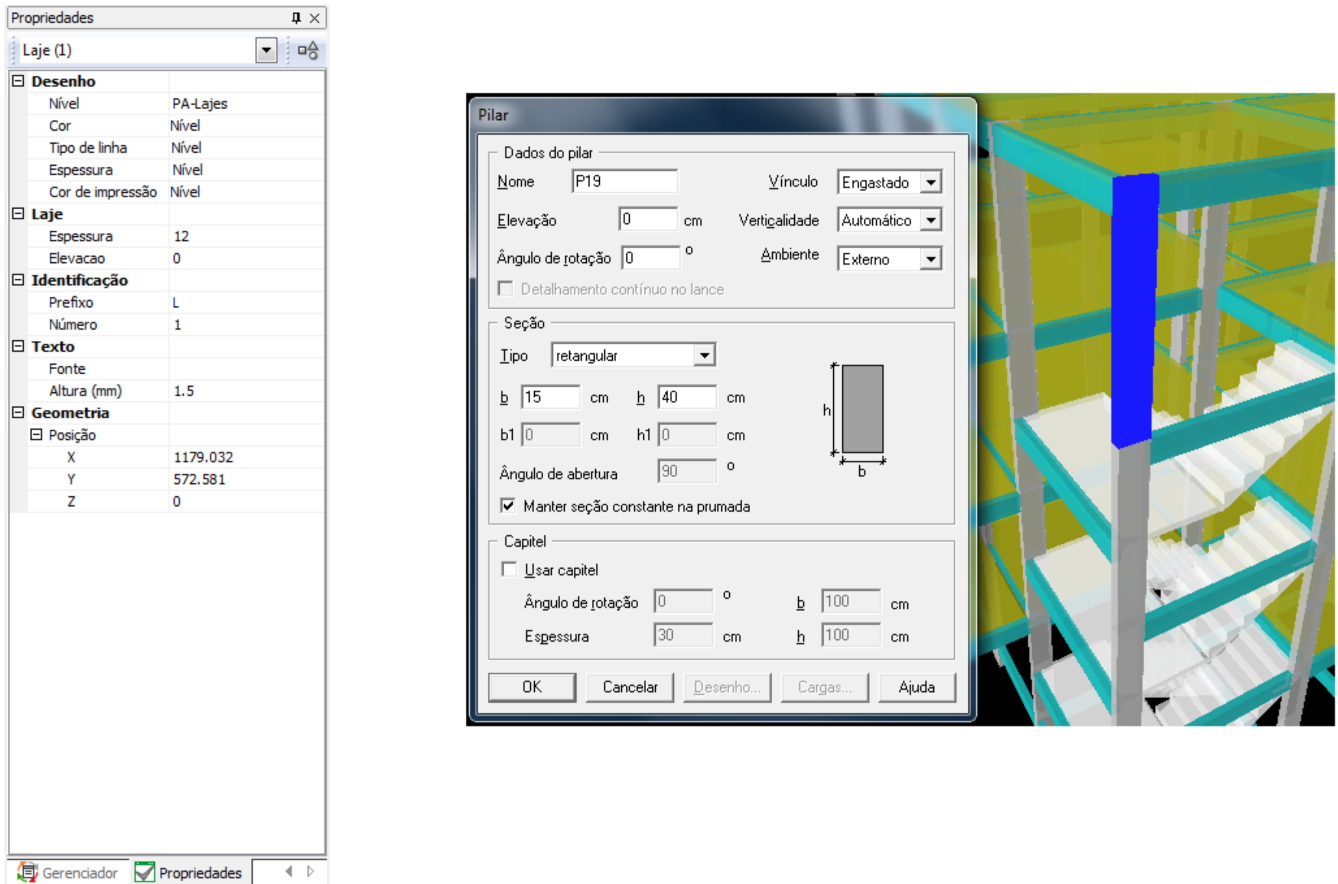


Figura 8 - Exemplos de propriedades no QiBuilder (esquerda) e Eberick (direita)

Neste grafo, cada objeto independente que representa uma entidade que pode se relacionais com outra, é chamado de “nó” do grafo. Cada “nó” tem um identificador (ID) único e suas propriedades podem ser um atributo simples ou uma referência para outro nó (atributos como estruturas complexas). Para definir um “nó” como propriedade de outro, este precisa ser “filho” do primeiro

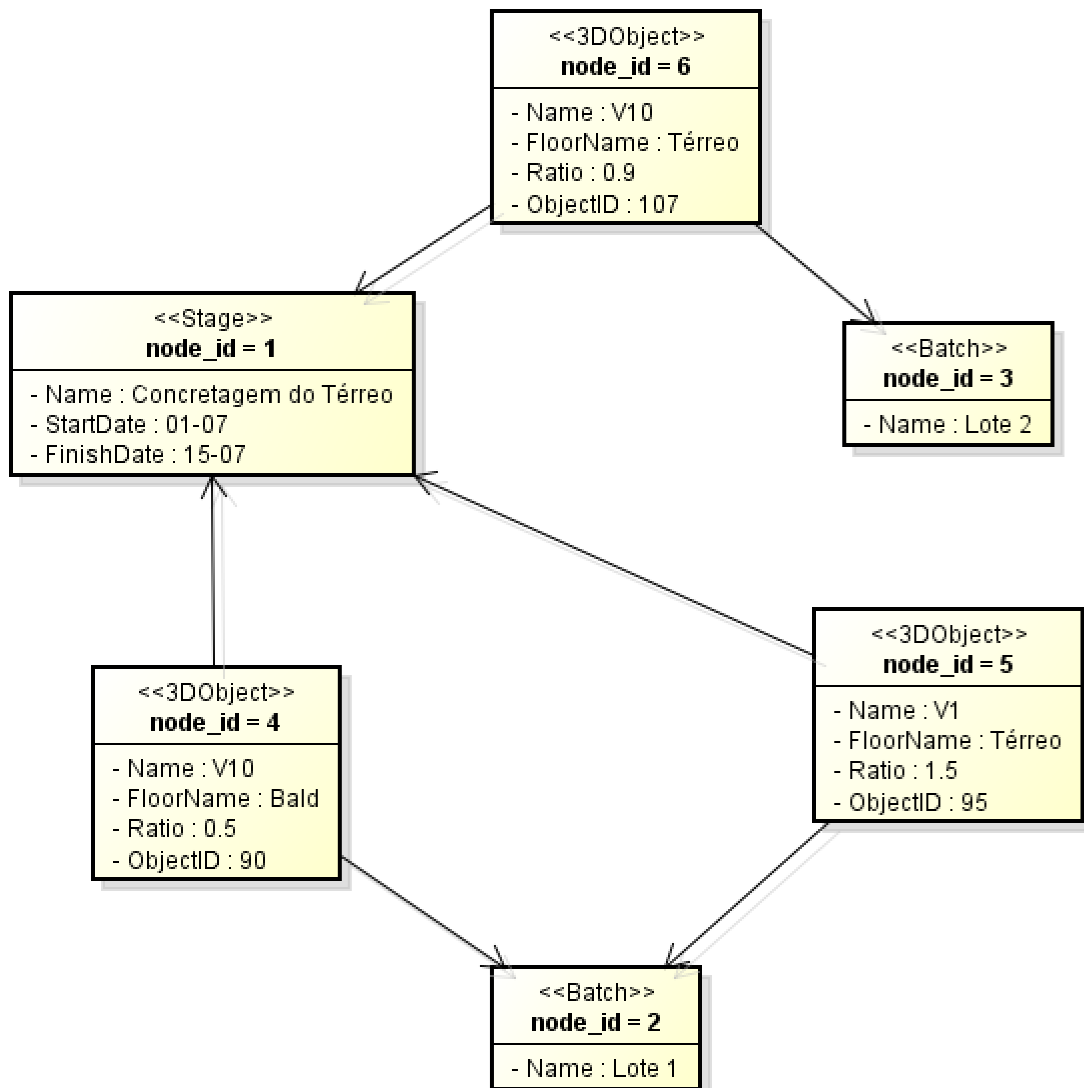


Figura 9 - Grafo acíclico dos metadados

Como exemplo, na figura acima, têm-se os nós "*3DObjects*", "*Batch*" e "*Stage*". Os atributos "*Name*" e "*FloorName*" são propriedades de cada "*3DObject*" e o nó "*Batch*" é uma propriedade de "*3DObject*" cujo identificador (node_id) é seis.

