Archimedes: Um CAD Livre desenvolvido com programação extrema e orientação a objetos

Hugo Corbucci
Mariana Vivian Bravo

Orientador: Fabio Kon

Sumário

1 Introdução

O projeto Archimedes nasceu sem saber em 2002 por um pedido de Jupira Corbucci, ao descobrir a existência de diversos programas abertos e gratuitos, como o Linux. Cansado dos sofrimentos vividos como usuário do Windows, ele pediu que seu filho fizesse um "AutoCAD livre" quando entrasse na faculdade. A necessidade era por um software de desenho técnico aberto voltado para arquitetura, que pudesse ser usado para elaboração de plantas detalhadas e rodasse em diferentes sistemas operacionais. A conversa parou por ali já que faltava muito conhecimento para poder iniciar tamanho projeto.

Anos depois, no FISL[1] 6.0, que ocorreu entre 01/06/2005 e 04/06/2005, houve uma palestra de Jon 'Maddog' Hall sobre pirataria de software. Ao final desta palestra, um tempo de perguntas foi aberto e uma delas foi[2]:

> "Como engenheiro, preciso utilizar o AutoCAD para elaboração de desenhos técnicos. Apesar das minhas buscas, não consigo encontrar soluções abertas viáveis para poder migrar para o ambiente Linux. Existe algum software aberto para desenho técnico que o senhor conheça?".

A resposta veio sob forma de um desafio[2]:

> "Infelizmente não conheço nenhum. Mas desafio qualquer um nesta sala a começar esse trabalho e trazê-lo para cá, neste mesmo evento, daqui um ou dois anos".

No jantar que seguiu a palestra, o pedido feito 3 anos antes voltou à tona e, numa discussão casual entre amigos, um pequeno grupo disposto a encarar o desafio se formou. As férias seguintes serviram para recrutar voluntários, organizar uma equipe de desenvolvimento, estabelecer uma linguagem de programação, bibliotecas gráficas e uma metodologia de trabalho e conseguir um grupo de usuários dispostos a ajudar. Também foram reservados os espaços para o projeto no Sourceforge[3] e no Código Livre[4], entre outros.

Para estudar melhor as tecnologias escolhidas para uso no Archimedes, uma parte do grupo se reuniu para desenvolver um projeto para a matéria de *MAC0332 - Engenharia de Software*. Nesse projeto foi possível aprender as bibliotecas escolhidas e

[1]Fórum Internacional de Software Livre
[2]Tradução livre que visa retratar apenas o conteúdo essencial da declaração.
[3]http://www.sourceforge.net
[4]http://www.codigolivre.org.br

aplicar algumas práticas da metodologia a ser adotada. Assim, no fim de 2005, as tecnologias já tinham sido aprovadas devido ao sucesso do projeto e um início de interface gráfica para o Archimedes já tinha sido desenvolvido. Decidiu-se, então, aproveitar a matéria *MAC0342 - Laboratório de Programação eXtrema* para conseguir um horário fixo de trabalho dos membros do grupo. Para preparar melhor esse trabalho, nas férias, ocorreram algumas reuniões para uma análise de uma arquitetura simples do sistema, tentando entender quais seriam os problemas enfrentados e os padrões para resolvê-los.

O AutoCAD é uma ferramenta de desenho técnico baseada em comandos, isto é, funciona de maneira semelhante a um *Shell* apenas refletindo o que é feito na área de visualização. Com isso, desenhar um projeto no AutoCAD é equivalente a descrevê-lo com os comandos disponíveis. Os arquitetos usam essa ferramenta para criar desenhos precisos nos quais os elementos se unem perfeitamente. Completado o desenho, eles podem imprimi-lo com a mesma precisão e assim criar as plantas dos prédios e casas que projetam. A principal diferença do AutoCAD com outro software específico é a interface de linha de comando apresentada. Apesar de possuir diversos atalhos e ícones, os arquitetos usuários utilizam principalmente os comandos, sendo que o *mouse* é usado apenas para "apontar" elementos ou "dar *enter*".

2 Conceitos e tecnologias estudadas

Os conceitos aplicados durante o desenvolvimento do Archimedes são principalmente sobre padrões de projeto, computação gráfica, álgebra linear e geometria computacional em casos mais difíceis.

Em primeiro lugar, como o trabalho realizado foi uma implementação, os padrões de projeto e de arquitetura estão misturados no código. Para poder percebê-los melhor é necessário tomar o recuo desejado e observar o projeto de uma forma geral. Alguns padrões são facilmente observados já que Java os usa explicitamente, como o padrão *Iterator* [8] que permite percorrer qualquer tipo de coleção com a mesma interface.

Em segundo lugar, por se tratar de um tratamento vetorial às formas geométricas, muito de geometria analítica e álgebra linear é utilizado. Por exemplo, operações como mover e rotacionar usam as representações e cálculos sobre figuras geométricas nas áreas citadas para funcionarem. Nos casos em que o problema a ser resolvido é mais complexo, pode ser necessário o uso de geometria computacional. Por exemplo, ao reduzir um polígono irregular é necessário descobrir se todas as faces do polígono serão mantidas na redução ou se, por efeito de concavidade, alguma face desaparece. Neste caso é necessário identificar esta ocorrência e então corrigir a figura gerada.

Finalmente, diversos conceitos da computação gráfica se aplicam pois, ainda que seja em duas dimensões, o Archimedes lida com modelagem de sistemas de objetos e sua visualização em um dispositivo limitado. Algoritmos como o de recorte devem ser aplicados e utilizou-se uma biblioteca gráfica (OpenGL) que já lida com esse problema.

É importante ressaltar que esses tópicos não foram estudados individualmente por cada membro da equipe. Na verdade, cada um estudou seus assuntos de interesse e pôde aplicar os conhecimentos adquiridos conforme o estudo.

Além disso, a equipe estudou métodos ágeis de desenvolvimento de software e adotou em particular a programação extrema. Isso aconteceu por um conjunto de motivos. O principal foi que a equipe não conhecia nem sequer por cima o programa a ser usado de base inicial (AutoCAD), e nem fazia idéia do que os clientes esperavam do Archimedes. Então ter um cliente presente e participante se tornou uma forte necessidade. Um segundo fator foi a inexperiência dos integrantes com projetos complexos como esse, que tornava muito atraente a idéia de poder desenvolver o sistema aos poucos.

Ao longo do projeto teve-se a constante preocupação em torno de um bom ambiente de programação para manter o interesse dos envolvidos e, com ele, a dedicação. A metodologia de programação extrema foi muito favorável a este ambiente já que a grande parte do trabalho ocorria em frente a uma tela de computador em companhia de um amigo.

A prática de montar a arquitetura do sistema conforme este cresce e a de programação pareada, ambas essenciais na metodologia usada, foram muito eficientes e permitiram difundir boas práticas de programação e de *design* no modelo do sistema. Aproximadamente uma vez por mês, uma refatoração[5] [7] importante na arquitetura era identificada e implementada. Com isso, diversos padrões arquiteturais e de projeto foram inseridos no sistema quando se tornaram claros e necessários.

Para prover a portabilidade requerida mantendo a possibilidade de usar sistemas operacionais diversos, a linguagem Java foi escolhida em sua versão 1.5 [10] (também conhecida como 5.0 ou *Tiger*). Optou-se por utilizar SWT na versão 3.2 [9] para interface com usuário pois, neste estado, ela já oferecia suporte a OpenGL, a biblioteca gráfica usada para o desenho técnico por questões de desempenho. OpenGL permitiu uma maior velocidade para desenho do projeto enquanto o SWT garantiu uma aparência à interface muito mais familiar ao usuário pois utiliza as configurações nativas de cada sistema operacional. Além destas tecnologias, foram utilizadas ferramentas de desenvolvimento para aumentar a produtividade como Eclipse, Subversion e Ant.

[5] Refatoração é uma técnica descrita por Martin Fowler em Refactoring [7]

3 Metodologia

O método tradicional de desenvolvimento de software é baseado em um modelo em cascata no qual uma fase leva a outra mas cada uma deve ser feita por completo (Figura 1, página 6).

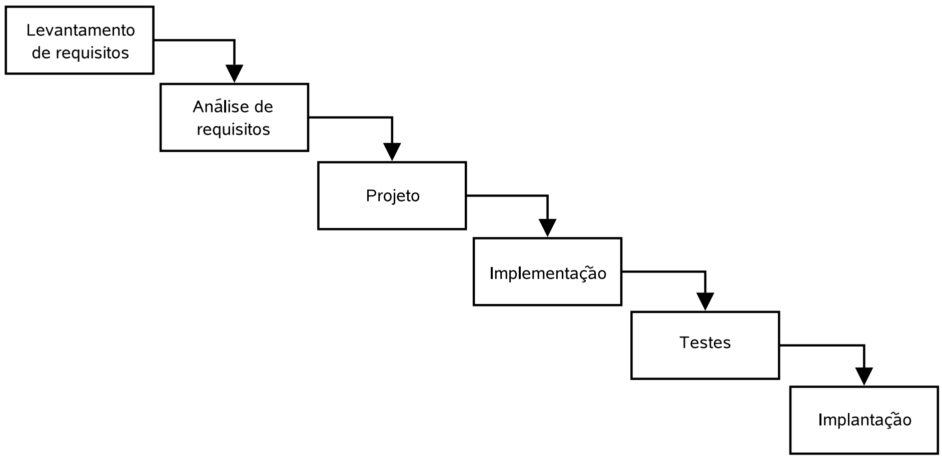

Figura 1: Modelo de desenvolvimento tradicional em cascata

O maior problema desta metodologia é que os profissionais devem planejar o projeto todo sem implementá-lo. Com isso, qualquer erro de planejamento ou mudança nos requisitos só é corrigido ou detectado na fase de testes com o sistema inteiro pronto. Isso faz com que seja muito mais difícil e custoso de lidar com esses problemas.

Programação extrema (XP) tenta resolver estes problemas unindo praticamente todas as fases. Assim, em XP, a fase de levantamento de requisitos é feita a cada 2 ou 3 semanas. As fases de análise de requisitos, planejamento de arquitetura, implementação e testes são englobadas em uma só, que é realizada todos os dias. Finalmente, a fase de produção é realizada ao final das 2 ou 3 semanas após o levantamento dos requisitos. A idéia é que os erros sejam detectados rapidamente e assim corrigidos mantendo uma maior qualidade do software.

Para que esse tipo de trabalho funcione, é preciso focar em alguns valores:

- **Comunicação** na equipe e com o cliente para que todos entendam o estado atual do projeto.
- **Simplicidade** também é essencial para que qualquer um possa pegar um trecho de código e corrigi-lo ou utilizá-lo.

- ***Feedback*** para que os programadores saibam o mais rápido possível se alguma coisa não está como deveria ser.

- **Coragem** para enfrentar os problemas e, se necessário, jogar fora o trabalho realizado para fazer algo melhor.

- **Respeito** para que não haja brigas internas ou desavenças que possam atrapalhar qualquer tipo de comunicação.

