Programação para Arquitetura

linguagens visuais e textuais em
Projeto Orientado ao Desempenho

Dissertação de Mestrado | 2019
Instituto de Arquitetura e Urbanismo
Universidade de São Paulo

Gabriele do Rosário Landim

Orientador Prof. Dr. David Moreno Sperling
Coorientador Prof. Dr. Cláudio Motta Toledo

UNIVERSIDADE DE SÃO PAULO

Curso de Pós-Graduação - Instituto de Arquitetura e Urbanismo

Programação para Arquitetura: linguagens visuais e textuais em Projeto Orientado ao Desempenho

Versão Corrigida

Dissertação apresentada ao Instituto de Arquitetura e
Urbanismo da Universidade de São Paulo,
para a obtenção do título de Mestre.

Área de Concentração: Arquitetura, Urbanismo e Tecnologia

Linha de Pesquisa: Projeto, Inovação e Sustentabilidade

Gabriele do Rosario Landim

Orientador: Professor Dr. David Moreno Sperling
Instituto de Arquitetura e Urbanismo
(IAU-USP)

Coorientador: Professor Dr. Cláudio Fabiano Motta Toledo
Instituto de Ciências Matemáticas e de Computação
(ICMC-USP)

São Carlos, 2019

AUTORIZO A REPRODUCAO TOTAL OU PARCIAL DESTE TRABALHO,
POR QUALQUER MEIO CONVENCIONAL OU ELETRONICO, PARA FINS
DE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE

email: gabriele.landim@gmail.com

Ficha catalográfica elaborada pela Biblioteca do Instituto de Arquitetura e Urbanismo
com os dados fornecidos pelo(a) autor(a)

LL257p Landim, Gabriele do Rosario
Programação para Arquitetura: linguagens visuais e
textuais em Projeto Orientado ao Desempenho /
Gabriele do Rosario Landim; orientador David Moreno
Sperling; co-orientador Cláudio Fabiano Motta
Toledo. -- São Carlos, 2019.
163 p.

Dissertação (Mestrado) - Programa de Pós-Graduação
em Arquitetura e Urbanismo, Arquitetura, Urbanismo e
Tecnologia -- Instituto de Arquitetura e Urbanismo,
Universidade de São Paulo, 2019.

1. Programação. 2. Projeto Orientado ao
Desempenho. 3. Design Computacional. 4. Projeto
Algorítmico. 5. Performance. I. Sperling, David
Moreno, orient. II. Toledo, Cláudio Fabiano Motta,
co-orient. III. Título.

Bibliotecária responsável pela estrutura de catalogação da publicação de acordo com a AACR2:
Brianda de Oliveira Ordonho Sígolo - CRB - 8/8229

FOLHA DE JULGAMENTO

Candidato(a): **Gabriele do Rosário Landim**

Título da dissertação: **"Programação para Arquitetura: linguagens visuais e textuais em Projeto Orientado ao Desempenho".**

Data da defesa: **13/05/2019**

Orientador: Prof. Dr. David Moreno Sperling

Comissão Julgadora:

Resultado:

Prof. Dr. David Moreno Sperling
(IAU/USP)

Não votante

Prof. Dr. Marcio Minto Fabricio
(IAU/USP)

APROVADA

Prof. Dr. André Luis de Araujo
(UFU)

APROVADA

p/

Profa. Dra. Maria Gabriela Caffarena Celani
(FEC/UNICAMP)

APROVADA

Coordenador e Presidente da Comissão de Pós-Graduação do Programa de Pós-Graduação em Arquitetura e Urbanismo: **Prof. Dr. Tomás Antonio Moreira.**

À Alexandra Elbakyan e ao
acesso livre ao conhecimento científico

Agradecimentos

Ao Professor David Sperling, por ter acolhido e orientado esta pesquisa. Pela acessibilidade e imensa gentileza com que trata a todos em sua volta.

Ao Professor Cláudio Toledo pela abertura e todas as contribuições que abriram caminhos preciosos durante este período.

À FAPESP e CAPES pelo apoio financeiro no desenvolvimento desta dissertação, processo nº 2016/05427-0 Fundação de Amparo à Pesquisa do Estado de São Paulo e financiamento para o período de estágio de pesquisa no exterior, processo nº 2017/26553-7 Fundação de Amparo à Pesquisa do Estado de São Paulo.

À Professor Antônio Leitão, Inês Caetano e Catarina Belém pela orientação e contribuições significativas no desenvolvimento do estudo de aplicação desta pesquisa.

À Jean Amaro pelo interesse em participar no desenvolvimento das pesquisas relacionadas a esta dissertação.

À Marcelo Bernal e John Haymaker da Perkins+Will pela permissão em utilizar este estudo de aplicação e compartilhamento de todo o material e informações necessárias.

À Mara e Flávia pela ajuda e trabalho impecável que exercem na Secretaria de Pós-Graduação deste Instituto.

Aos desenvolvedores e mantenedores do Sci-Hub, LibGen e Wikipedia pelas iniciativas de tamanho impacto e contribuição ao conhecimento humano.

À Rodolfo Sertori, Maurício Silva e Jéssica Tardivo pela prontidão em compartilhar informações para o período de estágio no exterior.

À Alexandre Villares e Victor Calixto pelas contribuições, que embora indiretas, foram valiosas em aspectos da vida acadêmica e profissional.

À Luciana Ruggiero, Radharani Rocha e Helder pelo acolhimento e companhia durante as primeiras aventuras em terras estrangeiras.

Aos queridos amigos que conheci em São Carlos e que tanto quero bem, sobretudo Anibal Junior, Jansen Faria, Barbara Helena, Rodrigo Lima, Lucas Melchiori, Rodrigo Schereen, Lucas Prates, Raiane Rosi e Luciana Roca. Agradeço pelos momentos, abraços e brisas compartilhadas.

À Marília Daniela pelos momentos, reflexões e pela revisão e comentários que contribuíram para finalizar este trabalho. À Diego Cintra pela parceria, gentileza e contribuição nas fases finais, cruciais na entrega desta dissertação.

À Roseli Landim e Marcelo Altarúgio pelo apoio incondicional em tudo. Sem eles nada disso teria sido possível.

Por fim e sobretudo à Dyego Digiandomenico, quem assentou os blocos para construir o que há hoje.

RESUMO

LANDIM, Gabriele do Rosário. Programação para Arquitetura: linguagens visuais e textuais em Projeto Orientado ao Desempenho. 163 p. Dissertação (Mestrado em Arquitetura e Urbanismo)- Instituto de Arquitetura e Urbanismo, Universidade de São Paulo, São Carlos, 2019.

A pesquisa está inserida no contexto do Design Computacional, área que explora a computação como uma ferramenta de projeto de arquitetura. Para declarar e desenvolver projetos, é comum a utilização de linguagens de programação visual. No entanto, limitações no uso de linguagens visuais são reconhecidas, como as restrições encontradas na flexibilidade e escalabilidade dos códigos. Para estender o acesso à métodos de resolução de problemas nesses sistemas, usuários frequentemente recorrem ao uso de linguagens de programação textual, que permitem a utilização de mais estruturas de controle. Ainda assim, é comum que arquitetos sejam induzidos a realizar mudanças nos processos para acomodar a falta de controle e resultados inesperados de projeto. Isso acontece em parte pela falta de clareza em como utilizar a programação e todo o potencial dos métodos computacionais. Além disso, há uma falta de evidências empíricas usando projetos reais de arquitetura que apoiem discussões sobre o futuro das ferramentas de programação para arquitetos. Esta pesquisa investiga a comparação entre o uso de linguagens de programação visual e textual e visa sugerir diretrizes para melhorar as interfaces de programação utilizadas atualmente na área. Para isso, realizamos análises baseadas em métricas de avaliação de linguagens de programação e nas Dimensões Cognitivas (GREEN; PETRE, 1996). Adicionalmente, exploramos como estudo de aplicação um Projeto Orientado ao Desempenho, o *Vancouver Academic Building* do escritório americano Perkins+Will. O escritório implementou inicialmente o projeto usando a linguagem visual Grasshopper, com o objetivo de otimizar o design de uma fachada explorando o equilíbrio entre aspectos de iluminância e eficiência energética. Reimplementamos o estudo de aplicação utilizando uma linguagem textual e o ambiente de programação Rosetta (LOPES; LEITÃO, 2011). À medida que procuramos solucionar a lacuna existente na comparação dos fluxos de trabalho entre linguagens visuais e textuais recorrendo a um projeto real, selecionamos este estudo de aplicação porque ele abrange um processo orientado ao desempenho em vez de um projeto apenas paramétrico. Existem diferenças fundamentais encontradas entre estes dois fluxos de trabalho em relação a estratégias de escalabilidade e fluxo de dados, pois o orientado ao desempenho requer mais estruturas de controle como iteração e recursão. Identificamos diferenças relevantes entre ambas as linguagens e o potencial no uso das textuais no aumento significativo da escalabilidade do código e do modelo; aumento da confiabilidade dos resultados de desempenho do projeto e maior controle e clareza do processo utilizado. Apontamos também para as qualidades de interação e expressividade inerente das linguagens visuais. A partir do cruzamento teórico-prático, indicamos diretrizes que compõem em que medida o uso preferencial de linguagens textuais e linguagens híbridas (que integram ambas) podem melhorar a confiabilidade do Projeto Orientado ao Desempenho, assim como a possibilidade de aplicar métodos mais claros e eficazes no desenvolvimento do Projeto Algorítmico.