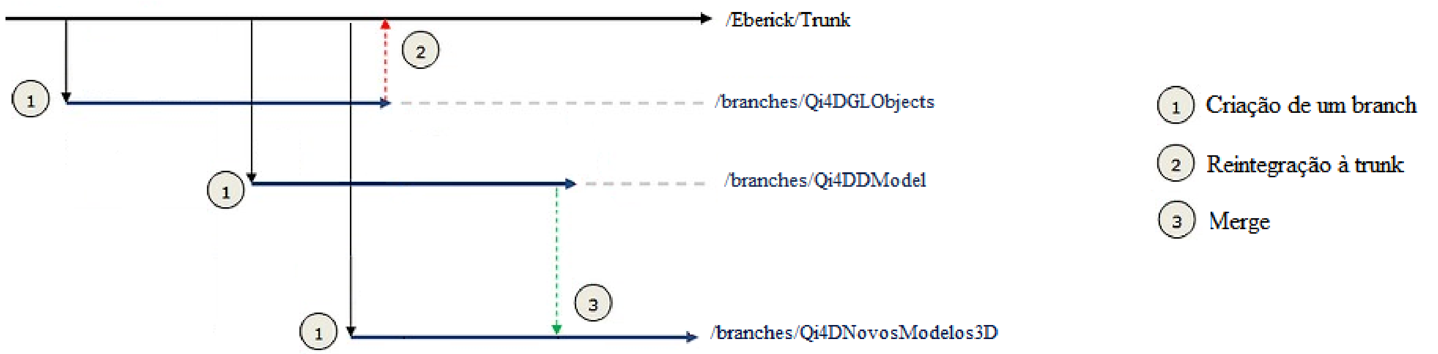
Um ponto importante de se frisar é que as aplicações da empresa continuaram o seu desenvolvimento paralelo, corrigindo *bugs* e adicionando funcionalidades previstas, pois como softwares comercias, são necessárias

constantes atualizações e inovações para continuarem competitivos. As mudanças propostas pelo projeto Qi4D são consideras de alto risco, pois podem desestabilizar os programas e gerar novos erros.

Para minimizar estes tipos de problemas, a empresa utiliza um sistema de controle de versões. Esses sistemas permitem o acompanhamento do trabalho e das modificações realizadas em um grupo de arquivos, fornecendo suporte para que vários desenvolvedores alterem os códigos do programa sem sobreposição de implementações. Além disso, é possível criar linhas de desenvolvimento paralelas com a linha original.

O sistema de controle de versões utilizado na empresa é o *Apache Subversion* (abreviado por SVN). Este baseia-se em uma metáfora de árvore para facilitar sua utilização pelos desenvolvedores. Para o SVN, a linha principal de desenvolvimento de um software é chamada de *trunk* (ou tronco, ao traduzir para o português) e as linhas paralelas são chamadas de *branches* (galhos). A todo o momento é possível criar um nova *branch* onde as modificações feitas não alteram o *trunk*, e caso seja necessário, pode-se reintegrar essa *branch* ao *trunk.* Isto é importante quando se espera grandes modificações em partes específicas de um sistema [18], como é o caso do projeto Qi4D. Assim pode-se abrir uma *branch*, realizar as modificações necessárias e após todos os testes e correções, reintegrá-la a linha de desenvolvimento principal.

O projeto Qi4D por si próprio é considerado uma linha principal de desenvolvimento. As modificações descritas nos próximos capítulos foram primeiramente aplicadas em *branchs*, tanto do Qi4D como dos programas principais da empresa, e depois reintegrados. A imagem abaixo ilustra essas linhas de desenvolvimento.



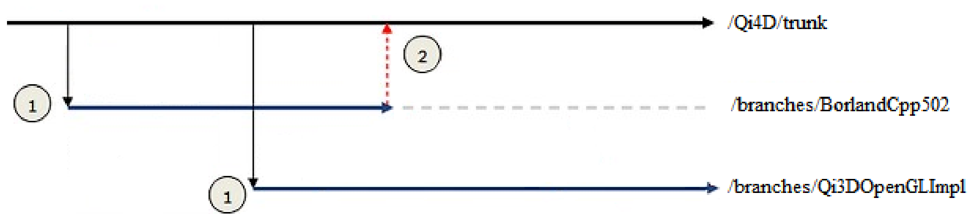


Figura 10 - Ilustração das branches criadas para a tarefa

Capítulo 5: Incorporando a API ao sistema existente

Como dito anteriormente, o PFC ocorreu durante o *Release* 3 (ou seja, a terceira parte) do projeto Qi4D. Esta parte é concentrada em estender a refatoração do Engine3D (tratado no QiBuilder durante o Release 2) também para o Eberick. Também inclui no seu escopo a primeira fase de exportação de arquivos no formato IFC e estudos sobre a persistência do modelo e aplicação de testes unitários.

Para início dos trabalhos, seguiu-se a metodologia explicada no capítulo 4, fazendo um levantamento das funcionalidades que o Eberick utilizava advindas do OpenGL. A imagem abaixo ilustra a antiga hierarquia de classes de desenho 3D, todas as classes filhas de T3DObject precisaram ser refeitas.

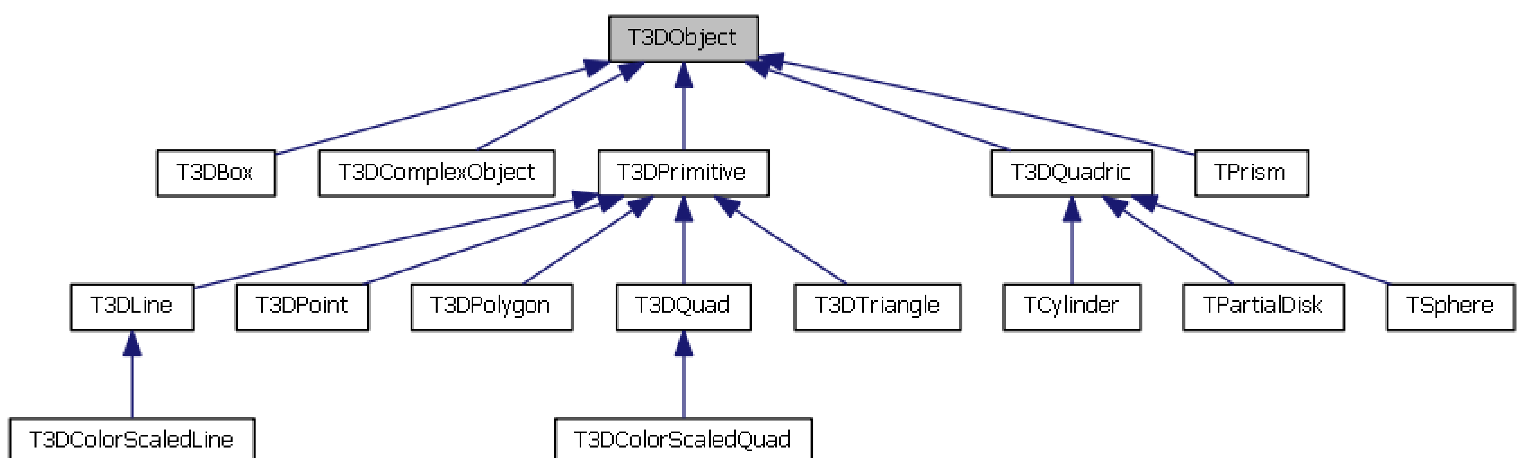


Figura 11 - Antiga hierarquia de objetos 3D no Eberick

A continuação deu-se pela criação de classes que representassem os objetos 3D OpenGL, com o único objetivo de saberem se desenhar no contexto do OpenGL, de forma independente das primitivas Qi4D originais. Essas classes compartilham de uma interface comum, chamada IQi3DGLObject, que define os métodos básicos necessários, permitindo o uso da pattern de injeção de dependência quando necessário. Esse objetos OpenGL ficaram organizados na seguinte estrutura:

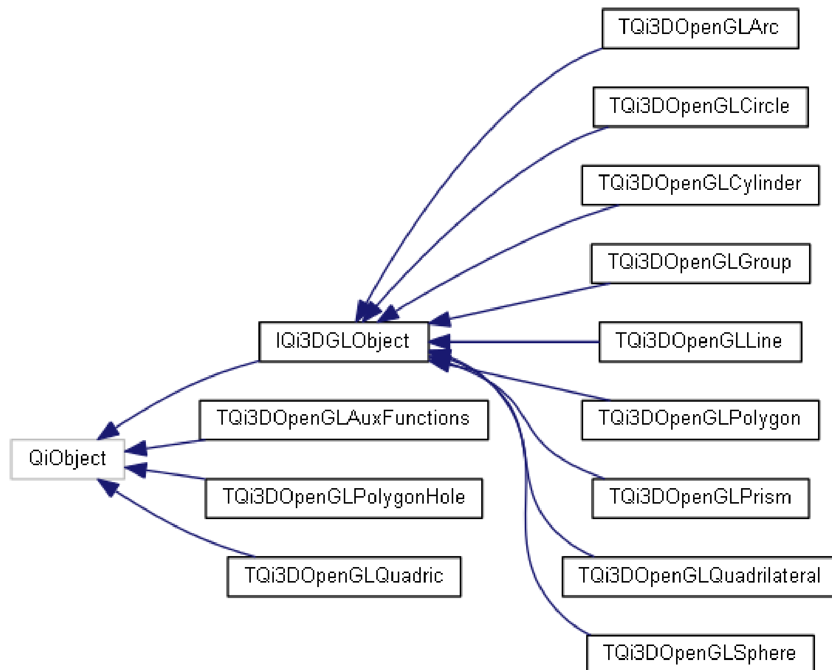


Figura 12 - Hierarquia dos objetosOpenGL no Qi4D

Como esperava-se que esta implementação resultasse num módulo independente e intercambiável entre os programas da empresa, passou-se para a criação de classes conversoras, com o intuito de converter as primitivas Qi3DObject (Qi4D) nos objetos 3D OpenGL criados. Esses conversores foram desenvolvidos seguindo os *design* patterns *Dispatcher* e *Builder*. Como pode-se ver nas figuras a seguir, nem todas as classes existentes no Qi4D precisaram ser incorporadas no Eberick, apenas aquelas que são efetivamente utilizadas.