Alguns princípios enunciados pelo autor de XP ajudam a manter estes valores na equipe:

- Exigir *feedback* rápido perguntando sempre se as coisas estão indo como deveriam e pedindo para ser avisado assim que algo diverge do desejado.

- Assumir simplicidade em todo o projeto sem ir procurar a fundo se é mesmo simples ou não.

- Efetuar mudanças incrementais, isto é, pequenas e constantes de maneira a facilitar a identificação de introdução de erros.

- Abraçar mudanças no projeto, considerá-las boas já que estão redirecionando a equipe para o bom caminho.

- Trabalho de qualidade por parte de toda a equipe para que todos mantenham o respeito entre si.

O time de desenvolvimento procurou adotar todas as práticas primárias citadas por Kent Beck [3], criador da metodologia. As práticas secundárias detalhadas na segunda edição [4] do livro não foram incorporadas por serem muito específicas a ambientes corporativos. No pré-projeto realizado para descobrir as tecnologias e a metodologia, a equipe focou-se em algumas práticas de desenvolvimento sugeridas. Foram elas:

- ***Sit together* (Sentar-se junto)**: A equipe se reunia aproximadamente a cada duas semanas para discutir o que tinha sido feito no projeto nas últimas semanas, como estava indo o trabalho e o que deveria ter sido melhor e o que foi bom.

- ***Whole team* (Todo o time)**: Toda a equipe estava envolvida no desenvolvimento do projeto como um todo apesar de cada um ter suas responsabilidades individuais em relação a uma ou outra parte do projeto.

- ***Pair programming* (Programação pareada)**: Nenhum trecho de código do software foi gerado por uma única pessoa em frente ao computador. Tudo foi feito por duas pessoas programando em conjunto e revisando constantemente o código.

- ***Continuous integration*** **(Integração contínua)**: Cada sessão de programação tinha duração variada indo de 30 minutos a 2 horas mas, sempre que uma seção terminava, os programadores enviavam o código gerado ao repositório para que todos tivessem acesso a ele. Isto era feito também ao fim de qualquer tarefa definida (mesmo que no meio de uma sessão).

- ***Ten-minute build*** **(Construção de dez minutos)**: O projeto era suficientemente pequeno para que isso não tenha sido um problema mas sempre tomou-se cuidado para que a compilação fosse suficientemente rápida e pudesse ser realizada constantemente.

- ***Test-first programming*** **(Programação com testes a priori)**: A equipe tentou, sempre que possível, criar testes antes de criar a implementação. Com isso a definição do que deveria ser programado ficava mais clara.

- ***Incremental design*** **(Projeto incremental)**: Apenas uma macro arquitetura do sistema foi desenvolvida no início deixando todos os principais detalhes da arquitetura para serem definidos com o tempo.

Com essa experiência adquirida, quando o time começou a trabalhar no projeto Archimedes foi possível manter as práticas primárias já testadas e ainda anexar algumas práticas derivadas mais sutis.

Mesmo seguindo todas as essas práticas e mantendo os valores, a metodologia teve que ser adaptada para a realidade da equipe. Isso foi necessário pois o ambiente de criação e de descrição de XP é totalmente corporativo com diversos enfoques na sua aplicação em empresas. No caso de um trabalho acadêmico com estudantes de graduação, diversos problemas não existiam neste ambiente, outros mantiveram-se e ainda outros apareceram. Segue uma relação das adaptações que foram feitas à metodologia para conseguir utilizá-la com sucesso.

Problemas que não existiram no projeto por ter sido desenvolvido num ambiente acadêmico:

- **Hierarquia na equipe**: Num ambiente corporativo, as equipes em geral têm hierarquias bem definidas nas quais cada "nível" tem suas funções ou especialidades, que devem ser adaptadas para que a equipe funcione como um todo, sobrepassando essas diferenças. Por tratar-se de estudantes, não existiu nenhum problema do tipo.

- **Troca de linguagem**: XP funciona muito bem com linguagens orientadas a objeto que facilitam o trabalho de desenvolvimento incremental e reutilização de código. Com isso, algumas empresas precisam ou optam por trocar de linguagem de desenvolvimento, o que insere um custo de aprendizagem considerável ao projeto. Esse problema não existiu já que o time todo estava familiarizado com Java.

- **Resistência de programadores experientes**: Programadores experientes em geral possuem alguns vícios de programação ou de trabalho e podem resistir às idéias da metodologia, enfraquecendo a equipe ou desrespeitando as práticas primárias da mesma. Como todos os programadores ainda são estudantes da graduação, nenhum deles enfrentou esse problema.

- **Exigência dos clientes**: Em geral, uma das partes mais difíceis de um projeto é negociar com o cliente o que será feito e quanto custa. É nesta fase que ocorrem as desavenças: o cliente não entende por que os programadores fazem tão pouco e os programadores não entendem o que o cliente pensa que está pedindo. Como o projeto não foi financiado por ninguém, não houve problemas negociações deste tipo. Isso porque os "clientes" do projeto não pagavam nada, e portanto não se viam em posição de exigir nenhuma velocidade.

Problemas que se mantiveram no desenvolvimento do projeto apesar do ambiente ser outro:

- **Continuidade da equipe**: Assim como num ambiente corporativo, o projeto sofreu com perda de membros da equipe. Com o tempo tornando-se escasso por conta do fim da faculdade, o grupo acabou perdendo alguns membros ao longo do tempo e enfrentou dificuldades para integrar novos membros. Assim, a continuidade da equipe é um problema comum aos dois ambientes. No entanto, assim como em empresas, quando novos membros foram encontrados, graças à metodologia seu tempo de aprendizado até se tornarem produtivos na equipe foi bem pequeno.

- **Organização e planejamento**: Lidar com tempo para realizar o projeto, número de pessoas na equipe e investimento (pessoal e financeiro) foi um problema para o desenvolvimento do projeto assim como qualquer outro projeto. Estimar o tempo para implementar as funcionalidades, arranjar pessoas disponíveis para cada horário e conseguir levantar fundos para os pequenos gastos do projeto são dificuldades que podem ser encontradas independente do ambiente no qual está inserido.

- **Ciclos curtos**: A necessidade de trabalhar com ciclos curtos de lançamentos ficou clara para o time assim como é para uma empresa. Essa base para o *feedback* foi ainda mais importante já que era uma das poucas oportunidades em que diversos clientes observavam o trabalho realizado.

- **Funcionalidades em histórias**: A definição do trabalho a ser realizado ficou muito mais clara e fácil sob a forma de histórias como sugerido para os ambientes corporativos. Essa quebra nos padrões de burocracia, comuns nas empresas, facilita a expressão do cliente e o entendimento dos programadores. Essa facilidade também foi observada neste projeto.

- **Ambiente de trabalho informativo**: O ambiente de trabalho com diversos cartazes informativos feitos pelo *tracker*[3] ajudar a manter a equipe atualizada a respeito do andamento do trabalho e das histórias. Essa prática, assim como em projetos corporativos, é muito importante para orientar a equipe se uma direção escolhida está de fato sendo seguida.

- **Propriedade coletiva de código**: Assim como numa empresa, foi absolutamente necessário excluir qualquer tipo de propriedade de código. Isto é, como a equipe não se encontrava por inteira sempre, era impossível ter algum código bloqueado por alguém e, por isso, a propriedade coletiva era altamente necessária. Qualquer código do projeto tinha que ser compartilhado em sua totalidade entre os programadores.

- **Alta qualidade**: Finalmente, trechos de código de má qualidade ou difíceis de entender foram um problema considerável no projeto. Por conta deles, alguma parte do projeto às vezes ficava travada uma semana até que o responsável pudesse sentar com alguém e limpar aquilo tudo. Assim, manter um código de alta qualidade foi essencial para o projeto e uma exigência de todos os membros do time.

Problemas encontrados no ambiente de desenvolvimento acadêmico que não se aplicam a ambientes corporativos:

- **Equipe voluntária**: Por ser um ambiente estudantil, o trabalho no projeto é não remunerado. Com isso, os membros são voluntários. Isso faz com que não exista sentimento de culpa ou possibilidade de cobrança para eventuais faltas ou desistências. Ao contrário do ambiente corporativo, onde os membros de uma equipe recebem salário e são, portanto, cobrados para cumprirem sua função, os membros do Archimedes só têm auto-cobrança.

- **Projetos em tempo livre**: Num ambiente corporativo, os horários de cada funcionário são determinados e sabe-se quando algum membro estará presente ou não. Já no ambiente acadêmico, cada membro tem seus horários de aulas, reuniões ou obrigações pessoais. Assim reunir a equipe é um trabalho árduo e algumas vezes impossível. Isso traz problemas no quesito de tempo de trabalho possível; em geral reduzindo o número de horas semanais de programação para um nível bem aquém de qualquer empresa.

- **Cliente voluntário**: Apesar do fato de não existir envolvimento financeiro por parte dos clientes eliminar o problema da cobrança por parte deles, ele também traz a falta de compromisso por parte destes. Como não sai nada do bolso deles, os clientes não sentem que devem estar presentes para que o trabalho seja bem feito. Tudo para eles é lucro e, portanto, não requer nenhum esforço, caindo novamente no problema do voluntariado.

4 Descrição técnica

O projeto Archimedes utiliza diversos conceitos de programação orientada a objetos para construir um base sólida de arquitetura e implementação. Para isso, um dos pilares utilizados são padrões reconhecidos pela comunidade de orientação a objetos.

Padrões são soluções conhecidas e utilizadas para problemas comuns e recorrentes em uma determinada área. Essa definição é bem genérica e não está de maneira alguma restrita ao mundo da computação. Inclusive, a idéia de padrões foi inicialmente lançada pelo arquiteto Christopher Alexander, que definiu padrões para construção de cidades e prédios [2]. A idéia é definir uma solução em um nível suficientemente alto para que seja possível adaptá-la para os diferentes problemas que surgem naquela área.

Ao longo de todo o desenvolvimento, a equipe se preocupou em tentar identificar os problemas no sistema e procurar padrões para resolvê-los conforme estes se evidenciavam. No início, poucos padrões foram utilizados já que poucos problemas existiam.

A seguir serão descritas diversas etapas da evolução do software. Para descrever a arquitetura do mesmo, será usada uma notação muito conhecida, chamada *Unified Modeling Language* (UML), foi extensamente explicada por Martin Fowler em um de seus livros [6].

4.1 A primeira modelagem

A primeira modelagem feita considerava que seriam necessários alguns grupos de classes bem definidos:

- Interface gráfica (*GUI*)
- Modelo de dados (*model*)
- Interpretador de comandos (*interpreter*)
- Controlador (*controller*)

A idéia dessa separação era seguir o padrão *Model-View-Controller* (MVC) [12], que é um padrão consolidado para arquitetura de programas com interfaces gráficas.