Palavras-chave: Programação; Projeto Orientado ao Desempenho, Design Computacional; Projeto Algorítmico; Performance.

ABSTRACT

LANDIM, Gabriele do Rosário. *Programming for Architecture: visual and textual languages in Performance-Driven Architectural Design*. 163 p. Dissertação (Mestrado em Arquitetura e Urbanismo) - Instituto de Arquitetura e Urbanismo, Universidade de São Paulo, São Carlos, 2019.

This research is contextualized in the Computational Design field, an area that explores computation as an architectural design tool. To declare and develop projects, it is common to use visual programming languages. However, limitations in the use of visual languages are well known, such as the constraints found in the flexibility and scalability of codes. To extend access to problem-solving methods in these systems, users often resort to the use of textual programming languages, which allow the use of more control structures. However, it is common for architects to be induced to make changes to processes to accommodate the lack of control and unexpected design results. This happens partly due to the lack of clarity in how to use programming and the full potential of computational methods. In addition, there is a lack of empirical evidence using real architectural designs that support discussions about the future of programming tools for architects. This research investigates the comparison between the use of visual and textual programming languages and aims to suggest guidelines to improve the programming interfaces currently used in the area. To achieve this, we perform analyzes based on evaluation metrics of programming languages and Cognitive Dimensions (GREEN; PETRE, 1996). In addition, we explored a Performance-driven Architectural Design as a study case, the Vancouver Academic Building, from Perkins+Will. The office initially implemented the design using Grasshopper visual language to optimize the design of a façade by exploring the balance between aspects of illuminance and energy efficiency. We reimplement the case study using a textual language and the Rosetta programming environment (LOPES; LEITÃO, 2011). As we attempt to solve the existing gap in the comparison of workflows between visual and textual languages using a real project, we select this case study because it covers a performance-oriented process rather than a parametric design only. There are fundamental differences found between these two workflows in relation to scalability and data flow strategies since the performance-driven one requires more control structures such as iteration and recursion. We identified relevant differences between both languages and the potential in the use of textual ones in the significant increase of code and model scalability; increased reliability of project performance results and better control and clarity of the process. We also point to the inherent qualities of interaction and expressiveness of visual languages. From the theoretical-practical cross, we indicate the guidelines that make up the extent to which the preferential use of textual languages and hybrid languages (which integrate both) can improve the reliability of the Performance-driven Design, as well as the possibility of applying clearer and more effective methods in the development of the Algorithmic Design.

Keywords: *Programming; Performance-driven Architectural Design, Computational Design; Algorithmic Design; Performance-oriented Architectural Design.*

Lista de Figuras

Figura 1: Algoritmo descrito em programação textual (Design Script) e programação visual (DynamoBIM). Fonte: adaptado de Modelab - DynamoPrimer http://dynamoprimer.com/index.html	20
Figura 2: Representação dos tipos de espaço de projeto. Fonte: adaptado de Nagy (2016).	32
Figura 3: Esquema conceitual do desempenho entre alternativas de projeto. Fonte: Haymaker et al. (2018)	33
Figura 4: Principais programas de modelagem tridimensional e suas linguagens de programação e sintaxes disponíveis para desenvolvimento. Fonte: adaptado de Villares (2018).	41
Figura 5: Quantidade de tipos de linguagens disponíveis, entre as apresentadas na tabela acima.	43
Figura 6: Quantidade de sintaxes disponíveis como linguagens de programação nos softwares de modelagem 3D.	43
Figura 7: Principais programas de modelagem que disponibilizam linguagens de programação visual. Fonte: adaptado de Villares (2018).	44
Figura 8: Representação de um Grafo Acíclico Dirigido, onde os círculos representam os nós, as arestas as conexões e as setas o fluxo de conexão.	50
Figura 9: Elementos de um Modelo Paramétrico Generalizado, identificados na VPL Grasshopper.	50
Figura 10 (à esquerda): Tipos de iteração permitidos em ambientes de programação VPL. Fonte: adaptado de Janssen e Stouffs (2015).	51
Figura 11 (acima): Tipos de modelagem e as iterações que suportam.	51
Figura 12: Softwares de modelagem tridimensional e TPLs disponíveis. Fonte: adaptado de Villares (2018)	54
Figura 13: sequência de pontos em uma espiral cônica.	56
Figura 14: o código VPL em Grasshopper que produz uma sequência de pontos em uma espiral cônica. Fonte: (Leitão, Santos, 2011).	56
Figura 15: Ambiente de programação do FormWriter. Fonte: GROSS (2001, p. 581).	60
Figura 16: Ambiente de programação Rosetta. Acima: programa AutoCAD (back-end) com algoritmo desenvolvido em JavaScript (front-end). Abaixo: programa Rhinoceros (back-end) com algoritmo em Dr.Racket (front-end). Fonte: LOPES, LEITÃO (2011, p. 200 e 202).	62
Figura 17: TPL DesignScript sendo executada por nós da VPL Dynamo. Fonte: Grupo Sfdug	63
Figura 18: Interface Yeti. Fonte: DAVIS (2013, P. 173).	64
Figura 19: A aplicação web Dexen mostrando o gráfico de Pareto através da otimização pelo Eddex. Fonte: JANSSEN (2015, p. 449)	64
Figura 20: Interface Möbius. Fonte: JANSSEN, LI, MOHANTY (2016, p. 160).	65
Figura 21: interface de programação de Luna Moth. Fonte: ALFAIATE (2017, p. 24)	66
Figura 22: Exemplo da característica de Rastreabilidade de Luna Moth. Fonte: ALFAIATE (2017, p. 29).	66
Figura 23: Dimensões Cognitivas definidas por Green e Petre (1996).	71
Figura 24: Critérios de avaliação de TPLs definido por Sebesta (2011)	71
Figura 25: Critérios de avaliação para desenvolvimento de software proposto por Meyer (1997).	72
Figura 26: Critérios de avaliação de VPLs definido por Kiper, Howard e Ames (1997)	72
Figura 27: Métricas para avaliação de VPLs utilizadas no ensino de arquitetura estabelecido por Aish e Hanna (2017).	72
Figura 28: A dimensão Viscosidade e suas relações com as demais métricas de avaliação de PLs.	75
Figura 29: A dimensão Abstração e suas relações com as demais métricas de avaliação de PLs.	78
Figura 30: Soma de uma lista na VPL Dynamo.	82
Figura 31: Soma de uma lista na VPL Grasshopper.	82
Figura 32: Soma de uma lista usando scripting em Python na VPL Grasshopper.	82
Figura 33: A dimensão Proximidade de Mapeamento e suas relações com as demais métricas de avaliação de PLs.	83
Figura 34: Exemplo de dependências ocultas entre dados de uma planilha. Fonte: adaptado de Green e Petre (1996).	86
Figura 35: A declaração na VPL Grasshopper das mesmas relações existentes na planilha.	86
Figura 36: A dimensão Dependências Ocultas e suas relações com as demais métricas de avaliação de PLs.	86
Figura 37: A dimensão Expressividade e suas relações com as demais métricas de avaliação de PLs.	87
Figura 38: A dimensão Notação Secundária e suas relações com as demais métricas de avaliação de PLs.	89
Figura 39: Comparação entre a legibilidade de um código estruturado (cima) e não estruturado (baixo) na VPL Grasshopper. Fonte: Davis, Burry e Burry (2011)	90
Figura 40: A dimensão Visibilidade e suas relações com as demais métricas de avaliação de PLs.	91
Figura 41: A dimensão Consistência e suas relações com as demais métricas de avaliação de PLs.	93
Figura 42: A dimensão Difusão/Concisão e suas relações com as demais métricas de avaliação de PLs.	95
Figura 43: A dimensão Propensão a Erros e suas relações com as demais métricas de avaliação de PLs.	96
Figura 44: Um exemplo de Verificação de Tipos na VPL Grasshopper.	97
Figura 45: A dimensão Avaliação Progressiva e suas relações com as demais métricas de avaliação de PLs.	98
Figura 46: A relação entre Funcionalidade de Meyer (1997) e Kiper, Howard e Ames (1997).	99
Figura 47: A relação entre Eficiência de Meyer (1997) e Interatividade de Aish e Hanna (2017).	101
Figura 48: Cruzamento e semelhanças encontradas entre CDs e as métricas de avaliação de linguagens dos autores citados.	103
Figura 49: Dimensões Cognitivas e métricas de avaliação que possuem similaridades mais relatadas entre autores.	103
Figura 50: A fachada do edifício existente. Fonte: Perkins+Will	109
Figura 51: Os oito tipos de painéis de irão compor a nova fachada do edifício. Fonte: Perkins+Wills	109
Figura 52: Planta do 3º pavimento. Os ambientes foram separados por zonas térmicas (em vermelho) que serão usadas nas análises de conforto ambiental. Fonte: Perkins+Will.	110
Figura 53: Fachadas do edifício e seus respectivos Grupos de Painéis. Fonte: Perkins+Will.	110