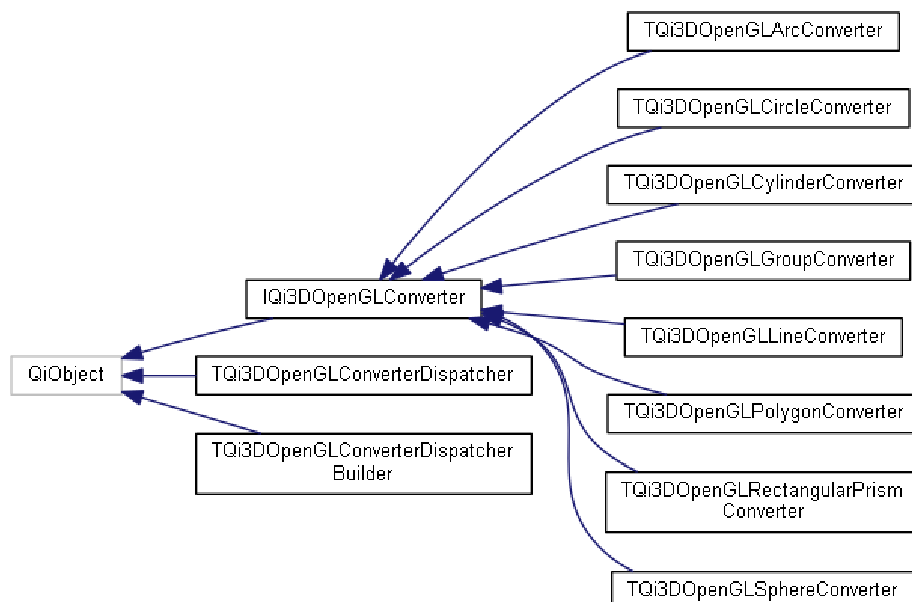


Figura 13 - Hierarquia dos conversores OpenGL no Qi4D

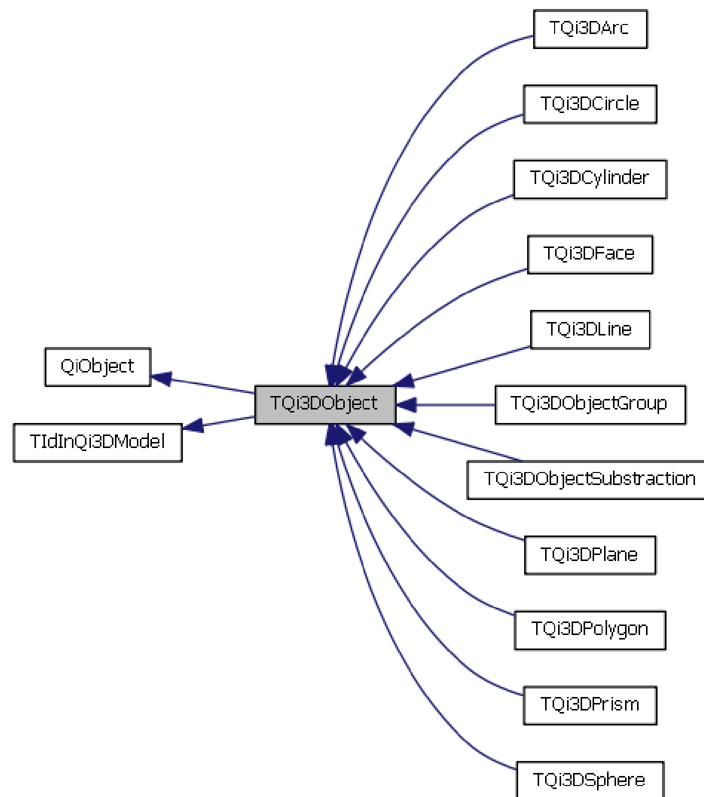


Figura 14 - Hierarquia de classes no projeto Qi4D

O *design pattern Dispatcher* [19] busca criar uma ponte entre a classe usuária do *dispatcher* e a execução de um fluxo de ações responsáveis pela geração de conteúdo genérico e dinâmico. Isto permite o desacoplamento entre o "usuário" da função e o "provedor" dos dados. O *pattern Builder* [20] se destina à separação da construção de um objeto complexo da sua representação, dessa forma, o mesmo processo de construção pode criar diferentes representações.

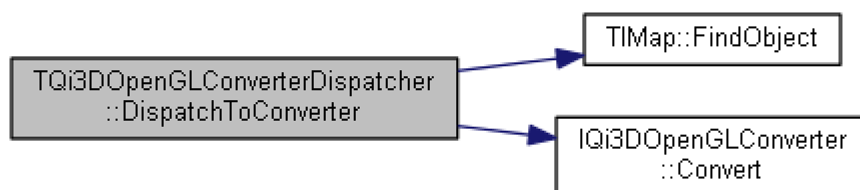


Figura 15 - Chamadas do método que busca por um conversor específico no hashmap

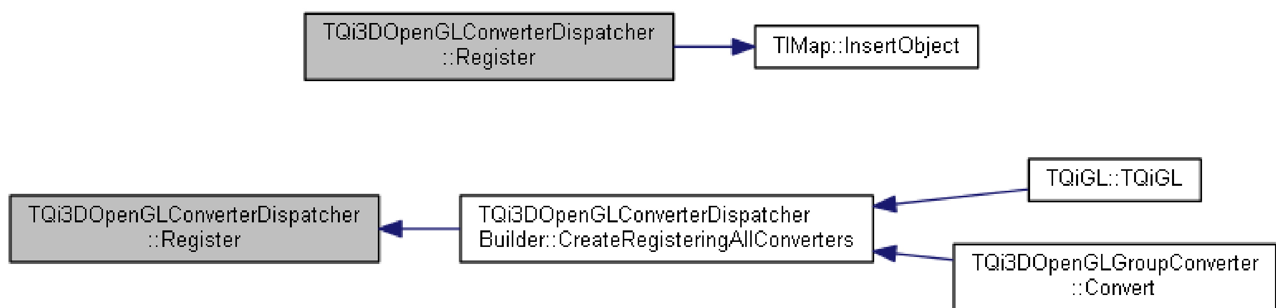


Figura 16 - Registro do conversor no hashmap ao ser criado pelo dispatcher e chamadas deste método

Ao fim dessa tarefa, obteve-se um módulo que é independente dos programas da empresa e que segue os padrões de qualidade impostos, ressaltando que sua compilação foi validada nos ambientes: Borland 5.02, Visual Studio 2006 e Borland C++Builder. Porém um dos requisitos não pode ser avaliado, a funcionalidade dinâmica com os programas em execução. Para garantir a “atomicidade” da solução, optou-se pela aplicação desse módulo criado no software Eberick da empresa, já que ele se utiliza do motor gráfico OpenGL, teoricamente, não precisando de muitas modificações para o funcionamento do novo módulo no programa.

Esta parte seguiu-se criando novos conversores, neste caso, para haver uma ponte entre os objetos já existentes no Eberick com os novos OpenGL criados. Esses conversores somente serviram para o momento de “desenho” dos elementos na janela 3D. Quando um dos elementos estava prestes a ser representado da tela, ele chamava seu conversor e então o desenho era realizado pelos novos objetos OpenGL, sem diferença com o conceito já existente no programa. Isto gerou um grande problema na performance do programa na hora da renderização dos objetos 3D, pois a cada frame era necessário destruir, converter e construir um novo objeto.