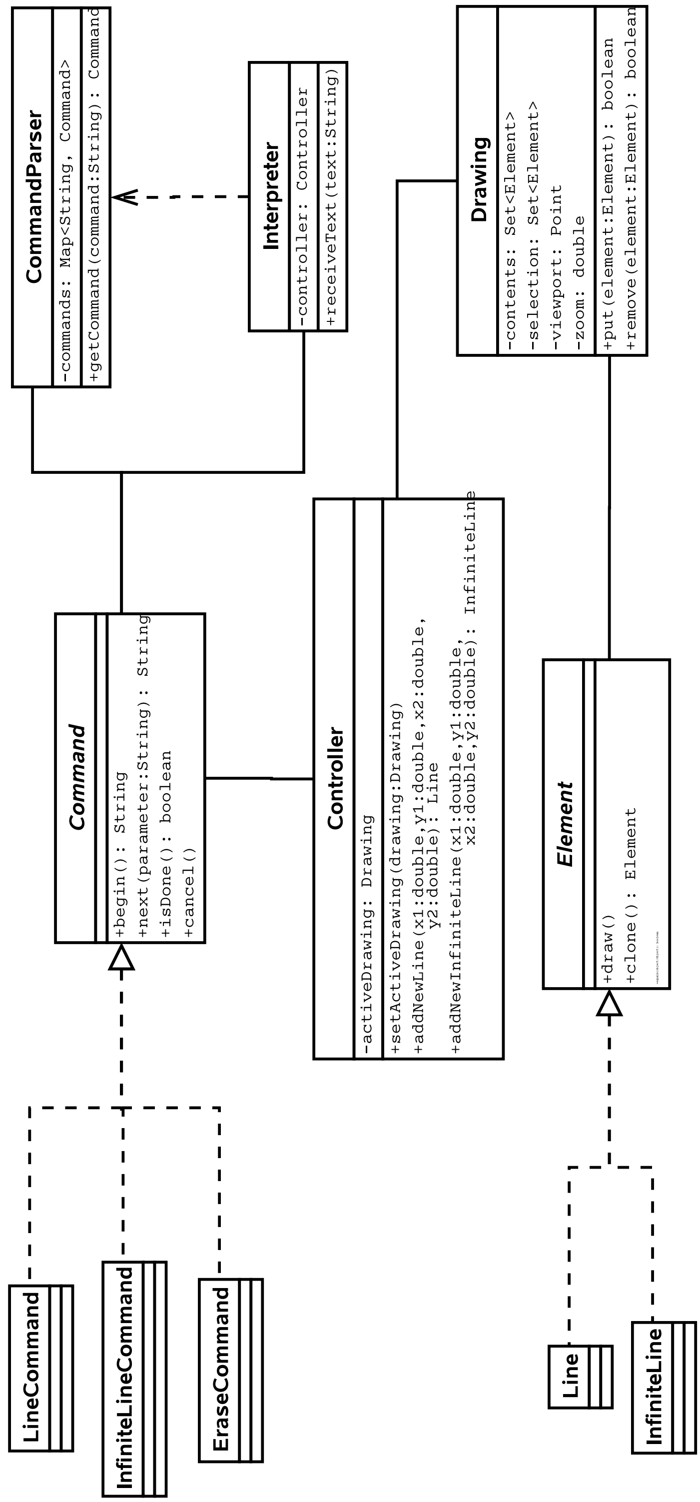

Figura 2: Primeira versão da arquitetura

O problema que motiva o padrão MVC é o de possuir um modelo mas ter que permitir diversas visualizações diferentes. Ele também é influenciado pelo forte acoplamento que muitas vezes ocorre entre o modelo e a visualização.

A solução sugerida pelo padrão é criar três camadas que permitam tornar o modelo independente da visualização. Com isso, a apresentação pode mudar constantemente ou ter diversas versões sem precisar se preocupar com o modelo. Outra vantagem é que o modelo pode mudar internamente sem alterar a visualização desde que este mantenha-se consistente para a visualização. Quando ocorre alguma mudança no modelo vinda da visualização (que é a interface com o usuário em geral), todas as mudanças devem ser realizadas através do controlador, que serve como canal e filtro de comunicação com o modelo, desacoplando o modelo da visualização.

No caso do Archimedes, existem dois modelos:

1. O desenho: O modelo essencial do software onde é feito o trabalho real. Este modelo é o que interessa o usuário pois é o que ele vai produzir.

2. O ambiente de trabalho: Este modelo é que mantém as informações a respeito do ambiente de trabalho do usuário. É neste modelo que são guardadas as informações a respeito dos desenhos que estão abertos e das preferências que estão ativadas ou não, entre outras coisas.

O controlador é único para cada execução do software pois diversas coisas não podem acontecer ao mesmo tempo nos desenhos para evitar confusão para os usuários. Finalmente a visualização é a interface gráfica incluindo as janelas de edição das propriedades e a janela principal.

Para garantir a unicidade do controlador em cada execução, optou-se por utilizar mais um padrão bem simples e conhecido: o *Singleton* [8]. O problema que ele propõe resolver é exatamente o que foi citado: garantir a unicidade de um objeto ao longo de uma execução. Para isso, a proposta é incluir no objeto uma variável privada estática da classe que guarda uma instância dessa mesma classe. Além dessa variável é necessário um método para acessá-la que vai inicializá-la assim que for necessário. Finalmente deve-se impedir que uma instância seja criada por qualquer outro método que este último.

Além dos componentes do MVC, um quarto componente foi incluído na arquitetura inicial: o interpretador. Como o AutoCAD funciona com uma interface de comandos, julgou-se necessário interpretar o que o usuário pudesse vir a digitar para transformá-lo em objetos do modelo. O interpretador consistia de uma class `Interpreter` responsável por receber, na forma de texto, os parâmetros digitados pelo usuário. Ele usava o `CommandParser` para obter comandos identificados por nomes. Os comandos, implementando a interface `Command`, recebiam os parâmetros do interpretador e realizavam as ações quando os parâmetros tivessem terminado. Por exemplo, o `LineCommand` recebia como parâmetros dois pontos (em um formato texto) e criava uma linha.

A maior parte das classes e componentes descritos nessa seção pode ser vista no diagrama da Figura 2, página 12. As classes `Drawing` e `Element` (com suas subclasses) representam o modelo do desenho. A modelagem da parte da interface gráfica e do modelo da área de trabalho foram omitidas por questões de simplicidade, e sempre serão omitidos daqui em diante.

4.2 Desfazer Comandos

A arquitetura inicial (Figura 2, página 12) não sofreu grandes alterações no começo. Novos comandos eram adicionados à interface dos elementos ou à do Controlador. No terceiro lançamento, no entanto, uma das funcionalidades pedidas levou a uma mudança na arquitetura: o pedido para desfazer e refazer comandos executados anteriormente, sem limite na quantidade de comandos que poderia ser desfeita. Para tanto, era necessário manter um histórico dos comandos que foram completados com sucesso. Adaptou-se então a arquitetura para que os comandos gerassem um objeto contendo todas as informações necessárias para se desfazer e se refazer. A lógica, no entanto, permaneceu no comando já que cada comando tinha seu próprio jeito de se fazer. Surgiu então a seguinte arquitetura descrita na Figura 3 (página 15) implementada por cima da primeira (Figura 2 na página 12).

Como os comandos é que causavam as mudanças no desenho e eles podiam ser cancelados ou finalizados em números de iterações diferentes, optou-se por utilizar o padrão *Observer* [8]. Graças a ele, o controlador era notificado sempre que um comando terminava e guardava então o objeto com as informações necessárias para desfazê-lo. Este jeito simples permitiu adicionar a funcionalidade de fazer ou desfazer sem alterar muito o resto do programa bastando que o comando criasse seus objetos `Executed` e que o controlador os registrasse junto com o desenho.

4.3 Repetição de Código dos Comandos

Até então só existiam os elementos linha e linha infinita e havia poucas maneiras para o usuário criá-las. No quarto lançamento, foram pedidos novos elementos e novas maneiras de criá-los. Por exemplo, uma linha poderia ser criada por dois pontos ou por um ponto, um tamanho e um ângulo. Além disso, essas novas maneiras deveriam ser usadas para mais de um comando. Por exemplo, o comando mover usava uma seleção de elementos e dois pontos, referência e destino, para ser completado. Assim, a referência e destino também deveriam poder ser especificadas como um ponto, uma distância e um ângulo.

Nessa época, os comandos recebiam como parâmetro do método `next` uma `String`

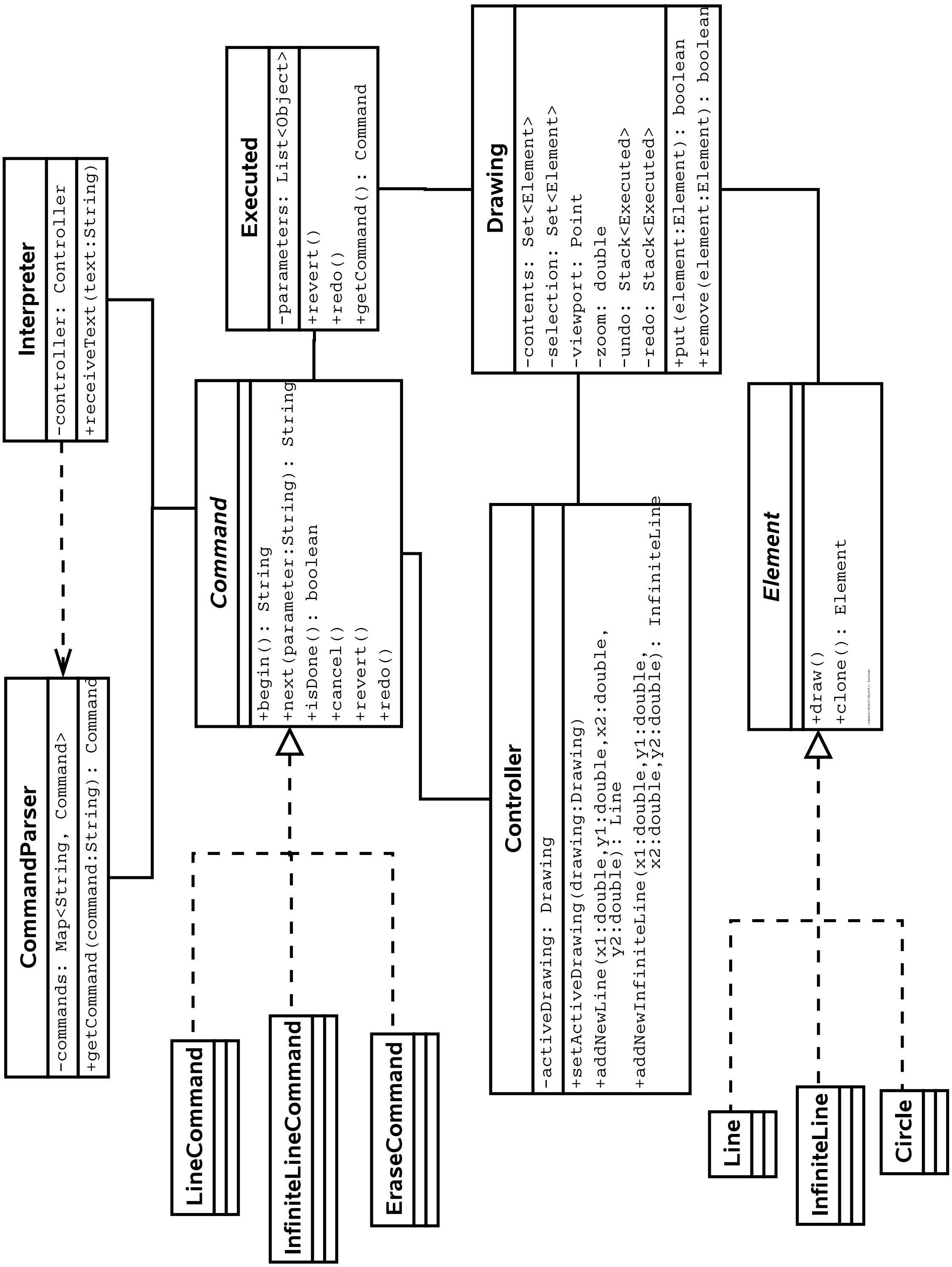

Figura 3: Arquitetura com *undo* e *redo*

digitada pelo usuário. Assim, cada comando, primeiro, lidava com uma seqüência desses parâmetros e depois era finalizado, passando os argumentos obtidos para o controlador. Isto é, para cada comando, havia aproximadamente um método correspondente no controlador. "Aproximadamente" pois alguns comandos na verdade burlavam o controlador, comunicando-se direto com o modelo.