Figura 54: Gráfico de Coordenadas Paralelas referente aos inputs e resultados de todas as alternativas de projeto. Fonte: Perkins+Will	111
Figura 55: Fluxo de trabalho de formulação de decisão de projeto. Fonte: Adaptado de PERKINS+WILL (2017)	112
Figura 56: Camadas da infraestrutura computacional. Fonte: Haymaker et al., 2017	112
Figura 57: Dimensão dos painéis da fachada definidos pela equipe de projeto na planilha	113
Figura 58: Dados dos painéis permitidos serem posicionados em cada zona de fachada	113
Figura 59: Interface Design Explorer. O gráfico de coordenadas paralelas (em cima) mostra todos os parâmetros de projeto e resultados de desempenho. É possível visualizar todas as alternativas de projeto (embaixo)	113
Figura 60: Código do DSC na VPL Grasshopper	114
Figura 61: Esquema de linguagens de programação textuais (à esquerda) suportadas pelo ambiente de programação Rosetta (centro) e programas conectados ao ambiente para receber a saída do modelo tridimensional (à direita). Fonte: citar renata ADA	118
Figura 62: Fluxo de trabalho computacional implementado originalmente pelo escritório Perkins+Will	119
Figura 63: Fluxo de trabalho computacional previsto inicialmente, considerando a substituição ideal da VPL pela TPL	120
Figura 64: Fluxo de trabalho computacional implementado na pesquisa, devido às restrições de conexão entre Rosetta e simuladores	121
Figura 65: Código disponibilizado pelo escritório P+W referente a apenas um dos pavimentos do edifício	122
Figura 66: Código e modelo adaptados para servir de base na comparação da implementação	122
Figura 67: Canvas do código DSC em Grasshopper. As setas vermelhas indicam os locais que possuem Dependências Ocultas	123
Figura 68: O nó Entwine recebe as entradas em ordem cruzada, de acordo com os painéis permitidos por área de fachada. Os inputs precisam ser adicionados manualmente e para parametrizar esta lógica, é necessário reconstruir grande parte do código	123
Figura 69: Funcionalidade da IDE Dr. Racket que indica onde estão localizados os objetos vinculados ao objeto selecionado	125
Figura 70: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico "Mudança na topologia do projeto"	126
Figura 71: Exemplo de sequência da lista de linhas da fachada no código original. A linha 0 está posicionada na extremidade esquerda da fachada	126
Figura 72: Exemplo de sequência de lista de linhas da fachada deslocada (shift). A linha 0 não está posicionada na extremidade da fachada, conforme esperado pelo código VPL	127
Figura 73: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico "Relação e ordenação entre geometria e listas"	128
Figura 75: Trecho de código adicionado ao Grasshopper para importar a geometria criada por Rosetta	131
Figura 76: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico "Comunicação com simuladores"	131
Figura 77: Valores numéricos dos parâmetros no Grasshopper	132
Figura 78: À esquerda, linhas criadas pelo Rosetta para os valores SHGC, U-Value e R-Value. À direita, trecho de código em Grasshopper que transforma a dimensão da linha em valor numérico	133
Figura 79: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico "Inputs numéricos na VPL"	133
Figura 80: Modelos analíticos adaptados a demandas específicas de simulação	134
Figura 81: Exemplo de necessidade de quebra de parametrização de um modelo para exportá-lo de CAD para BIM. Fonte: adaptado de Janssen, Stouffs (2015)	135
Figura 82: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico "Escalabilidade do código e modelo" ..	136
Figura 83: Exemplo de uma zona térmica do edifício	136
Figura 84: Valor com diferença no separador decimal entre a planilha (esquerda) e o resultado da simulação em Grasshopper (direita)	137
Figura 84: Valor com diferença no separador decimal entre a planilha (esquerda) e o resultado da simulação em Grasshopper (direita)	139
Figura 86: Dados na planilha ZoneInput, necessários para realizar a simulação de eficiência energética do edifício	139
Figura 82: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico "Escalabilidade do código e modelo" ..	140
Figura 88: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico "Centralização e controle de dados" ..	140
Figura 83: Exemplo de uma zona térmica do edifício	140
Figura 89: Organização e informação do código VPL criada através dos mecanismos de Notação Secundária disponíveis em Grasshopper	141
Figura 90: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico "Organização e clareza do código"	142
Figura 91: Diagrama de conexão entre os módulos e arquivos necessários para executar o programa em TPL	144
Figura 92: Código DSC em Grasshopper separado por módulos	144
Figura 93: Panorama geral do desempenho da VPL Grasshopper e TPL Racket/Rosetta nos tópicos abordados	145
Figura 94: Desempenho	146
da VPL Grasshopper e TPL Racket/Rosetta nas Dimensões Cognitivas	146
Figura 95: Trecho de código que contém os esboços e anotações dos projetistas na documentação do programa (à esquerda), o código (à direita em cima) e o modelo tridimensional resultante (à direita embaixo). Fonte: LEITÃO; LOPES; SANTOS (2014), p.294	149
Figura 96: Correlação entre a expressão no código e a geometria correspondente no modelo tridimensional. Fonte: LEITÃO; LOPES; SANTOS (2014), p.295	150
Figura 97: Correlação entre o modelo tridimensional e as expressões correspondentes no código. Fonte: LEITÃO; LOPES; SANTOS (2014), p.295	150
Figura 97: Rastreabilidade de Luna Moth. Fonte: ALFAIATE (2017, p. 29)	151

intro

Introdução	19
Impacto das Linguagens de Programação no Processo de Projeto em	
Arquitetura	20
Objetivos.....	23
Metodologia	24
Estrutura da dissertação	25
Notas sobre a linguagem	26