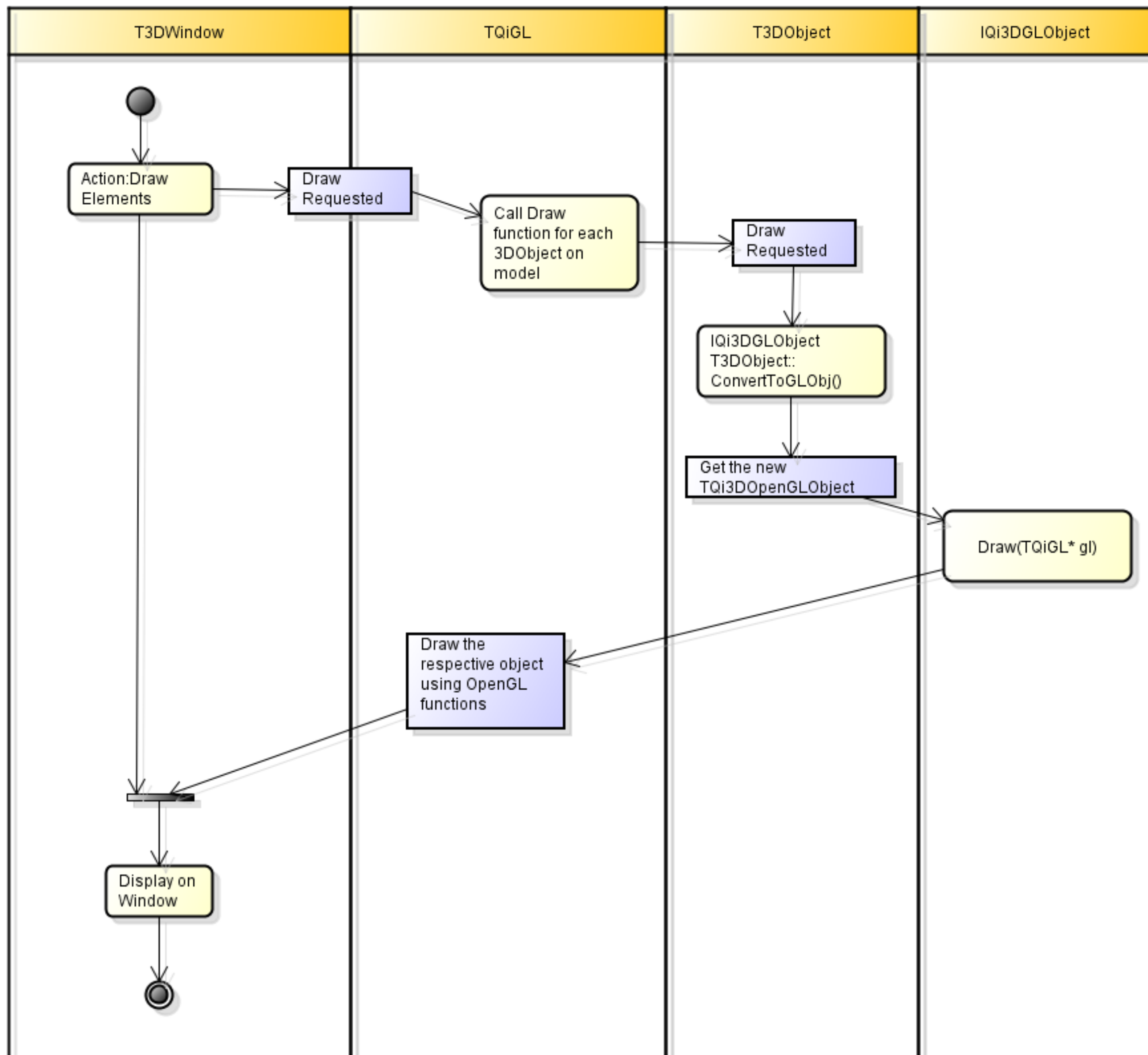


Figura 17 - Grafo de sequencia para a criação de um novo elemento a cada frame

A princípio, como o objetivo desta parte da tarefa era apenas avaliar a funcionalidade das novas primitivas *OpenGL* criadas, isto foi alcançado. Porém optou-se pela reintegração deste *branch* à *trunk* principal de desenvolvimento, e a queda de desempenho não é aceitável. Para resolver este novo problema, resolveu-se persistir os novos objetos *OpenGL* no programa, fazendo com que eles só fossem recriado quando houvessem alterações nas suas características. Neste momento, a interface criada anteriormente foi de suma importância, pois pode-se adotar o padrão de injeção de dependências para o novos objetos. Novamente, as modificações foram realizadas com sucesso e foi possível reintegrar essas mudanças a *trunk* do programa.

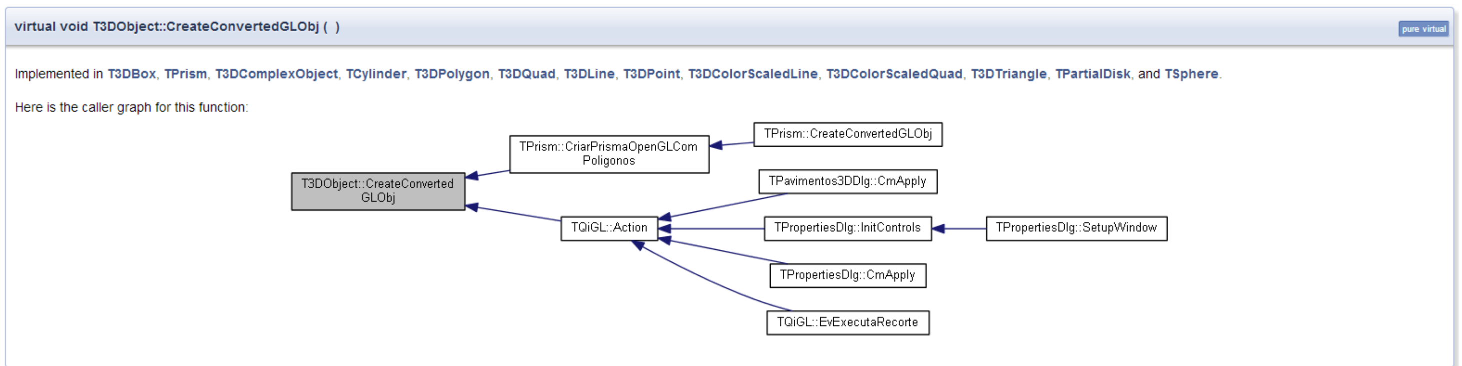


Figura 18 - Grafo dos métodos que requisitam pela criação de um novo elemento convertido e as classes que o implementam

O passo seguinte foi efetivamente substituir todos os elementos 3D gerados pelo software pelos elementos da *framework* Qi4D. O Eberick trabalhava com as seguintes classes gerando elementos 3D e suas particularidades:

- Trechos: geram vigas curvas, retas e inclinadas com diferentes seções, além de furos nas vigas;
- Lajes: geram as lajes, lajes com aberturas e rampas, com ou sem nervuras;
- Pilar: geram os pilares com suas respectivas variações de seção;
- Escadas: geram rampas, escadas com fundo plano e escadas plissadas;
- Fundação: geram os pilares de fundação, blocos, sapatas, tubulões e estacas, com diferentes seções;
- Elemento Nó: responsável pela geração do capitel;
- Muro: geração do muro em alvenaria;
- Parede: geração da parede em alvenaria;
- Barras: geração do pórtico unifilar e grelha das lajes.

Aproveitando essa necessidade de mudança, também decidiu-se implantar um *design pattern* de comportamento, o *Visitor* [21]. Este *design pattern* busca oferecer um novo comportamento para uma classe sem alterá-la. O propósito primário do *Visitor* é abstrair uma funcionalidade que será aplicada numa hierarquia agregada de objetos diferentes, possibilitando a criação de classe mais leves e flexíveis. Além disso, há a implementação do conceito "*double dispatch*", onde a operação a ser executada depende do nome da requisição e do tipo dos dois receptores (no caso, o tipo da classe *Visitor* e o tipo da classe a ser visitada).