Além disso ser indesejado, a modelagem em si apresentava diversos pontos de repetição de código. Primeiro, na maneira de lidar com os parâmetros, que como foi dito em muitos comandos era similar. Além disso, havia repetição no momento de adicionar um comando para criar um novo tipo de elemento. Apesar dos parâmetros recebidos pelo comando poderem ser diferentes, o código para completar o comando em si era muito parecido. Era necessário criar um método no controlador para adicionar o elemento no desenho e usá-lo no comando.

4.3.1 Os *Parsers*

Para sanar esse problemas de duplicação de código, que sempre é altamente indesejável, foi idealizada uma refatoração da arquitetura que promovesse maior reaproveitamento do código. Por ser muito grande, essa refatoração foi implementada em duas fases. A primeira (Figura 4 na página 17) trouxe a introdução da interface `Parser` para aproveitar o código que lidava com os parâmetros do usuário.

O `Parser` recebe os parâmetros digitados pelo usuário na forma de texto, gerando um parâmetro de mais alto nível para o comando. Assim, no exemplo dos comandos que recebem dois pontos, primeiro é usado um `PointParser` que produz um ponto simples e depois é usado um `VectorParser`, que aceita diversas maneiras de definição do segundo ponto e produz um vetor indicando o tamanho e direção definidos pelo usuário. Com isso, os comandos recebiam objetos já no nível que pudessem manipulá-los e, assim, simplificaram-se consideravelmente.

Ainda assim, havia o problema da replicação de código no controlador e da quebra na regra do MVC.

4.3.2 Usando a interface *Factory*

Assim, no sexto lançamento, veio a segunda fase da refatoração. Decidiu-se extrair a funcionalidade de recuperação de parâmetros da funcionalidade de execução de função. Com isso, foram criadas duas interfaces: `CommandFactory` e `Command` onde `CommandFactory` iria substituir os atuais comandos e `Command` iria representar um comando a ser realizado já com seus parâmetros. Podemos pensar no `Command` como um bloco que encapsula tanto o código que deve executar como os argumentos para executar este bloco.

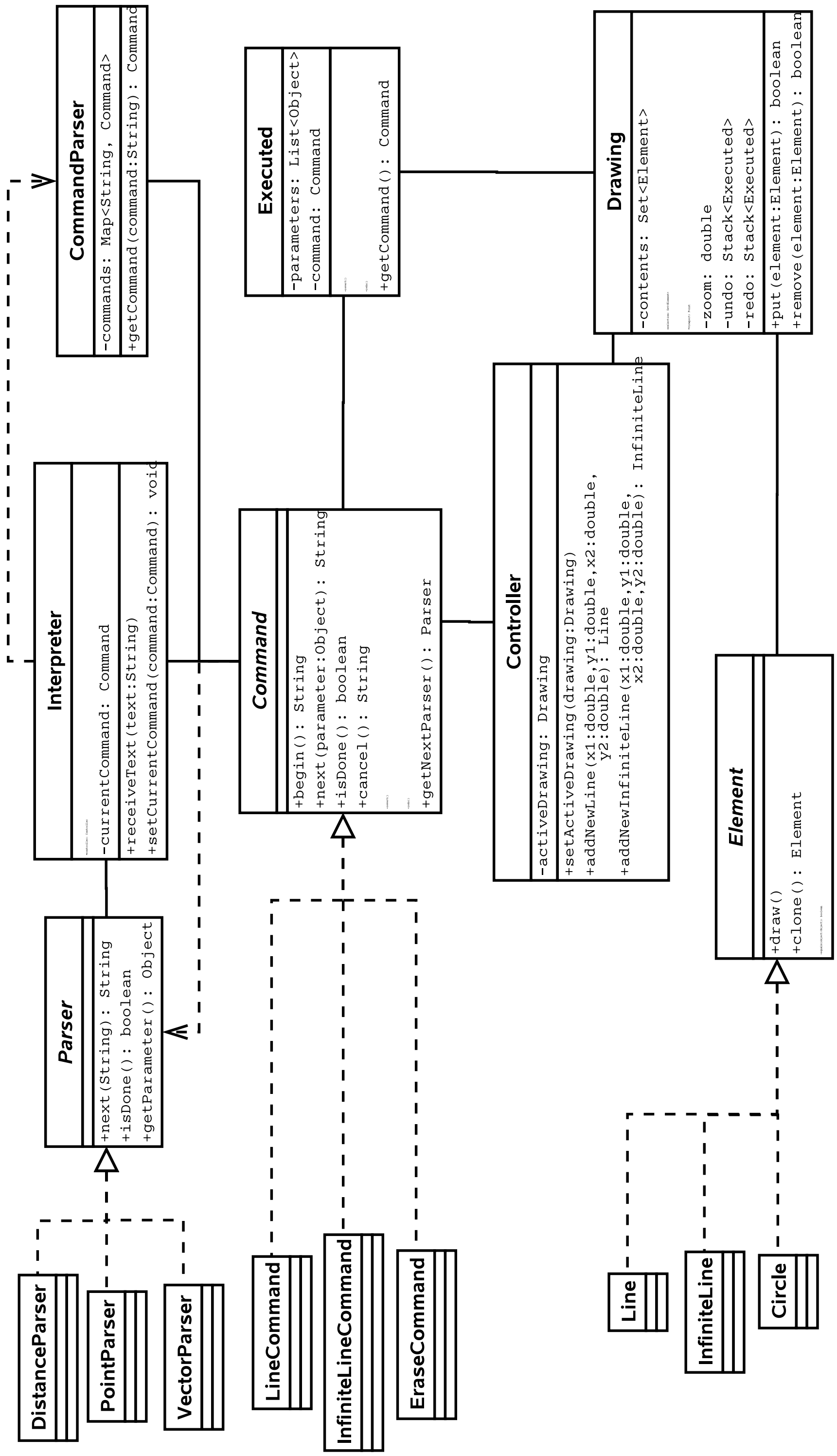

Figura 4: Arquitetura com `Parser`

Além de separar as funcionalidades, essa mudança traria a eliminação dos `Executed` para manter apenas pilhas de `Command` e simplificaria o `Controller` que apenas precisaria saber lidar com `Command`. Com isso a arquitetura ficou como na Figura 5 na página 19.

Essa arquitetura apresenta uma implementação do padrão *Command* [8], exatamente com o novo `Command`. O problema que motiva este padrão é justamente o de executar diversas operações sem sobrecarregar os objetos nos quais elas devem ser efetuadas. Para isso, utiliza-se um objeto que encapsula uma ação e recebe como parâmetro o objeto no qual ele deve ser aplicado. Exatamente o que acontece já que o `Command` recebe um `Drawing` no qual ele deve ser executado. Com isso, basta que o `Drawing` saiba receber um `Command` e executá-lo para que seja possível realizar muitas ações diferentes nele sem ter que modificá-lo. Com isso também ficou mais simples para o desenho desfazer e refazer comandos já que estes já guardam as informações necessárias para ambas operações.

Além disso, criou-se a interface `CommandFactory` que permitiu extrair a recepção de parâmetros dos comandos. Com isso, juntou-se todo o código que permitia receber dois pontos em uma classe abstrata e implementou apenas alguns métodos para criar os elementos necessários como linha, linha infinita, retângulo e outros. Essa solução é uma instância do padrão *Template* [8]. O problema que motiva este padrão é o mesmo que motivou a equipe: realizar praticamente a mesma coisa para diversos objetos sem, no entanto, fazer exatamente a mesma coisa em uma ou outra parte.

Outro padrão que se evidencia com a `CommandFactory` é uma adaptação do padrão *Factory* [8], que permite criar diferentes objetos de acordo com os argumentos passados. Com isso, uma mesma *Factory* pode dar origem a diversos tipos de objetos desde que estes tenham uma interface comum. No nosso caso, a interface comum é `Command` já que cada `CommandFactory` dá origem a uma ou mais instâncias de `Command`.

4.4 Os Escritores

Ainda no sexto lançamento, o Archimedes adquiriu as funcionalidades de impressão e exportação para alguns formatos de imagem. A implementação, nessa versão, trazia os métodos para desenhar em cada tipo de mídia nos próprios elementos, o que prejudicava bastante a coesão dos mesmos. Assim, na versão seguinte, foi criada uma interface `Writer` que define métodos para escrita do desenho e dos diferentes tipos de elemento. Para cada tipo de mídia é criado um `Writer` que encapsula a lógica de escrita nessa mídia. Esse modelo foi usado também para desenhar no próprio `Canvas` da janela e para salvar os arquivos XML[6] [14] do Archimedes.