01

1.1 Vantagens do uso do Design Computacional	30
1.1.1 Automação de tarefas e extensão de funcionalidades	31
1.1.2 Modelagem Paramétrica, Generativa e geração de formas complexas...	32
1.1.3 Processo de Projeto Orientado ao Desempenho	33
1.2 Intersecções entre Arquitetura e Programação.....	34
1.3 Considerações Parciais	37

02

2.1 Sistemas de Modelagem Paramétrica (Campo Teórico).....	39
2.2 Linguagens de Programação	41
2.3 Linguagens de Programação Visual (VPLs).....	43
2.3.1 Limitações e escalabilidade	46
2.3.2 Taxonomia dos Sistemas de Modelagem Paramétrica	49
2.4 Linguagens de Programação Textual (TPLs)	53
2.4.1 TPL no ensino de programação para arquitetura.....	57
2.4.2 Transição entre VPL e TPL	58
2.5 Cognição em programação	59
2.6 Interfaces de programação TPL para arquitetura	60
2.7 Considerações Parciais	67

3.1 Critérios de Avaliação de PLs.....	69
3.2 Dimensões Cognitivas	73
3.2.1 Viscosidade (<i>Viscosity</i>)	74
3.2.2 Abstração (<i>Abstraction Gradient</i>)	77
3.2.3 Comprometimento Prematuro (<i>Premature Commitment</i>)	80
3.2.4 Proximidade de Mapeamento (<i>Closeness of Mapping</i>)	81
3.2.5 Dependências Ocultas (<i>Hidden-Dependencies</i>)	85
3.2.6 Expressividade (<i>Role-Expressiveness</i>)	87
3.2.7 Notação Secundária (<i>Secondary Notation</i>)	88
3.2.8 Visibilidade (<i>Visibility</i>)	90
3.2.9 Operações Mentais Difíceis (<i>Hard-Mental Operations</i>)	91
3.2.10 Consistência (<i>Consistency</i>)	93
3.2.11 Difusão/Concisão (<i>Diffuseness/Terseness</i>)	94
3.2.12 Propensão a Erros (<i>Error-Proneness</i>)	95
3.2.13 Avaliação Progressiva (<i>Progressive Evaluation</i>)	97
3.2.14 Outros paralelos.....	99
3.3 Considerações Parciais	103

04

4.2 Estudo de Aplicação	108
4.2.1 <i>Vancouver Academic Building</i>	109
4.2.2 <i>Design Space Construction</i>	111
4.2.3 Implementação do código TPL.....	116
4.2.4 Rosetta	117
4.3 Implementação	118
4.3.1 Substituição de Grasshopper por Rosetta.....	119
4.4 Análise da implementação.....	121
4.4.1 Mudança na topologia do projeto	121
4.4.2 Relação e ordenação entre listas	126
4.4.3 Comunicação com simuladores	128
4.4.4 Inputs numéricos na VPL	132
4.4.5 Escalabilidade do código e modelo	134
4.4.6 Confiabilidade de resultados	136
4.4.7 Centralização e controle de dados.....	139
4.4.8 Organização e clareza do código	141
4.4.9 Manipulação de interface	142
4.4.10 Modularização e versionamento	143
4.5 Comparação entre VPL x TPL	145
4.5.2 Desempenho positivo da TPL Racket/Rosetta	146
4.5.1 Desempenho positivo da VPL Grasshopper (aspectos a melhorar na TPL)	148
Considerações Finais	152
Referências Bibliográficas	155

INTRODUÇÃO

Esta pesquisa está inserida na área de Design Computacional com enfoque na Programação, que permite investigar a adaptabilidade do processo de projeto às necessidades de resolução de problemas. Embora o Design Computacional seja tornado possível pela tecnologia, ele é considerado uma metodologia de resolução de problemas (MILLER, 2016). O arquiteto centrado em projetar a partir do pensamento computacional deverá ser capaz de descrever problemas de projeto e os fluxos de trabalho para auxiliar em suas resoluções (BENJAMIN, 2012; MILLER, 2016). Além disso, devido às potencialidades de ser mediado por regras e estabelecer relações entre todas as partes do projeto, desde relações geométricas até as de desempenho, seu potencial está direcionado em conseguir prover soluções mais completas e otimizar variáveis de maneira holística.

O uso da programação para resolução de problemas na arquitetura e design é uma prática conhecida na área acadêmica desde a década de 1960 (KNUTH, 1969; WEIZENBAUM, 1976; WINOGRAD, 1986; MITCHELL, 1987 apud BURRY, 1997) e usada no processo de projetos arquitetônicos com mais frequência a partir da década de 1990 (NATIVIDADE, 2011).

Consideramos o fluxo de trabalho Projeto Orientado ao Desempenho (*Performance-driven Architectural Design*) processo que enfatiza a otimização de vários desempenhos quantificáveis de edifícios (SHI; YANG, 2013). Trata-se de um processo de projeto integrado que envolve modelagem tridimensional, ferramentas de análise de desempenho e métodos de otimização.

Com os recentes desenvolvimentos dos Sistemas de Design Paramétrico (*Parametric Design Systems*, PDS), os modelos de edifícios podem variar paramétrica e generativamente de acordo com a intenção do projetista. Adicionalmente, a partir do que se conhece como *Algorithmic Design* (TERZIDIS, 2006; GLOBAL; MOLONEY; DONN, 2015; CASTELO BRANCO; LEITÃO, 2017; CAETANO et al. 2018), é possível utilizar a computação como parceira no processo de projeto para gerar ideias e estimular mais variedade de soluções considerando, inclusive, condições de desempenho (SHEA; AISH; GOURTOVAIA, 2005). A essência deste fluxo de trabalho está em combinar ferramentas de análise de simulação e algoritmos de otimização para ajudar a determinar a forma final do projeto. Isso “produzirá ferramentas que ajudam arquitetos, designers e engenheiros a pensar criticamente sobre o desempenho do sistema [...] de diferentes pontos

de vista, como o desempenho estrutural, espacial, custo e fabricação” (SHEA; AISH; GOURTOVAIA, 2005, p. 254, tradução nossa¹).

No entanto, é necessário se atentar para as ferramentas disponíveis e a maneira em que declaramos estes fluxos de trabalho. Basicamente, é necessário utilizar um software de modelagem tridimensional atrelado a um sistema de modelagem paramétrica que permita um processo iterativo para gerar alternativas de projeto. As ferramentas mais utilizadas atualmente abrem as funcionalidades dos softwares através da programação visual e textual. Essas ferramentas e linguagens de programação precisam permitir que o projetista descreva as intenções de projeto de maneira robusta e confiável. (SHEA; AISH; GOURTOVAIA, 2005). Verificaremos, no entanto, que ainda existem desafios significativos para que isso aconteça, pois existem lacunas no uso da programação por arquitetos. Desdobraremos a seguir alguns aspectos que interferem no desenvolvimento do processo de projeto computacional.

Impacto das Linguagens de Programação no Processo de Projeto em Arquitetura

Um dos eventos que impulsionaram a adoção do uso da computação no processo de projeto em arquitetura é o uso de linguagens de programação visual (*Visual Programming Languages* ou VLPs). Trata-se de uma linguagem que permite aos usuários manipular elementos graficamente ao invés de especificá-los textualmente (Figura 1) e está presente em muitas áreas de conhecimento. Sua popularização na arquitetura aconteceu principalmente após o lançamento dos programas mais usados até agora, como GenerativeComponents (Bentley), lançado em 2003 (AEC MAGAZINE, 2005), Grasshopper (McNeel), lançado em setembro de 2007 e Dynamo para o programa Revit (Autodesk), lançado em 2011. O uso desses programas tornou a computação acessível a um público mais vasto de arquitetos não programadores (AISH, 2013).

Figura 1: Algoritmo descrito em programação textual (Design Script) e programação visual (DynamoBIM). Fonte: adaptado de Modelab - DynamoPrimer <http://dynamoprimer.com/index.html>

TEXTUAL

```
myPoint = Point.ByCoordinates(0,0,0,0,0);  
x = 5.6;  
y = 11.5;  
attractorPoint = Point.ByCoordinates(x,y,0,0);  
dist = myPoint.DistanceTo(attractorPoint);  
myCircle = Circle.ByCenterPointRadius(myPoint,dist);
```

VISUAL

¹ Do original em inglês: “[...] will yield tools that help architects, designers and engineers think critically about system, rather than just component, performance from different viewpoints, e.g. engineering performance, spatial performance, cost, and fabrication, throughout design conception of innovative forms.”