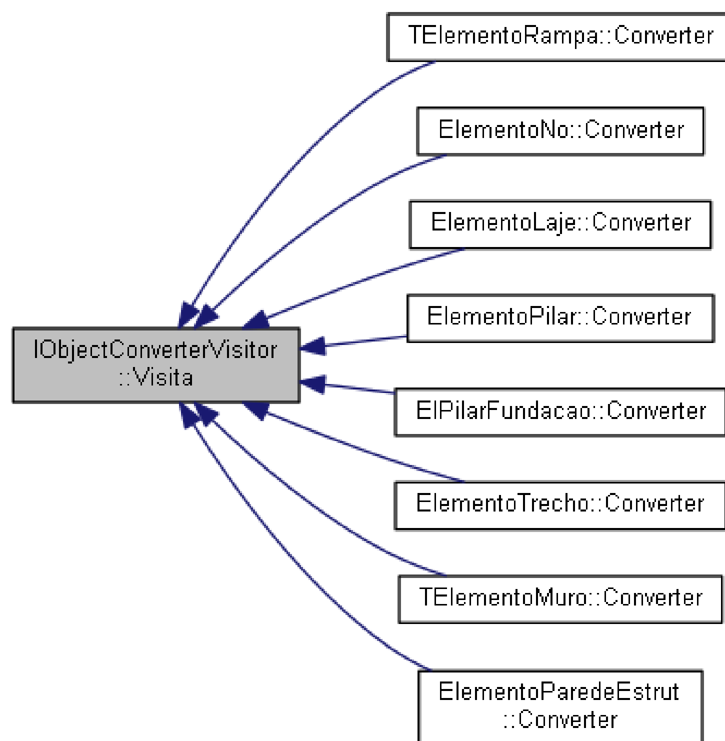


Figura 19 - Elementos que aceitam a visita

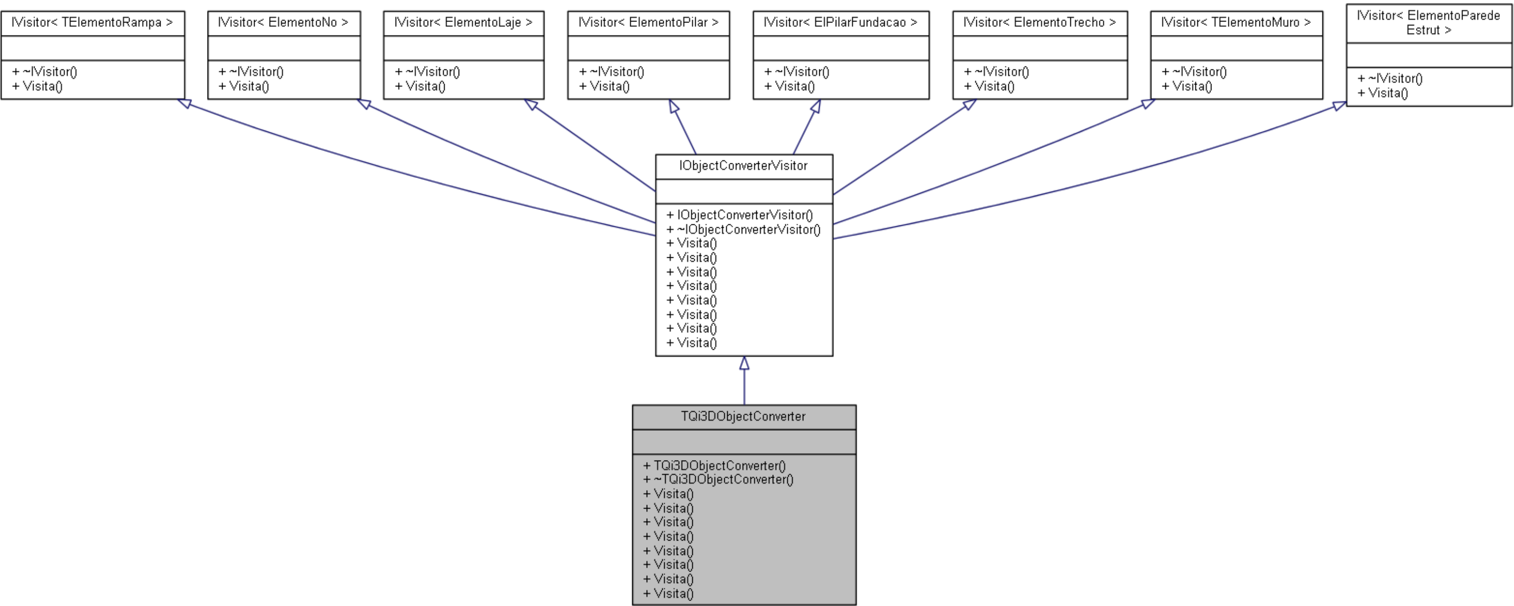


Figura 20 - Diagrama de classes dos visitor

Complementando o pattern *Visitor*, foi também implementado o *pattern Factory*, um *pattern* de criação de objetos [22] , onde define-se uma interface para criação dos objetos, mas permitem às subclasses decidirem quais objetos instanciarem, consequentemente, auxiliando na padronização do modelo de arquitetura.

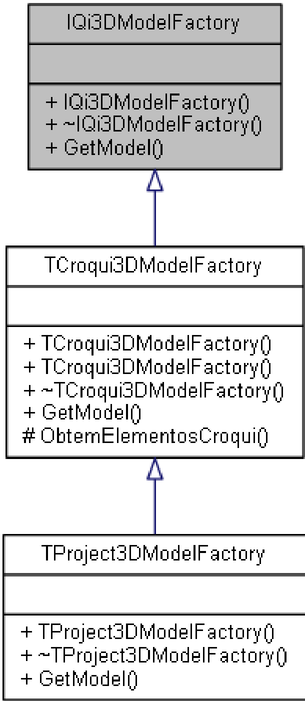


Figura 21 - Diagrama de classes das factories

Completando a nova forma que as informações são organizadas e manipuladas no programa, criou-se a estrutura de metadados citada anteriormente. Em particular, as primitivas tridimensionais do Eberick utilizam os seguintes tipos de informações:

- Nome do elemento: cada elemento tridimensional tem um nome único, que é o mesmo associado a sua entidade existente no croqui de desenho;
- Estado de exibição: é um atributo booleano que indica se o elemento deve ser exibido na janela 3D no momento ou não;
- Pavimento associado: como cada croqui representa um pavimento, esta informação também deve estar presente nos elementos 3D;
- Nome do grupo: cada elemento necessita saber a qual grupo ele pertence, por exemplo, se o elemento pertence a grupo de lajes, rampas ou escadas;
- Nível do pavimento: fornece a informação de continuidade do elemento entre pavimentos.

Utilizando o conceito do grafo acíclico explicado anteriormente, os pavimentos e grupos serão estruturas complexas de dados, além de serem partilhadas entre elementos diferentes, desta forma, serão associados, como filhos, ao metadados dos objetos tridimensionais.

A criação dos tipos de propriedades de metadados foi implementada utilizando-se de um *design pattern* de criação, o *Singleton*[23]. Este *pattern* em especial tem como caraterística marcante garantir que uma determinada classe tenha somente uma instância, fornecendo pontos de acesso a ela. Esta estratégia foi adotada para encapsular a forma como os metadados são criados e garantir que os mesmos tipos de elementos tenham os mesmos formatos de dados, apenas com valores diferentes.

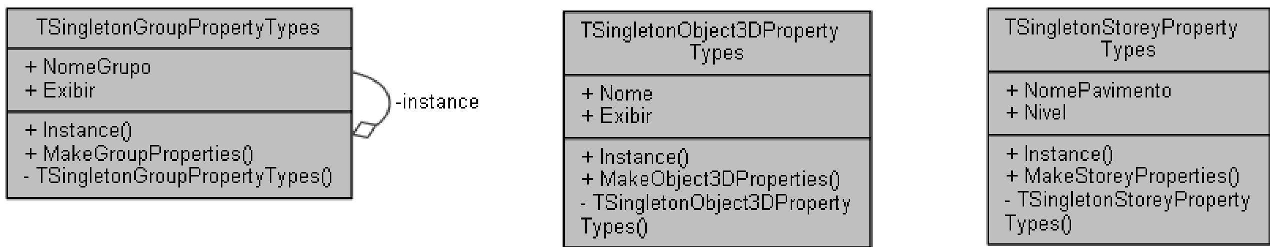


Figura 22 - Classes singletons para criação dos metadados

Complementado as funcionalidades necessárias, uma classe do tipo *Factory* foi criada para que, através do polimorfismo, possa encapsular a lógica de criação dos metadados para cada elemento estrutural do Eberick que cria seu respectivo objeto 3D.

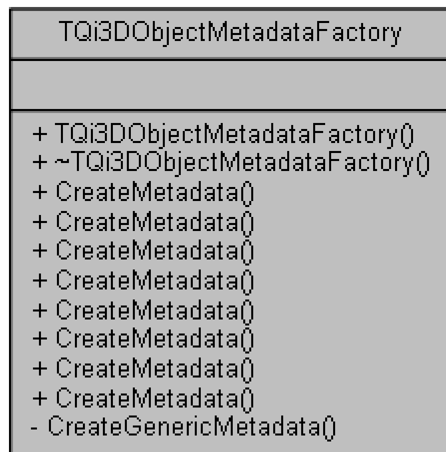


Figura 23 - Factory de metadados

Capítulo 6: Resultados Obtidos

Uma mudança que foi consequência do projeto, porém é de suma importância, é a validação do Eberick com as primitivas próprias da empresa presente no modulo QiPrimitives. Este módulo é utilizado no programa QiBuilder da empresa, e como foi desenvolvido numa plataforma bem mais atual, temia-se que não fosse retro compatível com o código legado. Porém com alguns esforços conseguiu-se validar sua solução e compilação na versão 5.02 do compilador Borland.