[6]XML: eXtensible Markup Language

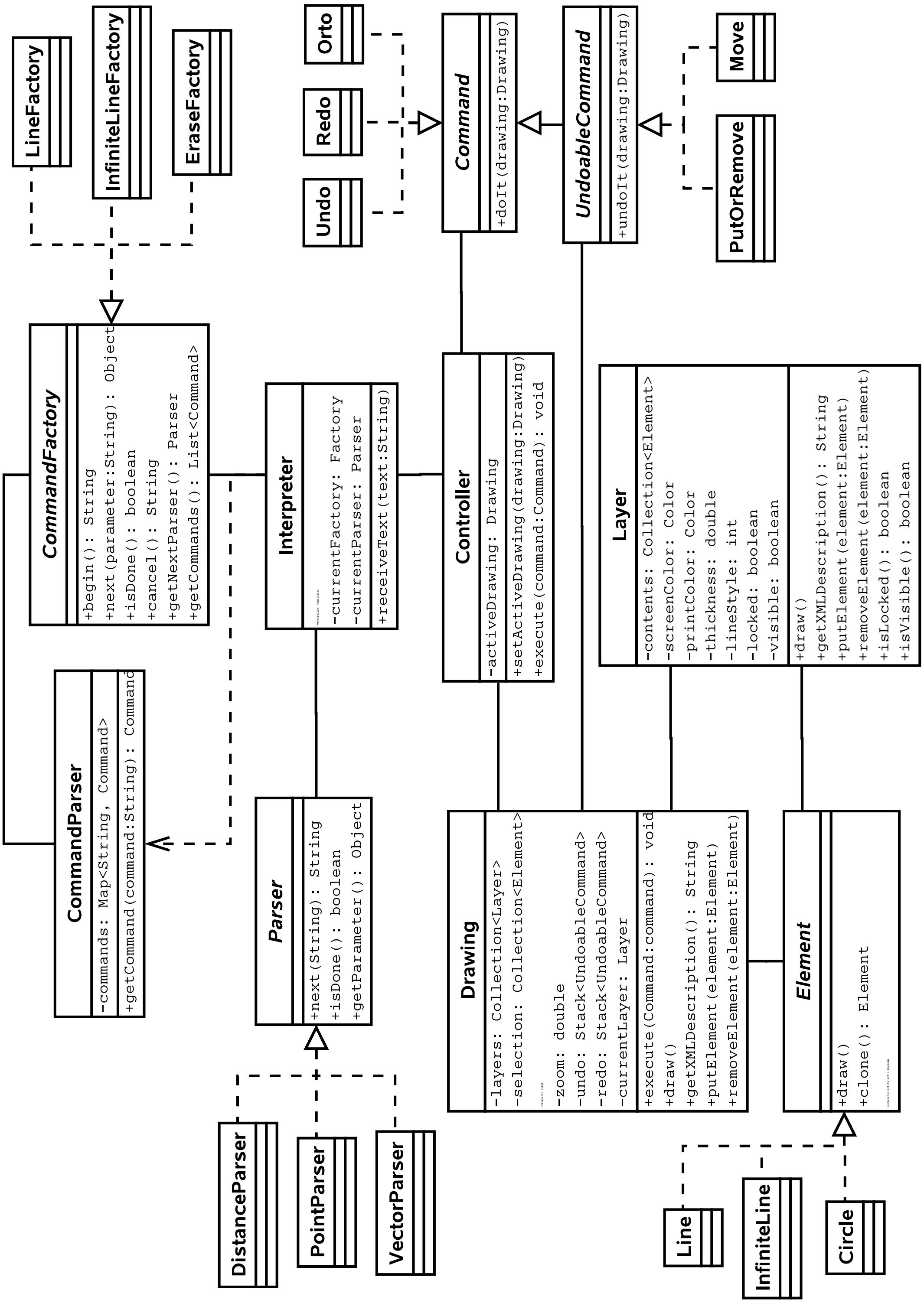

Figura 5: Arquitetura com o uso de `Factory`

Para lidar com os diferentes tipos de elementos, cada `Writer` usa uma técnica conhecida como *double dispatch*. Com isso, um elemento recebe o `Writer` no qual deve ser escrito e a seguir pede a ele que o escreva, permitindo identificar seu tipo. Por sobrecarga de método, o código certo do `Writer` será chamado.

4.5 `Interpreter` tornando-se `InputController` com estados

Com o tempo, a parte da visualização tornou-se muito complexa por causa da falta de experiência na modelagem de um sistema de janelas com SWT da equipe. Com o pedido de criação de uma quantidade enorme de ícones e menus semelhantes, percebeu-se que a situação da parte gráfica estava se tornando inviável. Seria então preciso uma grande refatoração que permitisse facilitar a inserção de componentes sem aumentar o código continuamente.

Em primeiro lugar, o estado de todos esses atalhos deveria ser atualizado conforme o estado do programa, isto é, o estado do interpretador. Para facilitar esse trabalho, resolveu-se evidenciar os diferentes estados possíveis do interpretador. Com essa evidenciação, percebeu-se que o `Interperter` não interpretava nada e simplesmente controlava o fluxo de entradas. Por isso, ele foi renomeado para `InputController` e ganhou uma implementação do padrão *State* [8] (Figura 7 na página 22).

Esse padrão se propõe a resolver os problemas decorrentes de uma estrutura com diversos estados. Para evitar os vários trechos de código com *if*'s e *else*'s, existe um controlador que mantém apenas o estado vigente definido por uma interface, e cada estado é responsável por implementar comportamento especifico a ele.

4.6 Arquitetura da interface ao usar a *Rich Client Platform* (RCP)

A plataforma RCP é um *framework* para criar aplicações gráficas com suporte a *plug-ins*. Ela foi desenvolvida junto com o aplicativo Eclipse[7] e provê uma biblioteca gráfica de alto nível com suporte a mecanismos de adição de módulos tanto gráficos quanto de modelo.

A necessidade de flexibilizar o software para permitir que outras pessoas desenvolvam módulos específicos fez com que a equipe procurasse um jeito fácil de suprir essa necessidade. O RCP foi escolhido pois é muito robusto e extremamente flexível. Outro ponto importante é o fato do RCP ser baseado no SWT que é e a biblioteca gráfica que era usada pelo projeto desde o início. Logo, optou-se por adotar essa solução e iniciou-se a transição (que ainda não acabou).