Embora publicado há pouco mais de uma década, algumas das inquietações do arquiteto e professor grego Kostas Terzidis, autor do livro “*Algorithmic Architecture*”, permanecem atuais e ainda não foram totalmente esclarecidas nas pesquisas da área de Design Computacional:

Termos, conceitos e processos que são vistos por um projetista humano como inconcebíveis, imprevisíveis ou simplesmente impossíveis podem ser explorados, implementados e desenvolvidos em estratégias de projeto totalmente novas dentro do mundo digital. Em vez disso, o que está acontecendo é o uso de computadores como ferramenta de marketing de formas estranhas cuja origem, processo ou racionalidade de geração são inteiramente desconhecidos e, portanto, são julgadas com base em sua aparência, muitas vezes utilizando textos místicos, enigmáticos ou ofuscentes para explicação. Nas duas últimas décadas, a arquitetura transformou-se de uma profissão de projeto e prática baseada em ferramentas orientadas à mão para um projeto baseado em formas e uma prática global baseada em computador. Essa transformação, embora impressionante, ainda não atingiu todo seu potencial. Parcialmente devido à falta de educação computacional de arquitetos ou à abundância de literatura confusa em design digital, dificilmente há exemplos claros em como usar computadores em todo o seu potencial como ferramentas de projeto (TERZIDIS, 2006, p. 39-40, tradução nossa²).

Existem duas principais limitações no uso de processos computacionais em projetos de arquitetura. A primeira é a restrição geralmente observada em programas CAD (*Computer Aided Design* ou Desenho Assistido por Computador) e BIM (*Building Information Modelling* ou Modelagem de Informação da Construção), que oferecem ferramentas pré-definidas para a construção e edição de geometrias tridimensionais. O arquiteto e professor Mark Burry (1997) define que o software feito sob medida para arquitetos geralmente são prescritivos, ou seja, determinam uma ordem, norma ou regra de uso, e que projetos de arquitetura podem estar sujeitos a essas limitações, mesmo sem intenção. Para evitar ser inadvertidamente restringidos, arquitetos podem aprender programação e personalizar os programas para atender outras necessidades do projeto de arquitetura (BURRY, 1997).

A segunda restrição já considera o uso da programação por arquitetos, principalmente com o uso da programação visual. É comum que o arquiteto se depare com problemas de lógica, de linguagem de programação e de estrutura de dados, motivado por desconhecimento de bases comuns na área da computação. Isso, muitas vezes, o induz a realizar mudanças no processo para acomodar erros de código e resultados inesperados no projeto.

Existem evidências na literatura de arquitetura que indicam a tendência que o projetista tem em modificar esse processo de projeto para acomodar esses problemas com o uso de programas (TERZIDIS, 2006; DAVIS, 2013; LEITÃO, 2013).

² Do original em inglês: “Terms, concepts, and processes that are seen as inconceivable, unpredictable, or simply impossible by a human designer can be explored, implemented, and developed into entirely new design strategies within the digital world. Instead, what is happening is the use of computers as marketing tools of strange forms whose origin, process, or rationale of generation is entirely unknown and so they are judged on the basis of their appearance often utilizing mystic, cryptic, or obfuscating texts for explanation. In the last two decades, architecture has transformed from a manually driven tool-based design and practice profession to a computer-driven form-based design and global practice. This transformation, while impressive, has not yet reached its full potential. Partially because of the lack of computational education of architects or the plethora of confusing literature on digital design, there is hardly any bright examples of using computers in their full potential as design tools.”

O professor português e pesquisador António Leitão (2013), observa que as operações e geometrias primitivas disponíveis em programas CAD podem influenciar o processo de projeto: “os usuários tendem a adotar as técnicas de modelagem que eles sabem que são uma boa aposta para a ferramenta de CAD particular que eles estão usando, ao invés de se concentrar nas técnicas de modelagem que melhorem o problema de projeto que eles têm” (LEITÃO, 2013, p.97, tradução nossa³).

Terzidis (2006) indica que a partir do uso da computação em projetos, a origem, processo ou racionalidade de geração do objeto final podem ser inteiramente desconhecidos (TERZIDIS, 2006, p. 40) e critica a falta de transparência, por entender que muitas vezes os projetos são justificados com base no resultado final, resultado de um processo que não esteve necessariamente dentro dos controles do projetista.

O arquiteto e pesquisador Daniel Davis (2013), ao analisar a flexibilidade da modelagem paramétrica escrita através de linguagens de programação visual, indica que quando o projetista encontra problemas com o código, geralmente precisa escolher entre reconstruir o modelo ou “comprometer suas intenções de projeto para atender às limitações de um modelo paramétrico existente” (DAVIS, 2013, p. 42, tradução nossa⁴).

Limitações similares – que influenciam no processo de projeto – também são relatadas no processo de desenvolvimento de software, através das linguagens de programação textual (SEBESTA, 2011). Por exemplo, algoritmos podem estar limitados por restrições nos tipos de estruturas de controle, estrutura de dados e abstrações devido à linguagem escolhida para o desenvolvimento (SEBESTA, 2011). Isso se dá pela capacidade que as pessoas têm em expressar ideias, ou seja, sua linguagem:

Acredita-se que a profundidade com a qual as pessoas podem pensar é influenciada pelo poder de expressividade da linguagem que elas usam para comunicar seus pensamentos. As pessoas que têm apenas um fraco entendimento da linguagem natural são limitadas na complexidade de seus pensamentos, particularmente na profundidade de abstração (SEBESTA, 2011, p. 20)

Neste sentido:

Uma linguagem de programação é um meio formal para expressar ideias e não apenas uma maneira em que um computador executa operações. Isso significa que as linguagens de programação devem corresponder ao processo de pensamento humano, que inclui a capacidade de combinar ideias simples para formar formas e a capacidade de abstrair ideias complexas para que elas se tornem mais gerais. (LOCKE, 1690, apud LEITÃO; SANTOS, 2011, p. 550, tradução nossa⁵)

³ Do original em inglês: “[...] users tend to adopt the modeling techniques that they know are a good bet for the particular CAD tool they are using, instead of focusing on the modeling techniques that better solve the design problem they have.”

⁴ Do original em inglês: “[...] compromising your design intentions to fit the limitations of an existing parametric model.”

⁵ Do original em inglês: “A programming language is a formal medium for expressing ideas and not just a way to get a computer to perform operations. This means that programming languages should match the human thinking process, which includes the ability to combine simple ideas to form compound ones and the ability to abstract complex ideas so that they become more general (Locke, 1690).”

Isso indica que conhecer mais amplamente a variedade de recursos de linguagens de programação pode aumentar o domínio de processos mentais, ao possibilitar que o programador aprenda outras construções de linguagem (SEBESTA, 2011):

[...] o estudo de conceitos de linguagens de programação constrói uma apreciação de recursos e construções valiosas das linguagens e incentiva os programadores a usá-las, mesmo quando a linguagem utilizada não oferece suporte direto para esses recursos e construções (SEBESTA, 2011, p. 20).

Sendo assim, essa pesquisa busca se aproximar do desenvolvimento prático através de algoritmos descritos por linguagens de programação visuais e textuais em sistemas de modelagem paramétrica e generativa. Rolan Hudson (2010), arquiteto e atualmente professor na Dalhousie University no Canadá, argumenta que o conhecimento atrelado ao Design Computacional não pode ser adquirido somente a partir de revisões de literaturas teóricas:

Vários acadêmicos pediram um vínculo mais forte entre pesquisa e prática, Kvan (2004) sugeriu benefícios no fortalecimento de vínculos entre pesquisa e prática em projeto assistido por computador. Martens et al. (2007) recomendam que os arquitetos precisam se envolver com o software e enfatiza a importância da pesquisa em projetos computacionais (HUDSON, 2010, p. 32-33, tradução nossa⁶).