Este sucesso, além da importância vital para o projeto, já que o novo visualizador tridimensional é escrito sob estas primitivas, também estendeu as soluções possíveis de serem utilizadas no Eberick. Somando-se a isso, prioriza-se pela utilização destas primitivas para a criação de novas funcionalidades e em momentos de refatoração de código, para que em tempo hábil, seja possível substituir totalmente estas primitivas do Eberick, tornando o modulo citado outra parte da interoperabilidade entre estes sistemas.

Seguindo adiante, outra mudança não tão facilmente mensurável foi a refatoração direta do código. Ao reorganiza-lo e reescreve-lo sob os requisitos de qualidade do projeto Qi4D e do *clean code*, obteve-se um código mais legível, com um entendimento mais intuitivo e consequentemente facilitando sua manutenção. Não foi citado anteriormente, porém todo o código presente na *framework* Qi4D deve ser escrito em inglês, assim o novo código está nesta língua, aumentando o leque de possibilidades para "*outsourcing*" das soluções necessárias na empresa, sem precisar abrir totalmente seu código a terceiros.

Um dos objetivos principais, a redução da complexidade ciclomática e reestruturação do programa dentro das métricas de qualidade, foi alcançado com sucesso. O grafo de Kiviat abaixo ilustra como estavam os valores base antes das alterações, comparando-os com as métricas impostas:

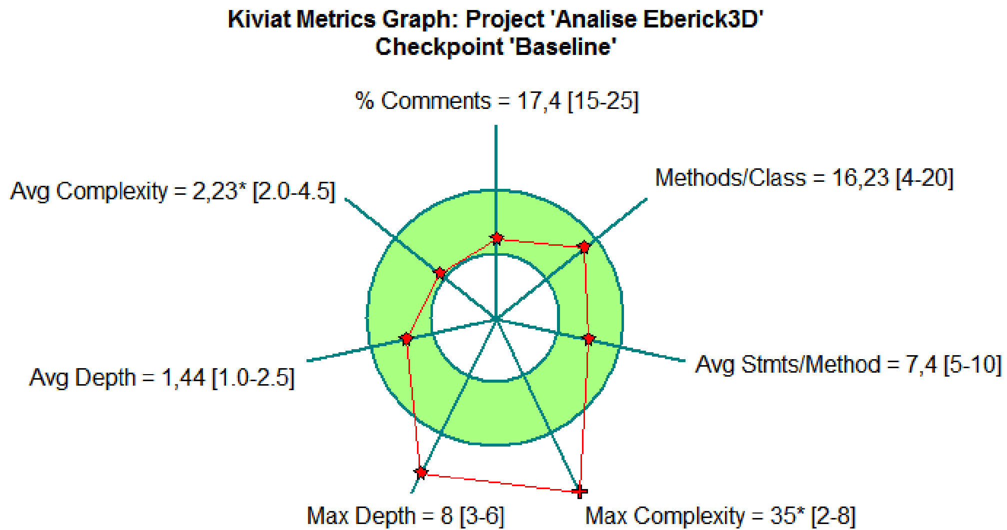


Figura 24 - Complexidade ciclomática antes das mudanças

E como pode ser visto na próxima figura, os valores obtidos para o projeto estão dentro do desejado.

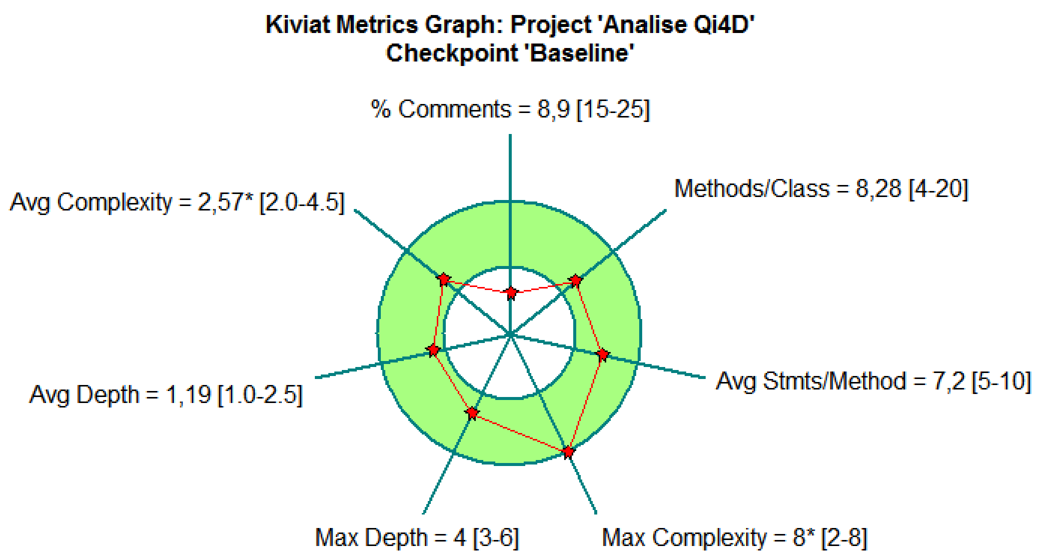


Figura 25 - Complexidade ciclomática após as alterações

Foi possível obter o desacoplamento das primitivas 3D desejadas. As antigas primitivas derivadas de *T3DObject* (vide figura 11) foram completamente removidas do código, assim como suas estruturas de dados associadas.

Os padrões de projeto aplicados contribuíram fortemente para a modularização da solução, ao concentrar a geração dos objetos 3D num único ponto, facilita-se a manutenção do código e possibilidade de melhorias futuras.

No decorrer do projeto, observou-se a necessidade da criação de uma interface para os motores gráficos utilizados. Isto, além de ser arquiteturalmente mais correto, reduziu a complexidade ciclomática do modulo Qi3DOpenGL, pois anteriormente nele estava incluso a antiga implementação concreta da engine OpenGL.

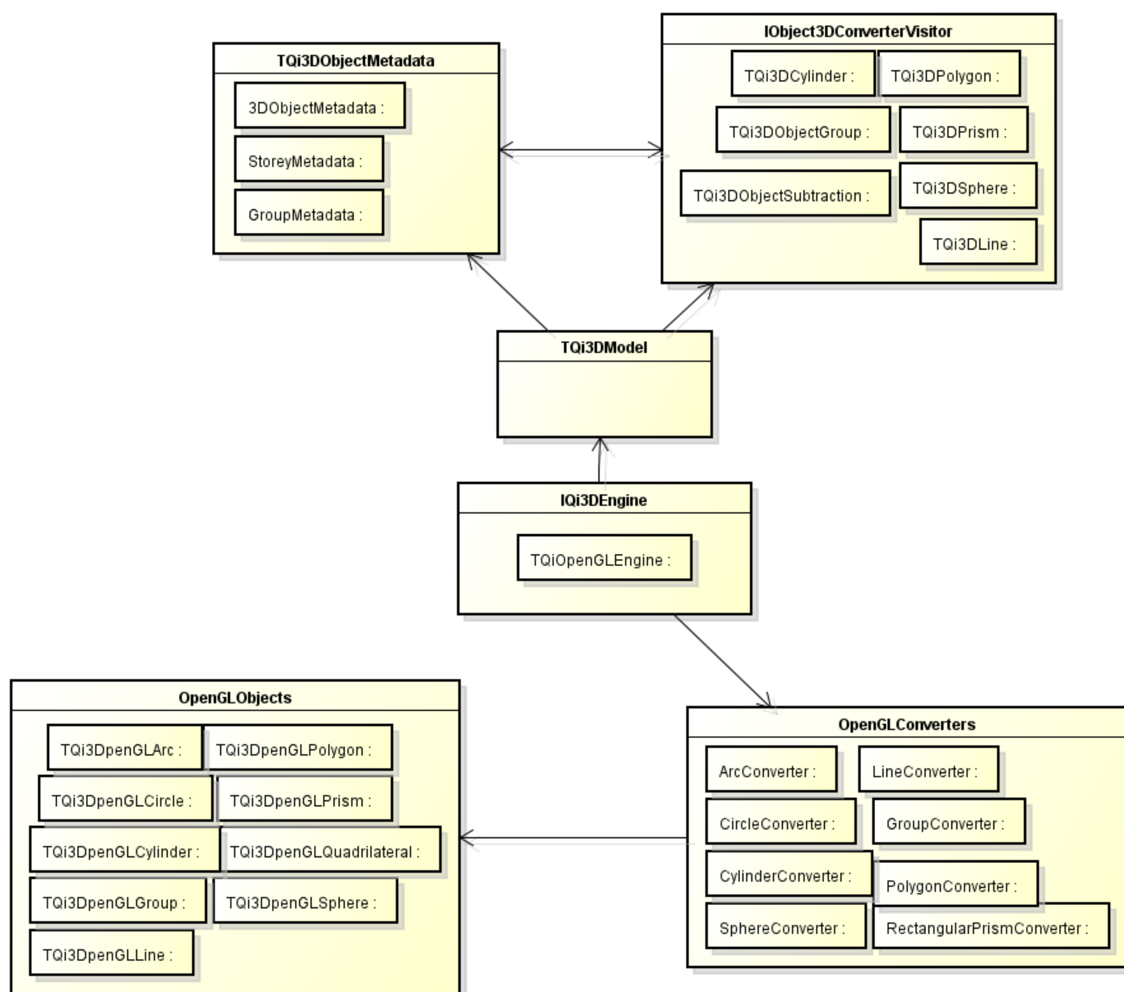


Figura 26 - Ilustração dos módulos desacoplados

A criação dos elementos OpenGL e seus respectivos conversores também foram aprovados na geração automática de versões pelo servidor de *Builds* contínuas. A figura 27 aponta a duplicação de código encontrada, na versão #210 e corrigida em seguida, na versão #211, como mostrada na figura 28.