[7] `http://www.eclipse.org`

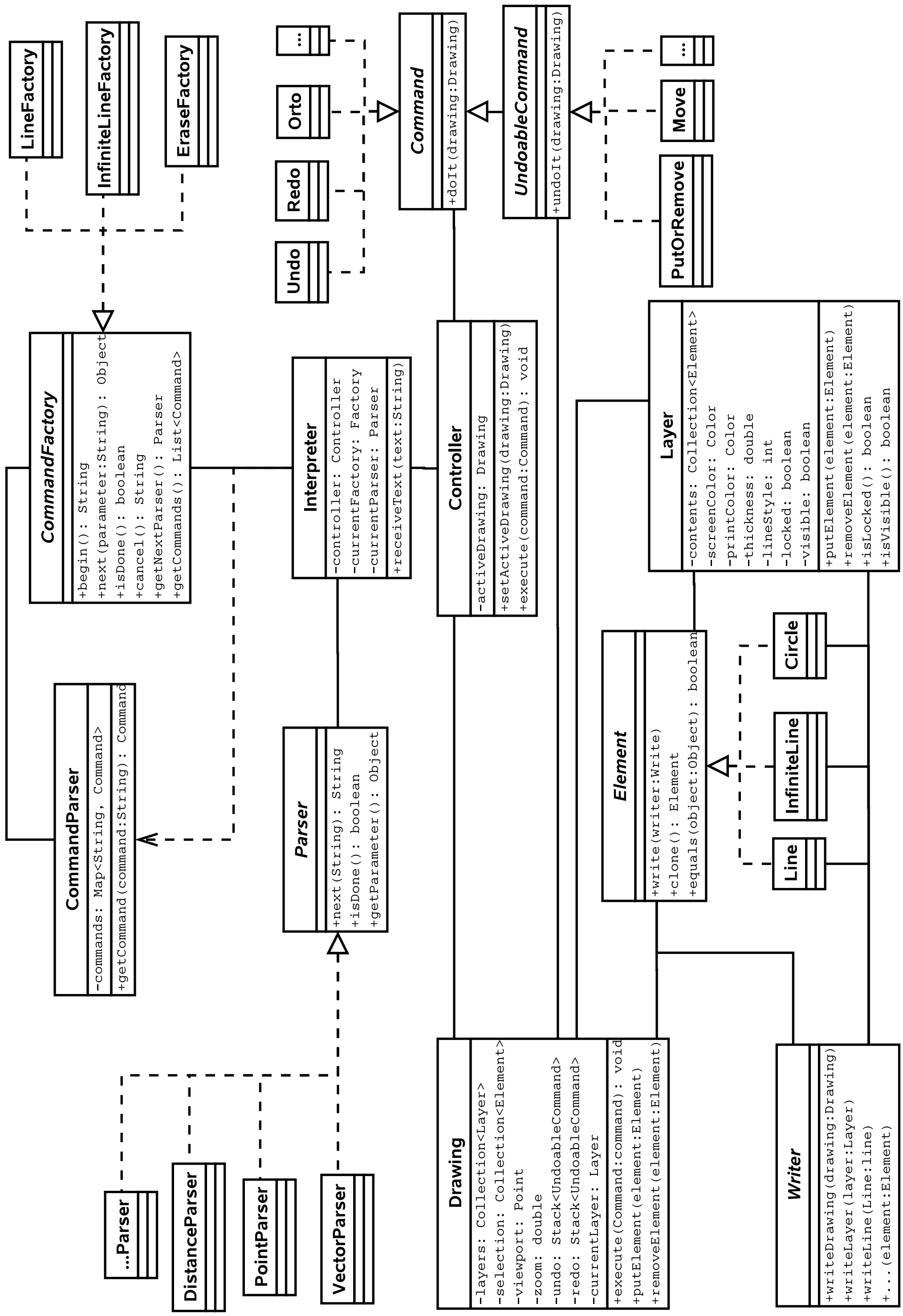

Figura 6: Arquitetura com o uso de `Writer` para a saída

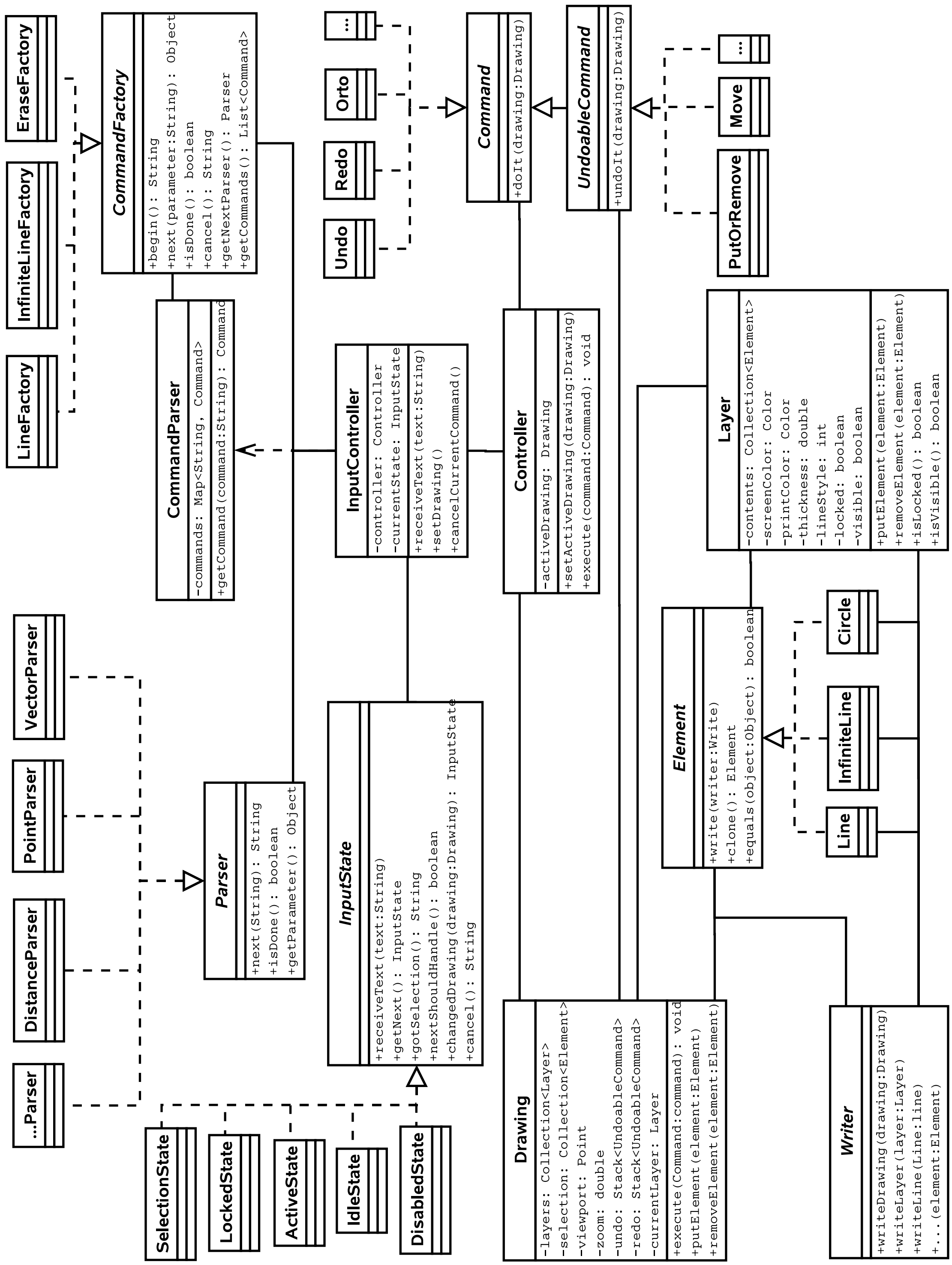

Figura 7: Utilização do padrão *State* no `InputController` (ex-`Interpreter`)

Essa mudança afetou principalmente a arquitetura da interface gráfica que não tinha sido apresentada até agora por não ter muito interesse. Pelo uso do *framework*, a arquitetura foi muito direcionada pela estrutura proposta.

Não faz muito sentido apresentar um diagrama com as classes envolvidas pois a maioria das ligações entre classes são feitas pela paltaforma e tornam-se transparentes para os desenvolvedores. Assim, as classes mais importantes da interface serão descritas de acordo com suas funcionalidades.

Inicialmente, a plataforma exige cinco classes básicas que definem uma janela de aplicação. A primeira classe, chamada de `Activator`, é apenas um ativador chamado pela plataforma para manter uma referência às outras classes. Em seguida, o *framework* começa a trabalhar com o `Workbench`, que representa o conceito da mesa de trabalho, e para isso instancia a classe `ApplicationWorkbenchAdvisor`.

Essa classe é responsável por determinar qual perspectiva deverá ser usada nesse `Workbench`, além de instanciar a classe `ApplicationWorkbenchWindowAdvisor`. Esta, por sua vez, organiza a janela da mesa de trabalho, criando uma `ApplicationActionBarAdvisor`. Esta, finalmente, cria as ações que serão usadas no menu ou na barra de ferramenta e as ordena nesses elementos.

Além dessas, existe a classe `Perspective`, que determina a perspectiva que foi citada anteriormente. Ela é responsável por decidir o que será mostrado na janela e em que posições.

Com esse quinteto de base, é possível começar a trabalhar com a plataforma, seja por *plug-ins*, seja implementando o software normalmente. No caso do Archimedes, foi necessário criar um editor capaz de exibir os desenhos abertos e manter o modelo associado. Por isso foram criadas, respectivamente, as classes `DrawingEditor` e `DrawingInput` que são instanciadas pela plataforma conforme se abre um novo editor de desenho.

Fora essas mudanças, o máximo que foi necessário fazer foi criar os arquivos XML que configuram a plataforma e criar algumas classes de ações que eram específicas do Archimedes. Apesar de implementada, essa funcionalidade ainda não foi disponibilizada para os usuários por estar ainda em testes já que é necessário criar alguns pontos de extensão que permitam a funcionalidade real dos *plug-ins*.

Após esse visão geral do sistema, é interessante observar os algoritmos desenvolvidos para resolver um ou outro problema.

5 Algoritmos específicos

Algumas partes do software possuem algoritmos não tão triviais. Principalmente quando o objetivo é uma operação geométrica genérica não tão básica. Um algoritmo clássico de computação gráfica é o algoritmo de recorte [5], que permite determinar que elementos (ou partes deles) aparecem na tela para poder mostrá-los corretamente. Como a parte gráfica ficou a cargo do OpenGL, uma grande parte dos recortes era feito pela biblioteca. Porém, no caso de linha infinita, o trabalho tem que ser feito pelo programa. Pelo modelo de dados que tínhamos adotado, uma linha infinita era definida por uma linha. A linha define uma posição e um ângulo, e o algoritmo deve determinar que segmento da linha infinita com esse ângulo passando pela posição especificada está dentro da área mostrada a cada momento (Figura 8 na página 24).

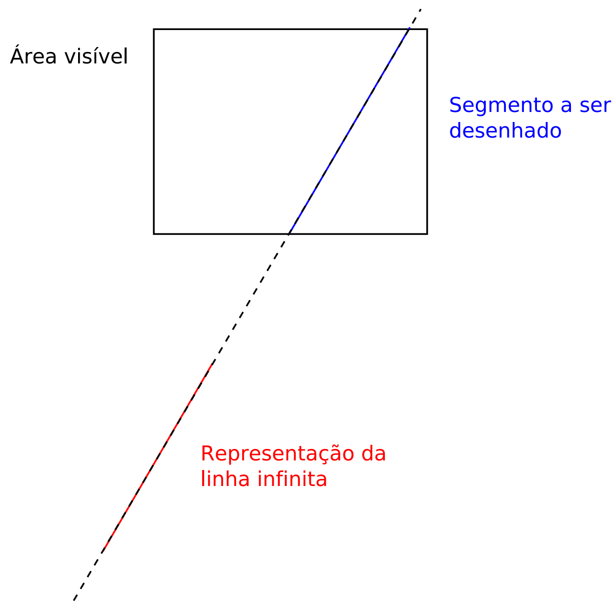

Figura 8: Recortando uma linha infinita

Para implementar esse algoritmo, foram utilizados os algoritmos de intersecção entre linhas. A idéia do algoritmo é, dado um retângulo que determina a área visível, procurar as intersecções com os lados desse retângulo e então formar uma linha de uma intersecção à outra. Nos casos extremos em que não há ponto de intersecção, só há um ponto de intersecção ou há infinitos pontos de intersecção, o resultado do algoritmo é não mostrar nada já que significam que a linha infinita está nos limites da visualização ou fora dela (Figura 9 na página 25).

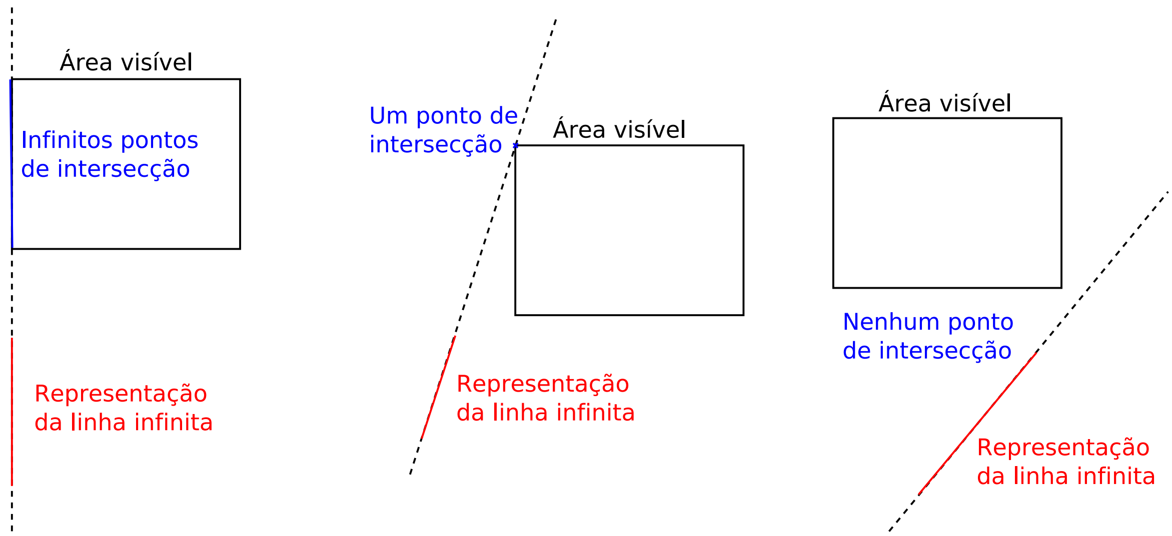

Figura 9: Casos extremos de recorte

Uma outra necessidade do programa é determinar a intersecção entre quaisquer dois elementos existentes. Isso permite pontos de grude, e é usado nas operações de corte e extensão de elementos. A solução que se encontrou usa, em primeiro lugar, o *double dispatch* para que seja possível descobrir os tipos específicos dos elementos envolvidos. Como a operação de intersecção é comutativa, cortam-se alguns casos. Assim, cada elemento sabe calcular sua intersecção com todos os tipos de elementos. Com isso, o algoritmo que calcula a intersecção pode pedir informações específicas de cada tipo de elementos para determinar a melhor forma de descobrir suas intersecções.

Outro problema complicado é o de criar elementos "deslocados" (*offseted*). A idéia é que é possível criar uma cópia deslocada de qualquer elemento como na Figura 10 página 26. No caso de linhas é bem simples, é uma paralela. A coisa complica conforme a forma do elemento é mais livre, por exemplo, fazer uma cópia deslocada de um círculo é mudar seu raio mas, então, existe um limite, pois o raio não pode ser nulo ou negativo. O problema fica complexo quando começamos a lidar com polígonos irregulares. Nesse caso, determinar se alguma coisa passou do limite não é trivial. Ainda existem outros problemas, como por exemplo o desaparecimento de alguma face como na Figura 11 (página 27).

O algoritmo que resolve esse problema utiliza diversos conceitos de álgebra linear. O que se faz é, primeiro, determinar se o deslocamento é pra um lado ou para o outro, isto é, determina-se a orientação entre os pontos que definem o elemento e o lado clicado. Feito isso, como a orientação dos pontos que definem um elemento é sempre

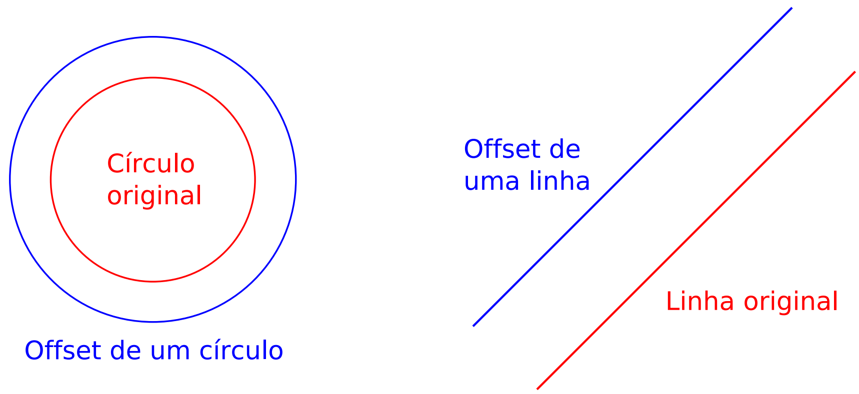

Figura 10: Deslocamento de uma linha e de um círculo

relativamente a mesma, realiza-se o deslocamento para o lado escolhido de todos os trechos do elemento. Pega-se então a intersecção desses itens deslocados com as bissetrizes que os originais formam uns com os outros (como mostrado na Figura 11, página 27).