Somente mais recentemente estão sendo estruturados conhecimentos mais atrelados à prática da programação e computação, com os quais arquitetos esclarecem os pontos críticos das limitações de linguagens e paradigmas de programação (WOODBURY, 2010; JANSSEN; WEE, 2011; DAVIS, 2013; JANSSEN; STOUFFS, 2015).

Neste sentido, reunir detalhes sobre o funcionamento da computação aplicada ao processo de projeto em arquitetura pode contribuir no entendimento de como algoritmos e tipos de linguagens de programação podem apoiar o pensamento do projetista na concepção e resolução de problemas em projetos, tornando o processo de projetar menos restringido por software e com as menores limitações possíveis.

Objetivos

Explorar, avaliar e sugerir diretrizes para interfaces de programação para arquitetura no contexto do Projeto Orientado ao Desempenho.

Desta maneira, essa pesquisa busca através da: (1) revisão de literatura sobre o uso de VPLs e TPLs em arquitetura; (2) embasamento teórico sobre os métodos de cognição para programação; e (3) a experiência dos arquitetos e programadores envolvidos na equipe, a partir da implementação de um estudo de aplicação em TPL comparado a um fluxo de trabalho em VPL; levantar reflexões sobre as estratégias do uso computacional em arquitetura, além de indicar diretrizes para as interfaces de programação para arquitetos.

⁶ Do original em inglês: "Various academics have called for a stronger link between research and practice, Kvan (2004) suggested benefits in strengthening links between research and practice in computer aided design. Martens et al. (2007) recommend that architects need to engage with software and emphasise the importance of research in computers and design."

Metodologia

O tema possui inter-relações complexas e questões que emergem durante a prática. Podemos conduzir a investigação de dentro da prática a partir do que Schön (1984) descreve como a reflexão na ação e reflexão sobre a ação (*reflection-in-action* e *reflection-on-action*). Pesquisadores que trabalham com programação argumentam que tal abordagem pode ser útil onde há um universo de possíveis respostas a uma pergunta (DAVIS, 2013).

Para subsidiar a observação do estado da arte do problema de pesquisa, realizamos a revisão *ad-hoc* (pesquisa exploratória prévia). Foram consultados os principais livros e teses sobre Design Computacional. No caso das teses, foram analisadas prioritariamente as publicações mais recentes. Essa fase ofereceu as bases iniciais para o desenvolvimento da Introdução e Capítulo 1.

Uma investigação de artigos que abordam o uso da programação visual e foi conduzida utilizando as bases comuns de indexação de artigos, como Web of Science, Scopus, ACM Library e principalmente CUMINCAD (*Cumulative Index about publications in Computer Aided Architectural Design*). Novamente, por se tratar de artigos específicos sobre o uso da programação no domínio específico da arquitetura, priorizamos os artigos de publicação mais recente. O objetivo foi identificar pesquisas e relatos de estudos que consideram uma aproximação maior da prática da programação e não apenas com a teoria do Design Computacional. Esses artigos compõem a maior parte da discussão apresentada no Capítulo 2.

Conduzimos também uma revisão bibliográfica em artigos das áreas da Ciências da Programação e Engenharia de Software que tratam sobre metodologias para comparação de linguagens visuais e textuais. Muitos destes artigos discutem os assuntos do ponto de vista cognitivo e prático da programação, e compõem a discussão apresentada no capítulo 3.

Em sua pesquisa sobre o uso de interfaces de programação para usuários finais (*end-user programming*), Blackwell (2006) indica que as reflexões sobre as diretrizes utilizadas por programadores e engenheiros de software para projetar interfaces e linguagens de programação devem ser guiadas por: (1) estudos empíricos; (2) pela participação dos próprios usuários do sistema e (3) pela psicologia da programação.

Desta maneira, essa pesquisa utilizou uma implementação prática para triangular as análises realizadas com a revisão da literatura. Exploramos um estudo de aplicação usando VPL e TPL aplicado em um projeto real. O algoritmo foi desenvolvido inicialmente pelo escritório americano Perkins+Will usando Grasshopper. O objetivo foi otimizar o projeto da fachada explorando o equilíbrio entre aspectos de eficiência energética e iluminância. Em seguida, o projeto foi reimplementado usando a TPL Racket e o ambiente de programação Rosetta (LOPES; LEITÃO, 2011). Selecionamos este estudo de aplicação porque ele abrange todos os estágios do Projeto Orientado ao Desempenho - concepção, simulação e otimização - em vez de um projeto simplesmente paramétrico. Existem diferenças fundamentais encontradas entre os dois fluxos de trabalho em estratégias de escalabilidade e fluxo de dados. Consequentemente, os resultados deste estudo visam orientar os futuros aprimoramentos nas linguagens e interfaces de programação para arquitetura.

Embora essa implementação utilizada como base para uma comparação entre VPL e TPL não tenha sido conduzida como um experimento controlado, ela nos ajuda a observar evidências empíricas sobre a confiabilidade dos fluxos de trabalho em Design Computacional.

Estrutura da dissertação

A dissertação está dividida em quatro capítulos, além da introdução e considerações finais.

No capítulo 1 apresentamos uma breve conceituação do campo de estudo. São abordados os principais usos dos recursos computacionais na arquitetura e suas vantagens nos diferentes processos de projeto, como automação de tarefas, projetos paramétricos e generativos e projetos orientados ao desempenho, objeto de estudo nesta pesquisa. A relação entre a programação específica para o domínio e a arquitetura também é apresentada e visa nivelar as intersecções e limites entre ambas as áreas.

O capítulo 2 aprofunda a investigação sobre o uso da programação na arquitetura e se concentra no deslocamento das discussões de maior peso teórico rumo às pesquisas mais aproximadas do campo da computação e da prática de programação. Desdobramos detalhes sobre o uso da programação visual, assim como suas taxonomias e principais limitações dos sistemas de modelagem. Apresentamos o panorama geral do uso das linguagens textuais e sua capacidade em melhorar a escalabilidade de projetos que se tornam maiores e mais complexos ao longo de seu desenvolvimento. Por fim, reunimos as principais interfaces de programação disponíveis na área que objetivam ampliar as funcionalidades disponíveis nos sistemas de modelagem paramétrica ou oferecer uma abordagem algorítmica no desenvolvimento de projetos.

No Capítulo 3, apresentamos algumas metodologias de avaliação para comparar o uso de linguagens visuais e textuais. Realizamos a análise destas metodologias com base nas Dimensões Cognitivas de Green e Petre (1996), uma estrutura conceitual para avaliar estas linguagens e interfaces.

O Capítulo 4 introduz o Estudo de Aplicação utilizado para triangular a comparação entre linguagens visuais e textuais. Para isso, apresentamos o projeto Vancouver Academic Building do escritório Perkins+Will, implementado inicialmente em Grasshopper. Reimplementamos todo o fluxo de trabalho utilizando a linguagem textual Racket e a biblioteca Rosetta (LOPES; LEITÃO 2011). Analisamos as evidências empíricas que emergiram durante o desenvolvimento com base nas Dimensões Cognitivas discutidas no Capítulo 3. Realizamos a comparação entre os dois tipos de linguagem visando apontar os pontos positivos de cada abordagem, assim como conferir ênfase aos pontos em que as textuais podem concentrar esforços na melhoria das interfaces.

Por fim, apresentamos as considerações finais, onde refletimos sobre as diretrizes gerais para o desenvolvimento das interfaces de programação que estejam mais apropriadas a escalabilidade de projetos, confiabilidade do fluxo de trabalho implementado e sobrevivência do conhecimento adquirido por arquitetos nas áreas de programação.

Notas sobre a linguagem

Serão resumidamente definidos abaixo, os principais significados de termos amplamente usados na área de CAAD (*Computer-aided Architectural Design* ou Projeto de Arquitetura Assistido por Computador).

CAD

Computer-aided design (Projeto Assistido por Computador ou Desenho Assistido por Computador)

Para representar o projeto de arquitetura é preciso o uso de programas que disponibilizam geometrias tridimensionais. Programas CAD oferecem primitivas de sólidos básicos e suas respectivas ferramentas para edição e transformação desses sólidos: “apenas uma pequena minoria da atividade de projeto computacional ocorre fora de um ambiente CAD existente” (STEINFELD; KO, 2013, p.501, tradução nossa⁷).