Figura 27 - Erro de duplicação de código na geração continua de versões

Figura 28 - Nenhum erro encontrado na geração continua de versões

Para a conformidade da aplicação das primitivas da API no programa Eberick não pode-se validar uma melhoria quantitativa para a refatoração através dos grados de Kiviat, pois o código antigo estava espalhado por todo o programa, “contaminando e sendo contaminado” por outras partes, logo o resultado não refletiria o real cenário da aplicação. Para a nova refatoração, procurou-se manter as lógicas existentes, apenas alterando o que fosse necessário para as novas primitivas. Essa escolha foi tomada, pois as estruturas para desenhos 3D existentes são complexas e extensas, uma

refatoração total tomaria muito mais tempo que o disponível para ser concluída, além de possivelmente levar a novos erros inesperados, que necessitariam mais correções. Como ilustração, o grafo abaixo mostra o estado atual da classe responsável por gerar as primitivas tridimensionais. É possível ver que para este caso está ultrapassando as métricas de qualidade, mas foi aceito como válido devido às limitações citadas.

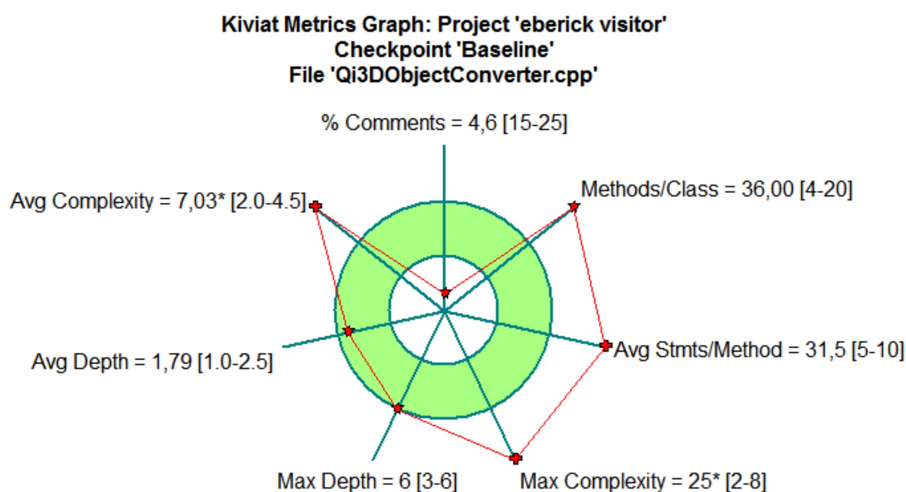


Figura 29 - Grafo de Kiviat para classe responsável pela criação dos novos elementos 3D

De modo como foi apresentado no capítulo 3, as modificações passaram pela bateria de testes dinâmicos. Como as modificações aconteceram em uma *branch* do programa, foi possível criar uma tarefa no servidor de geração de *builds* para gerar versões com estas alterações. O próprio sucesso na geração da versão é um teste, pois para isso o programa necessita ser completamente recompilado sem erros. Como é possível ver na figura 30, alguns "*bugs*" ocorreram durante o desenvolvimento da tarefa, estes então são corrigidos e passados para novos testes dinâmicos, a tarefa é dada por concluída quando todas as funcionalidades desejadas estão implementadas e todos os erros corrigidos. Quando todas as tarefas de um projeto foram concluídas, ele é encerrado.

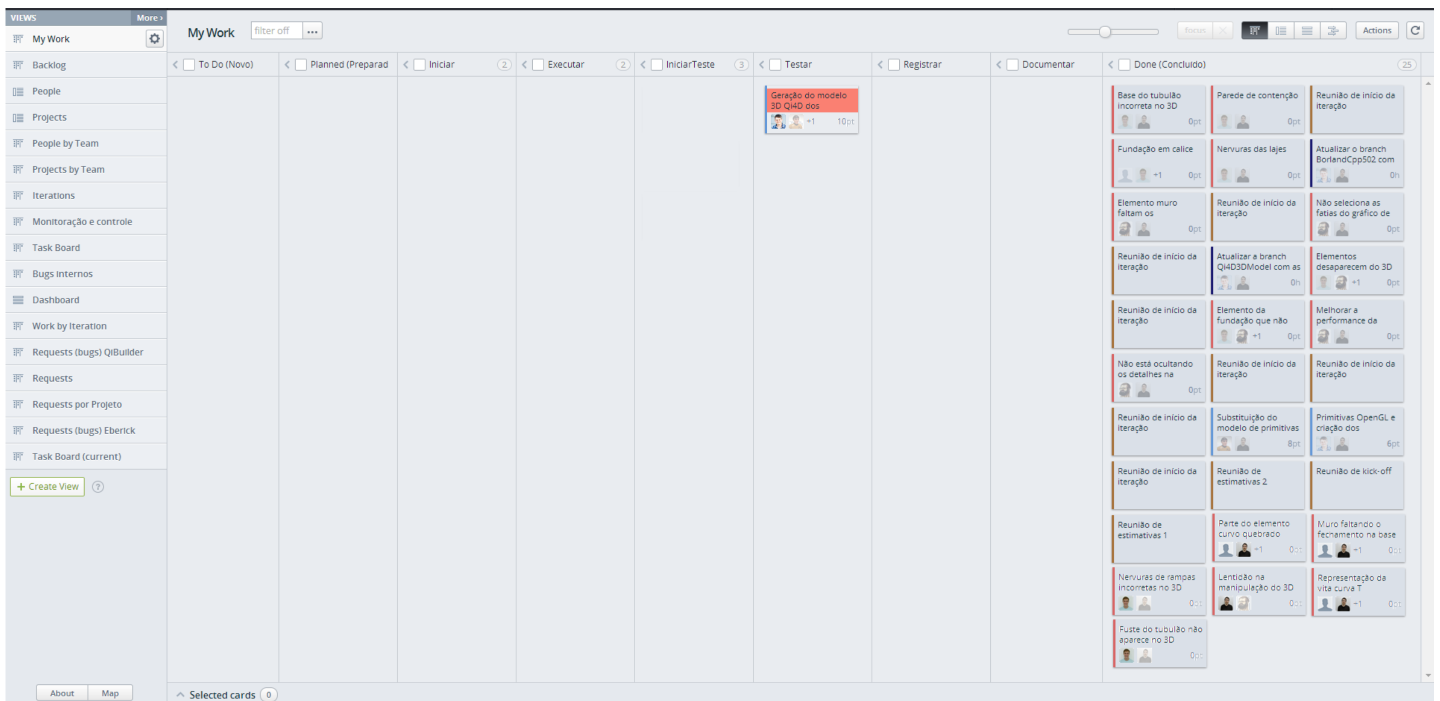


Figura 30 - Ferramenta de gerenciamento de projetos

Por fim, foi possível associar a nova estrutura de metadados aos novos elementos 3D, até o momento da conclusão deste trabalho, o visualizador estava gerando todos os elementos existentes no programa sem erros e as funcionalidades de manipulação do modelo, troca de cores e transparência via diálogo haviam sido reestabelecidas. Também já haviam sido concluídos a visão dos elementos no modo “*wireframe*”, a opção de desativar a visibilidade de um grupo de elementos e a estrutura dos planos de cortes.

Capítulo 7: Conclusões e Perspectivas

Os ideais de automação de sistemas e tarefas se mesclam com as técnicas e conceitos da Tecnologia da informação. Os softwares criados pela empresa para auxiliar os profissionais da área de construção civil na criação, execução e gerenciamento de seus projetos são mais um dos exemplos disso, além de reforçar que a automação vai muito além do chão de fábrica.