Este algoritmo, como muitos em computação gráfica (e geometria computacional), mostra que o que é muito fácil para um ser humano (saber o lado do elemento para o qual ele deve ser deslocado) nem sempre é simples geometricamente. Apesar da idéia não ser tão complexa, a solução requer uma organização da estrutura de dados e aplicações de algebra linear bem mais complexas do que pode parecer.

Outro problema que parece trivial para um ser humano é traçar um círculo ou um arco. Por mais que traçar um círculo perfeito seja difícil, arcos e "círculos" são facilmente feitos desenhando manualmente. Para um computador, no entanto, desenhar um arco ou um círculo não é possível. A matemática envolvida num computador é discreta e, portanto, não é possível desenhar um círculo corretamente.

É necessário transformar esse círculo em muitas retas pequenas para conseguir obter algo que pareça um círculo aos olhos humanos. O Archimedes teve que enfrentar esse problema e, com isso, terá de encarar o problema de performance decorrente dele em breve. Para atingir um nível satisfatório de velocidade, será necessário utilizar um número variável de retas que iriam formar o círculo. Essa variável depende explicitamente do número de *pixels* que o círculo ocupa na tela a cada instante. O cálculo feito leva em consideração o mapeamento do modelo para a tela e então o tamanho do círculo ou arco na tela. O número de retas que formam o elemento é então proporcional ao tamanho que ele irá ocupar na tela.

Outro problema parecido com esse é o de ocultar elementos na tela que são peque-

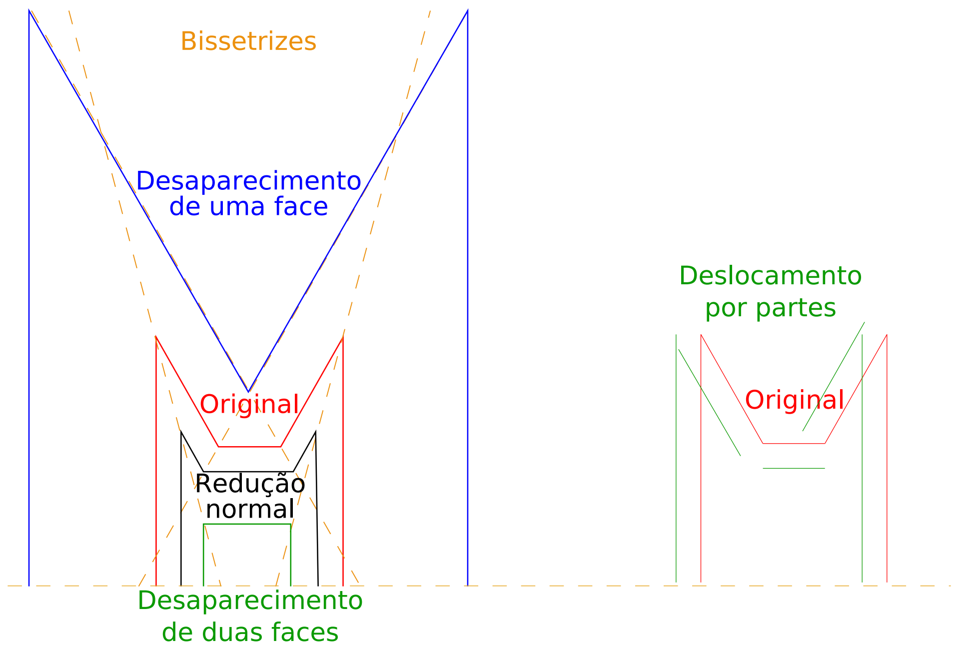

Figura 11: Deslocamento de um polígono qualquer

nos demais. A idéia é a mesma que a citada anteriormente com a diferença de ter um tamanho mínimo a partir do qual o elemento pode ser desprezado. Isso servirá para otimizar a velocidade de renderização da imagem quando muitos elementos se encontram em um único ponto.

Finalmente sobraram dois algoritmos geométricos cuja complexidade é considerável. O primeiro é o algoritmo de corte de elementos dadas diversas referências. Esta operação chamada *trim* permite ao usuário eliminar pedaços de um elemento removendo os trechos dele que interceptam outros elementos considerados de referência. O algoritmo em pseudo-código é algo como na Figura 12 na página 28.

Cuidados devem ser tomados nos casos extremos em que todo um lado de um elemento deve sumir. Também pode ocorrer de não existir intersecções com as referências ou do clique estar bem em uma intersecção com a referência.

Em último, o algoritmo responsável por unir duas linhas por um arco com raio definido também não é trivial. Primeiro, é necessário observar que o centro do arco que une quaisquer duas linhas não colineares, se existir, estará na bissetriz. Com isso, é necessário descobrir em que posição da bissetriz ele estará. Para descobrir isso, a idéia é realizar um deslocamento das duas linhas envolvidas de uma distância igual ao raio

```
foreach(click) {
  element = getElementUnder(click);
  closerOneSide = element.initial;
  closerOtherSide = element.ending;
  foreach(reference) {
    intersection = element.getIntersections(reference);
    if(intersection.calculateDistance(click) > closerOneSide) {
      closerOneSide = intersection;
    }
    else if(intersection.calculateDistance(click) < closerOtherSide) {
      closerOtherSide = intersection;
    }
  }
  delete(element);
  addNewElement(element.initial, closerOneSide);
  addNewElement(closerOtherSide, element.ending);
}
```

Figura 12: Pseudo-código para a operação de corte

do arco. Novamente, o problema aqui é determinar o que é considerado "pra dentro" entre duas linhas arbitrárias. Resolvido este problema, bastar estender as linhas até intersectarem o círculo cujo centro acaba de ser encontrado. O arco é então o que une as duas linhas e tem como centro o que foi calculado.

6 Atividades realizadas

O projeto se encontra agora perto de sua versão 0.18.0 (*Screenshot* da versÃ£o 0.17.1 na Figura 13 (página 29), em um estado quase satisfatório para que os usuários possam usá-la freqüentemente. Além de salvar e abrir arquivos próprios, o Archimedes deverá conseguir abrir arquivos DWG e DXF usando as bibliotecas disponibilizadas pela *OpenDWG community*. Atualmente já é possível imprimir os desenhos do Archimedes nos tipos de impressoras mais comuns (como impressoras a jato de tinta, a laser, matriciais ou plotters). Além disso a manipulação dos desenhos é realizada com um conjunto de ferramentas determinadas por arquitetos usuários de AutoCAD.

Ao longo de seu desenvolvimento, o Archimedes conseguiu chamar a atenção de algumas pessoas da comunidade de software livre, sendo mencionado algumas vezes. Uma delas foi um comentário de César Brod sobre o seu primeiro contato com o Archimedes,

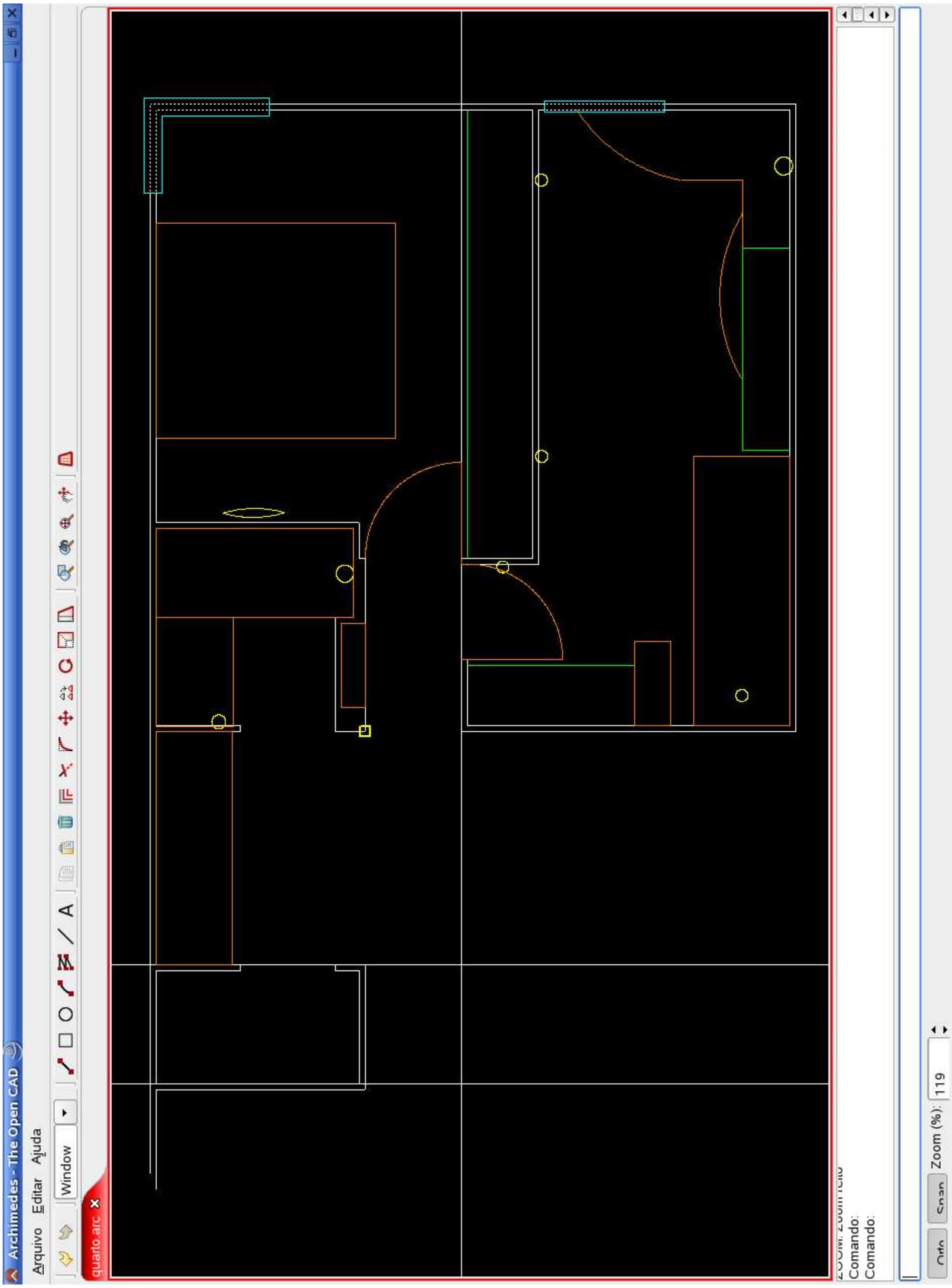

Figura 13: Tela do Archimedes 0.17.1 em uso no Linux

disponível em `http://br-linux.org/linux/maleiros-e-cad`. Mais recentemente, o projeto foi citado numa entrevista com Sérgio Amadeu disponível como *podcast* na IDGNow [8] ou com uma transcrição na BR-Linux [9].