BIM

Building Information Modeling (Modelagem de Informação da Construção)

Sistemas BIM permitem gerenciar informações entre múltiplos agentes (disciplinas ou áreas) em todos os ciclos e fases de projeto, da concepção à pós-ocupação. Podemos afirmar que o BIM reduz retrabalho através da criação de um repositório centralizado de dados que todas as representações podem recorrer, ou seja, estabelecer os dados uma vez e, teoricamente, ter as atualizações de todo o conjunto quando acontecer alguma modificação no modelo (EASTMAN et al. 2011).

BIM é atualmente referido como: “[...] uma versão melhorada de processos de projeto e representações mais tradicionais, acrescidos de metadados não geométricos” (DRITSAS, 2016, p.612, tradução nossa⁸). É frequentemente considerado como uma plataforma que expande as funcionalidades das ferramentas CAD, por não considerar os objetos modelados como sólidos de geometria, mas sim objetos que possuem diversos metadados:

[...] todas as informações sobre o ciclo de vida do projeto são armazenadas nesses objetos e podem ser usadas para criar documentos relacionados à fabricação, estimativa de custos e até mesmo gerenciamento de construção. As ferramentas CAD não exigem essas informações, mas o BIM faz isso” (FERREIRA; LEITÃO, 2015, p. 636, tradução nossa⁹).

Paramétrico

Arquitetos atribuem uma variedade de significados a “paramétrico” e frequentemente usam os termos “modelagem paramétrica”, “design paramétrico”, “parametrização”, “parametria” e derivados.

⁷ Do original em inglês: “Only a small minority of computational design activity takes place outside of an existing CAD environment.”

⁸ Do original em inglês: “[...] an improved version of rather traditional design processes and representations augmented by non-geometric metadata.”

⁹ Do original em inglês: “[...] all the information about the project life cycle is stored in these objects and can be used to create documents related to fabrication, cost estimation and even building management. CAD tools do not require this information but BIM does.”

Davis (2013) indica que “modelo paramétrico” é um termo controverso, e que mesmo especialistas têm dificuldade em definir de forma articulada uma definição concisa sobre o termo. Ele argumenta que no capítulo “O que é modelagem paramétrica?” do livro de Robert Woodbury (2010) “*Elements of Parametric Design*”, a descrição possui cerca de doze páginas para definir um entendimento geral sobre paramétrico. Porém, o autor¹⁰ apresenta uma definição concisa que mais se aproxima ao que a arquitetura se refere: “um modelo paramétrico é o conjunto de equações que expressam um modelo geométrico como funções explícitas de vários parâmetros” (DAVIS, 2013, p.9, tradução nossa¹¹).

Generativo

Da mesma maneira, o termo “generativo” possui uma ampla gama de aplicações e é frequentemente usado em companhia dos termos referidos acima. Celani e Vaz (2012) o descrevem como:

Os sistemas de projeto generativo foram descritos por Mitchell em seu artigo seminal “Os fundamentos teóricos do projeto arquitetônico assistido por computador” como dispositivos que são capazes de gerar soluções potenciais para um determinado problema. A maioria dos sistemas de design generativo usa a variação paramétrica e/ou as regras de transformação como suas principais estratégias. (CELANI, VAZ, 2012, p. 123, tradução nossa¹²)

Isso significa que sistemas generativos descrevem um processo iterativo gerado através de algoritmos que ajustam os parâmetros e a forma do projeto de acordo com as regras estabelecidas. Algumas das regras normalmente aplicadas ao projeto generativo são *L-systems* (PRUSINKIEWICZ; LINDENMAYER, 2012), gramáticas de formas (MITCHELL, 1990), fractais (MANDELBROT, 1983), autômatos celulares (WOLFRAM, 2002), entre outros.

¹⁰ Davis apresenta uma pesquisa aprofundada sobre o tema em sua tese “*Modelled on Software Engineering: Flexible Parametric Models in the Practice of Architecture*”, 2013.

¹¹ Do original em inglês: “[...] a parametric model is set of equations that express a geometric model as explicit functions of a number of parameters.”

¹² Do original em inglês: “Generative design systems have been described by Mitchell in his seminal paper ‘The theoretical foundations of computer-aided architectural design’ as devices that are capable of generating potential solutions for a given problem. Most generative design systems use parametric variation and/or transformation rules as their main strategies.”

Design Computacional

O engenheiro e professor John Harding (2014) afirma que os termos “Projeto de Arquitetura Assistido por Computador (CAAD)”, “Design Digital” e “Design Computacional” são sinônimos e todos se referem, de maneira geral, a métodos de projeto que utilizam poder computacional em projetos de arquitetura.

Para suportar efetivamente toda a diversidade de representações e operações exigidas pela prática de projeto arquitetônico, um ambiente de projeto computacional deve atender aos seguintes requisitos, muitas vezes concorrentes: (1) Fornecer acesso a um rico conjunto de elementos primitivos: construções geométricas e algorítmicas específicas de domínio que fazem o mundo que define o projeto; (2) Fornecer estruturas de código adequadas: convenções e métodos de abstração embutidos que permitem ao usuário manipular, modificar e estender facilmente o conjunto de elementos primitivos (STEINFELD; KO, 2013, p.500, tradução nossa¹³).

O fluxo de trabalho inicial em Design Computacional compreende partir de programas que oferecem modelagem e manipulação de geometrias, normalmente CAD e BIM. Para o desenvolvimento de algoritmos e regras generativas, são usadas linguagens de programação visual (visual programming languages, VPL) através de editores gráficos de algoritmos como Grasshopper, GenerativeComponents, DynamoBIM, entre outros, e linguagens de programação textual (*textual programming languages*, TPL), em ambientes de programação dentro ou fora dos programas de modelagem.

Para entendermos o significado do termo “Design Computacional¹⁴” buscamos estabelecer a relevância da computação para o projeto de arquitetura.

Com o crescente uso de estratégias computacionais, o ato de projetar, o processo de resolução de problemas e a compreensão e representação de artefatos de projeto mudaram. Através de

¹³ Do original em inglês: “To effectively support the full diversity of representations and operations required by architectural design practice, a computational design environment must meet the following, often competing, requirements: • Provide access to a rich set of primitive elements: domain-specific geometric and algorithmic constructs that make up the world that defines the design. • Provide appropriate code structures: conventions and built-in abstraction methods that allow the user to easily manipulate, modify and extend the set of primitive elements.”

¹⁴ Embora seja possível traduzir o termo “*computational design*” para “projeto computacional”, optamos por usar o termo original por se tratar de uma convenção na área.

métodos de Design Computacional, o entendimento de projeto como produto dá lugar ao projeto como processo.

É comum nos referirmos à computação como qualquer fenômeno físico que ocorre dentro de um computador. Porém, ao contrário desta generalização, Schmitt (1998), Terzidis (2006) e Menges e Ahlquist (2011), discutem papéis distintos no uso do computador. Eles argumentam que é preciso distinguir seu uso como um processo digital na arquitetura e entender a diferença entre computação¹⁵ e informatização¹⁶.

Embora conceitos estabelecidos separadamente para indicar o estado do uso do Design Computacional hoje, ambos, computação e informatização estão intrinsecamente ligados com o conceito geral do uso da computação na teoria e prática arquitetônica. Essa ligação foi percebida desde os primeiros sistemas gráficos de projeto desenvolvidos na década de 1960, ao tentar imitar as possibilidades preditivas de projeto já praticadas no campo da engenharia estrutural.

As capacidades de otimização do projeto se fundiram com a facilidade de automatização dos processos. Desta maneira, os requisitos, restrições e intenções de um problema de arquitetura não são mais tratados como entidades isoladas. O Design Computacional pode resolver problemas de maneira holística pela capacidade de programar a interação entre estas entidades, uma vez que programas de computador podem processar desenhos complexos em soluções arquitetônicas cada vez mais específicas (MENGES; AHLQUIST, 2011).