Com a evolução tecnológica e social, estes mesmos softwares criados para facilitar as tarefas acabaram encontrando um novo problema, cada vez mais comum na área de automação de sistemas: a falta de interoperabilidade entre eles. O que é demonstrado por várias características comuns nestes casos: as tecnologias utilizadas não são totalmente compatíveis entre os softwares, as terminologias são diferentes, a arquitetura utilizada no desenvolvimento está defasada, os sistemas não mostram seus dados e resultados nas novas plataformas e um dos softwares já pode ser considerado um sistema legado.

Para tentar combater estes problemas e fornecer uma maior interoperabilidade, buscou-se adotar um padrão aberto para troca de informações, e o padrão que está em voga na área de construção civil é o IFC.

A adequação dos produtos desenvolvidos nesse padrão foi custosa, tomou tempo e havia o risco de não obter-se os resultados esperados. Vale lembrar que os softwares desenvolvidos são produtos comerciais finais, que precisam sempre de novas funcionalidades para garantir sua permanência no mercado, logo, ter uma equipe destinada exclusivamente à refatorações e adequações internas dos produtos é estrategicamente sensível.

O projeto mostrou-se mais extenso e complexo que o previsto, havendo uma extensão de aproximadamente dois meses a mais que o estimado. Entre os fatores que contribuíram para isso, podem-se destacar:

- As adequações necessárias ao código fonte das novas funcionalidades ao tenta-las portar para o compilador Borland 5.02;
- A necessidade de retrabalhado duplo para erros na geração de elementos pelo motor gráfico *OpenGL*. Isto acontece pois foi decidido manter suporte para duas versões da linguagem gráfica, já que alguns usuários não possuem o hardware mínimo para a utilização de novas soluções 3D.
- Expansões necessárias e não previstas de funcionalidades na API para adequar-se com problemas encontrados durante as refatorações nos programas. Em alguns casos, mudanças realizadas em métodos já existentes resultavam no retrabalho dos dois programas.

Os esforços desprendidos resultaram no sucesso de aplicar o novo padrão, o projeto realizado também conseguiu desacoplar partes significativas dos sistemas, fornecendo uma maior vida útil a eles. Algumas funcionalidades não estavam totalmente implementadas até o fim do PFC. Podem-se citar:

- O caso de edição de elementos dentro do visualizador, pois a lógica utilizada antigamente não pode ser replicada devido à nova utilização da estrutura de metadados;
- A importação e exportação do modelo IFC dentro do Eberick, apesar da estrutura de dados estar adequada para isto, ainda necessita-se criar uma interface GUI e adequar o código para estes novos métodos da API. Para o QiBuilder, ainda ocorrem algumas bugs na importação/exportação, que estão em fase de correção.

Apesar disto, as mudanças forneceram um precedente para futuros projetos que busquem aumentar esta modularização dos sistemas. Somou-se a isso a isso a criação de uma framework própria e compartilhada entre os softwares, que pode ser expandida para as novas funcionalidades desejadas com certa garantia de sucesso, fornecida pela conclusão positiva do projeto.

Como perspectivas futuras, o projeto do Qi4D deveria continuar sua expansão para os ambientes *web* e *mobile*, como desejado. Sobretudo, também é necessário continuar com esse conceito de desacoplamento e modularização dos softwares, para que em tempo hábil seja possível implantar novas soluções tecnológicas sem grandes custos e num escopo temporal factível para o mercado. Ainda há muito trabalho a ser feito para chegar neste ponto, mas a empresa está seguindo pelo caminho certo.

Bibliografia:

[1] AYRES, Marcela. “Indústria da construção civil deve crescer 2,8% em 2014”. Disponível em http://exame.abril.com.br/economia/noticias/industria-brasileira-de-construcao-civil-deve-crescer-2-8-em-2014, acesso em 17 maio 2014.

[2]CBIC. “Balanço Nacional da Indústria da Construção”. Disponível em http://www.cbicdados.com.br/menu/estudos-especificos-da-construcao-civil/balanco-nacional-da-industria-da-construcao, acesso em 17 maio 2014.

[3]WIKIPEDIA. “BIM”. Disponível em http://pt.wikipedia.org/wiki/BIM, acesso em 18 maio 2014.

NATIONAL BIM STANDARD. “What is a BIM?”. Disponível em http://www.nationalbimstandard.org/faq.php#faq1, acesso em 18 maio 2014.

[4]COORDENAR – CONSULTORIA DE AÇÃO. “O IFC é muito mais que um simples formato de arquivo”. Disponível em http://www.coordenar.com.br/o-ifc-e-muito-mais-que-um-simples-formato-de-arquivo/, acesso em 18 maio 2014.

[5]ALTOQI. “Wiki AltoQi”. Disponível em http://wiki.altoqi.com.br/, primeiro acesso em mar. 2014.

[6]EASTMAM, Chuck; TEICHOLZ, Paul; SACKS, Rafael; LISTON, Kathleen. “BIM Handbook: A guide to Building Information Modeling for Owners, Managers, Designers, Engineers, and Contractors”. New Jersey: John Wiley & Sons, 2008.

[7]PINHO, Sérgio Miguel Ferreira de. “O modelo IFC como agente de interoperabilidade”. Mestrado Integrado em Engenharia Civil - 2012/2013, Departamento de Engenharia Civil, Faculdade de Engenharia da Universidade do Porto, Porto, Portugal, 2013.

[8]WIKIPEDIA. “4D BIM”. Disponível em http://en.wikipedia.org/wiki/4D_BIM, acesso em 12 jul. 2014.

[9]3SCALE – infrastructure for the programmable web. “What is an API? Your guide to the Internet Business (R)evolution”. Disponível em http://www.3scale.net/wp-content/uploads/2012/06/What-is-an-API-1.0.pdf, acesso em 11 jul. 2014.

[10]MARTIN, Robert C. “Clean Code: A handbook of agile software craftsmanship”. Prentice Hall, 2008.

[11]WIKIPEDIA. “Design structure matrix”. Disponível em http://en.wikipedia.org/wiki/Design_structure_matrix, acesso em 24 maio 2014.

[12]QUICOLI Paulo. “O padrão MVP (Model – View – Presenter)”. Disponível em http://www.devmedia.com.br/o-padrao-mvp-model-view-presenter/3043, acesso em 24 maio 2014.

[13]MCCABEE T.J.; WATSON A. H. “Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric”. National Institute of Standards and Technology Special Publication 500-235, September 1996.

[14]SOFTWARE ENGINEERING INSTITUTE. “C4 Software Technology Reference Guide – A Prototype”. Carnegie Mellon University, pp. 145-147, jan. 1997.

[15]SOMMERVILLE, Ian. “Software Engineering”, Addison Wesley, 7ª edição, 2014.

[16]PIRES, Eduardo. “DDD, TDD, BDD, Afinal o que são essas siglas?”. Disponível em http://eduardopires.net.br/2012/06/ddd-tdd-bdd/, acesso em 25 maio 2014.

[17]ANDRADE, Max Lira Veras X. de; RUSCHEL, Regina Coeli. “Interoperabilidade de aplicativos BIM usados em arquitetura por meio do formato IFC”. Gestão & Tecnologia de Projetos, Brasil, v. 4, n. 2, p. p.76-111, jan. 2010. ISSN 1981-1543. Disponível em: <http://www.revistas.usp.br/gestaodeprojetos/article/view/50960>, acesso em: 7 Jun. 2014.

[18]MARTINS, André. “SVN: conceitos, boas práticas e dicas de utilização”. Disponível em http://intentor.com.br/svn-conceitos-boas-praticas-dicas-de-utilizacao/ , acesso em 7 jun. 2014.

[19]DOAN Duy Hai. “Design Pattern: the Asynchronous Dispatcher”. Disponível em http://doanduyhai.wordpress.com/2012/08/04/design-pattern-the-asynchronous-dispatcher/, acesso em 14 jun. 2014.

[20]FREY Gerhard; PAVLOVA, Marina; SHVETS, Alexander. “Builder Design Pattern”. Disponível em http://sourcemaking.com/design_patterns/builder, acesso em 14 jun. 2014.

[21]FREY Gerhard; PAVLOVA, Marina; SHVETS, Alexander. “Visitor Design Pattern”. Disponível em http://sourcemaking.com/design_patterns/visitor, acesso em 14 jun. 2014.

[22]FREY Gerhard; PAVLOVA, Marina; SHVETS, Alexander. “Factory Method Design Pattern”. Disponível em http://sourcemaking.com/design_patterns/factory_method, acesso em 14 jun. 2014.

[23]FREY Gerhard; PAVLOVA, Marina; SHVETS, Alexander. “Singleton Design Pattern”. Disponível em http://sourcemaking.com/design_patterns/singleton, acesso em 21 jun. 2014.