O projeto foi apresentado informalmente na LinuxWorld Brasil 2006, onde os autores foram apresentadores de mesa nas palestras entre 23 e 25 de maio. A participação no CONISLI[10] entre os dias 3 e 5 de novembro de 2006 realizado pelo IDEPES[11] em São Paulo também foi essencial para a divulgação do projeto. Ocorreram também apresentações do projeto no 3^o Workshop da TIDIA[12], no SINAENCO[13] para o grupo de Estudos de Informática Aplicada e para a SERPRO[14].

Além desses, no dia 8 de dezembro, o projeto será apresentado na Universidade Federal de Santa Catarina numa seqüência de palestras sobre software livre. Finalmente, o software também garantiu a oportunidade aos autores desta monografia de apresentar duas palestras sobre este trabalho no Seminário de Desenvolvimento de Software Livre que ocorrerá no início de 2007 em Lajeado - RS.

7 Resultados e produtos obtidos

O projeto foi muito bem recebido pela comunidade de software livre e ganhou ainda mais apoio desde que a Sun Microsystems liberou o código fonte do Java sob a licensa GPL v.2.

Todo o trabalho está disponível em diversos sites do projeto. Os principais são:

- Portal da Incubadora da FAPESP

  `http://www.archimedes.org.br`

  O site do projeto com notícias, ajudas e informações.

- Sourceforge.net

  `http://sourceforge.net/projects/arquimedes`

  O espaço do projeto no sourceforge onde encontram-se todas as versões do software disponíveis para *download.*

[8] `http://idgnow.uol.com.br/idgestaticas/podcasts/sergio_amadeu_281106.mp3`

[9] http://br-linux.org/linux/em-entrevista-sergio-amadeu-comenta-a-pesquisa-da-abes-e-prega-decreto-presidencial-em-favor-da-liberdade

[10] Congresso Internacional de Software Livre

[11] Instituto de Desenvolvimento e Pesquisa de Software

[12] Tecnologia da Informação no Desenvolvimento da Internet Avançada

[13] Sindicato Nacional das Empresas de Arquitetura e Engenharia Consultivas

[14] Serviço de Processamento de Dados do Governo

Em pouco mais de 8 meses de desenvolvimento, o projeto passou a marca de 1200 downloads por versão estável apesar de ainda não estar pronto para uso freqüente. Por essa rápida difusão, o projeto ganhou ajudas voluntárias de tradução por parte de um italiano, Davide Pesenti, de um alemão, Mario Fichtenmayer, e de outro voluntário que está trabalhando na tradução para espanhol.

Além dessas traduções, o projeto também recebeu o apoio de um brasileiro, Alexandre Erwinn, que começou a desenvolver um AutoLISP (script para o AutoCAD) que permite, de dentro do software, exportar o desenho ativo para o formato de arquivos do Archimedes.

Com isso, o projeto está agora disponível em português, inglês, francês, italiano, alemão e futuramente em espanhol. Além disso, o projeto funciona nos principais sistemas operacionais apesar de encontrar problemas no Mac OS X devido a incompatibilidades do sistema com a biblioteca gráfica adotada. O projeto atualmente permite trabalhar com duas dimensões e possui os principais elmentos simples desse nível.

O Archimedes tem como elementos principais:

1. **Segmento**: Chamado "linha". É o principal elemento de desenho.
2. **Semi-linha**: Obtida apenas cortando linhas infinitas.
3. **Linha**: Chamado "linha infinita". Serve para definir referências ou guias para o desenho.
4. **Círculo**: Muito usado para depois ser cortado e tornar-se um arco.
5. **Arco**: O elemento curvo mais usado. Por enquanto só pode ser arco de círculo e não de elipse.
6. **Poli-linha**: Um conjunto de segmentos "agrupados", que se comportam como um elemento apenas.
7. **Retângulo**: Uma poli-linha com uma interface de criação muito mais simples.
8. **Texto**[15]: Atualmente é um conjunto de elementos que formam as letras. Tem a vantagem de trabalhar como qualquer elemento mas a desvantagem de não ser profissional.
9. **Cota**: É um conjunto de elementos que permite marcar uma distância entre dois pontos quaisquer do desenho. Está em desenvolvimento.

[15]Não utiliza fontes

As principais operações disponíveis são:

1. **Copiar para a área de tranferência**: Copia uma seleção e suas camadas para a área de transferência.

2. **Colar da área de transferência**: Cola o que estiver na área de transferência para o desenho corrente.

3. **Apagar**: Apaga os elementos selecionados.

4. **Mover**: Move os elementos selecionados.

5. **Copiar e colar no desenho**: Cria cópias da seleção no mesmo desenho.

6. ***Offset* (cópia com distância)**: A partir de uma seleção, cria uma cópia dela na direção do mouse que é paralela à primeira, no caso de um elemento aberto, ou semelhante, se for um elemento fechado.

7. ***Trim* (corte)**: Permite cortar elementos a partir de elementos de referência.

8. ***Fillet* (completar)**: Permite esticar ou cortar elementos até uma intersecção comum e até permite fazer essa "união" com um arco de raio definido pelo usuário.

9. **Estender**: Estende os elementos a partir de elementos de referência.

10. **Espelhar**: Espelha elementos selecionados por um eixo de simetria definido pelo usuário.

11. **Rotacionar**: Rotaciona a seleção de um ângulo dado pelo usuário.

12. **Escalar**: Aumenta ou diminui uma seleção em relação a um ponto de referência.

13. **Esticar**: Permite modificar os elementos "puxando" partes dele, isto é, movendo seus pontos.

14. ***Pan* (Arrastar desenho)**: Permite "puxar" a folha de desenho para observar outra parte do mesmo.

15. ***Zoom***: Permite se aproximar ou se afastar do desenho.

16. **Área**: Calcula a área definida pelo polígono que o usuário especificar.

17. ***Layon/Layoff***: Essa dupla permite exibir rapidamente todas as camadas do desenho ou torná-las invisíveis especificando um elemento delas.

Apesar de não prover suporte aos arquivos de outros software até agora, o Archimedes permite que os desenhos criados sejam exportados para alguns formatos. Como imagens matriciais, é possível exportar o Archimedes para Bitmap e JPEG. Em formatos vetoriais, o Archimedes suporta Portable Document Format (PDF) [1] e Scalable Vector Graphics (SVG) [13], além do seu formato padrão de trabalho que foi definido pela equipe de desenvolvimento e cujo descritor (XML Schema [15]) é distribuído com o software.

Finalmente, o projeto permitiu analisar o desenvolvimento de um software livre e os problemas que podem ser encontrados ao adotar a metodologia de programação extrema. Apesar das dificuldades enunciadas anteriormente, o uso de programação extrema foi essencial para atingir uma velocidade de desenvolvimento suficientemente grande para que o software passe rapidamente a fase perigosa de vida em que não há gente suficiente interessada nele para que a motivação continue existindo. De acordo com o artigo [11] escrito por Danilo Sato, aluno de mestrado, entre diversas equipes de programação extrema analisadas no primeiro semestre de 2006, a do projeto Archimedes apresentou resultados de adoção e qualidade excelentes.

8 Conclusões

Apesar de ter atingido um nível bem avançado para um projeto de conclusão de curso, o Archimedes ainda tem muita coisa para provar. Ultimamente, começaram a surgir diversos pedidos vindos da comunidade de software livre e dos usuários como, por exemplo, o suporte a arquivos de outros software, melhora da velocidade de desenho e a possibilidade de adicionar diversos *plug-ins* que permitam um crescimento acelerado da ferramenta.

Atualmente, a equipe está focada no desenvolvimento do último pedido graças à arquitetura RCP (Seção 4.6 na página 20) apresentada. Apesar do sistema ainda não estar completamente migrado, a maior parte já funciona perfeitamente nesta arquitetura. Falta apenas modificar o sistema para que as funcionalidades existentes atualmente sejam transformadas em *plug-ins*.

Nos próximos meses, o objetivo é atingir uma versão 1.0 que seja estável e pronta para o uso educacional em universidades ou escolas. Para isso, os problemas de performance deverão ser resolvidos assim como a ausência de alguns elementos essenciais para o desenho como curvas de Bézier e blocos. Espera-se atingir esse objetivo em meados de março 2007.

A partir daí, além de dar manutenção para essa versão, a equipe pretende trabalhar para um objetivo maior que inclua o uso do software no mundo profissional, viabilizando a adoção deste por escritórios de arquitetura. Para isso será necessário trabalhar com as questões de compatibilidade com outros sistemas, impressão e até mesmo de uma migração para uma modelagem tridimensional.

A estimativa da equipe é que esta versão 2.0 só deverá atingir um estado maduro em três ou quatro anos caso mantenha-se o número de desenvolvedores e de horas trabalhadas. Levando em consideração que uma grande parte da equipe está se formando, é mais provável que os membros diminuam o seu tempo de trabalho. Isso causaria um aumento no prazo para a maturidade da versão 2.0. Considerando este fato, a equipe iniciou uma busca por possíveis interessados em financiar o projeto, esperando acelerar seu desenvolvimento e, com isso, garantir que ele atinja este nível rapidamente e evitar que ele seja abandonado. Entre os possíveis interessados estão empresas públicas, empresas privadas nacionais e internacionais e sindicatos.

Referências

[1] Portable document format. `http://en.wikipedia.org/wiki/PDF`. The PDF definition published on the wikipedia the 1st of december 2006.

[2] Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.

[3] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, us ed edition, 10 1999.

[4] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change, 2nd Edition.* The XP Series. Addison-Wesley Professional, 2 edition, 11 2004.

[5] Luis Velhos e Jonas Gomes. *Sistemas gráficos 3D.* IMPA, 1 edition, 2001.

[6] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley Professional, 3 edition, 11 2003.

[7] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1st edition, 06 1999.

[8] Erich Gamma, Richard Helm, John Vlissides, and Ralf Johnson. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, first edition, 01 1995.

[9] IBM. Eclipse help. `http://help.eclipse.org/help32/index.jsp`. The eclipse 3.2 help web site. It contains informations about SWT and RCP including their API.

[10] Sun Microsystems. Java 1.5.0 api. `http://java.sun.com/j2se/1.5.0/docs/api/`, 06 2005.

[11] Danilo Sato, Dairton Bassi, Mariana Bravo, Alfredo Goldman, and Fabio Kon. Experiences on tracking agile projects: an empirical study. Accepted for publication, 2007.

[12] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, volume 1. John Wiley & Sons, 1 edition, 08 1996.

[13] World Wide Web Consortium (W3C). Svg definition. `http://www.w3.org/Graphics/SVG/`.

[14] World Wide Web Consortium (W3C). Xml definition. `http://www.w3.org/XML/`.

[15] World Wide Web Consortium (W3C). Xml schema definition. `http://www.w3.org/XML/Schema`.