Sendo assim, o uso da computação na arquitetura abre espaço para metodologias de projeto que fornecem estruturas para negociar a inter-relação de conjuntos de dados de informação que são relevantes para o projeto arquitetônico e com isso gerar, derivar e transformar ordem, forma e estruturas complexas (KOLAREVIC, 2003; MENGES, AHLQUIST, 2011).

1.1 Vantagens do uso do Design Computacional

O uso da programação no processo de projeto em arquitetura pode proporcionar vantagens em diversos aspectos. O desenvolvimento de projetos de maneira tradicional e “manual” (que não utilizam meios digitais ou computacionais) possui liberdade em relação a métodos, mídia, inspirações e níveis de abstração, que ajudam a reduzir a complexidade e tornam possível que o projetista processe e interaja com o problema em evolução (CHASZAR; JOYCE, 2016). No entanto, a maneira tradicional se tornou limitada em: (1) tratar de formas e problemas complexos; (2) velocidade e precisão do desenvolvimento de projetos; e (3) variações possíveis de geometria.

¹⁵ *Computation*: Computação pode ser considerado como determinar algo por métodos matemáticos ou lógicos através do procedimento de cálculo. Trata sobre explorar processos indeterminados, pouco claros e muitas vezes mal formulados, usando “racionalização, raciocínio, lógica, algoritmo, dedução, indução, extrapolação, exploração e estimativa” para a “resolução de problemas, estruturas mentais, cognição, simulação e inteligência baseada em regras” (TERZIDIS, 2006, pg. 57-58, tradução nossa). Neste sentido, Schmitt observa o uso do computador como suporte de projeto que “auxilia os projetistas em áreas onde eles não possuem conhecimento ou competência suficientes” (SCHMITT, 1998).

¹⁶ *Computerization*: A informatização pode ser considerada como o ato de entrar, processar ou armazenar informações em um computador ou sistema computadorizado. Trata da “automação, mecanização, digitalização e conversão” geralmente de “entidades ou processos preconcebidos, predeterminados e bem definidos” (TERZIDIS, 2006, pg. 57-58, tradução nossa). Schmitt (1998) ressalta já em 1998 que o uso do computador como uma ferramenta deve principalmente reduzir os custos, aumentar a qualidade e eliminar atividades realizadas anteriormente apenas por humanos.

1.1.1 Automação de tarefas e extensão de funcionalidades

A primeira grande característica considerada como vantagem do Design Computacional em relação a outros métodos de projeto é a automação¹⁷. Terzidis (2006) considera a automação como um processo de informatização e não de computação, isso significa, que em seu nível mais básico, automação é o processo que usa o computador ou a mecânica para diminuir o uso de mão-de-obra em qualquer processo. Um exemplo, é a geração automática de desenhos técnicos para projetos, no qual infinitos desenhos de corte, planta, elevação e detalhes arquitetônicos podem ser gerados a partir da construção de um único modelo tridimensional. Esse nível de automatização é uma das principais características das plataformas BIM, embora também seja possível automatizar tarefas de desenho em software CAD, desde que o projetista escreva um algoritmo/script¹⁸ que execute esta tarefa.

Além desse aspecto, a possibilidade de personalizar e estender recursos de software CAD e BIM também faz parte das principais vantagens do uso da computação (SANTOS; LOPES; LEITÃO, 2012; LEITÃO, 2013). Isso é possível pela capacidade de programar através das linguagens de programação visuais e textuais em softwares usados na arquitetura que oferecem essa extensão: “[...] o software só pode fornecer a funcionalidade desejada usando scripts ou programação, para preencher as lacunas do conjunto de recursos [...]” (BOEYKENS; NEUCKERMANS, 2006, p.41, tradução nossa¹⁹). Isso beneficia não só grandes equipes de engenharia e arquitetura, mas também equipes menores e projetos de pequena escala, com ganhos de produtividade consideráveis (BURRY; DATTA; ANSON, 2000):

Mesmo antes dessa nova tendência, os empreiteiros de construção tinham desenvolvido suas próprias ferramentas de software “internas” para resolver problemas relacionados à geometria e fabricação e para decompor a geometria em código de máquina, desenhos de produção e planos de montagem. No entanto, os serviços de consultoria e suas ferramentas de software são exclusivos de vários arquitetos conhecidos, mas não acessíveis para escritórios de arquitetura de pequena escala. [...] Ferramentas personalizadas internas permitem que pequenas empresas de arquitetura convençam clientes de suas capacidades tecnológicas: não tem que ser mais propriedade única dos ‘grandes jogadores’. (BRELL-COKCAN et al., 2012, p.455 e 464, tradução nossa²⁰)

A implementação de ferramentas personalizadas através da programação em programas CAD e BIM tem potencial de impacto no processo de projeto ao fazer o software se adaptar mais

¹⁷ É importante considerar que em inglês a palavra “*automation*” descreve o que em português chamamos de “automação” e “automatização”. Embora em português usemos dois termos para descrever este processo, mantemos como referência “automação”, por entendermos que seu conceito geral pode ser encontrado em todos os níveis do projeto digital, desde a automatização de tarefas repetitivas até os processos mais avançados de programação textual, que automatizam processos e tarefas em vários níveis (SANTOS; LOPES; LEITÃO, 2012; DRITSAS, 2016).

¹⁸ Consideramos “algoritmo” como a ordem lógica das operações que serão executadas para se solucionar um problema, e “*script*” como um conjunto de instruções que se utiliza para executar o algoritmo. O *script* pode variar de acordo com a linguagem utilizada. O algoritmo (a lógica) não varia, pois sua lógica pode ser utilizada em diversas linguagens.

¹⁹ Do original em inglês: “*the software can only provide the desired functionality using scripting or programming, to fill in the gaps of the software feature set.*”

²⁰ Do original em inglês: “*Even before this new trend, building contractors had developed their own “in house” software tools to solve geometry and fabrication related problems and to break down geometry into machine code, production drawings and mounting plans. However, consulting services and their software tools are exclusive to a number of well-known architects, but not affordable for small scale architectural offices.*” / “*Inhouse customised tools allow small architecture firms to convince clients of their technological capabilities: Freeform architecture therefore does not have to be the sole property of the big players anymore.*”

à metodologia estabelecida pela equipe e diretrizes de projeto, do que o projeto se adaptar as ferramentas disponíveis dentro desses programas (BURRY; DATTA; ANSON, 2000).

1.1.2 Modelagem Paramétrica, Generativa e geração de formas complexas

Outra principal característica do Design Computacional é a variabilidade de soluções, a capacidade de gerar diferentes variações do mesmo projeto.

Em seu estágio inicial, a geração rápida de alternativas está relacionada à característica mais básica da “parametria”, que segundo Woodbury (2010, p.24) significa estabelecer as relações pelas quais as partes se conectam, permitindo que o sistema projetado se encarregue de manter as concepções consistentes com as relações.

Explorar as combinações de restrições e requisitos que caracterizam o projeto através de sistemas paramétricos oferece ao projetista a possibilidade de exploração de um espaço de projeto (Figura 2) maior, a economia de tempo em realizar modificações no projeto e a inventividade, ou seja, a capacidade de encontrar opções de soluções mais criativas e inovativas, resultado do maior número de opções geradas (KILLIAN, 2006, CORREIA; LEITÃO, 2015; FERREIRA; LEITÃO, 2015; CHASZAR; JOYCE, 2016). Essa abordagem, que podemos considerar como Modelagem Generativa, possui capacidades mais avançadas que a Modelagem Paramétrica, por combinar relações associativas da parametria à geração de geometrias através de regras.

Figura 2: Representação dos tipos de espaço de projeto. Fonte: adaptado de Nagy (2016).

Outra característica é a possibilidade de gerar formas geométricas que não estão diretamente disponíveis em plataformas CAD (LEITÃO, 2013). Mesmo com o uso das ferramentas de desenho mais avançadas, algumas formas são muito complexas para serem geradas manualmente (LEITÃO; CABECINHAS; MARTINS, 2010).

1.1.3 Processo de Projeto Orientado ao Desempenho

A capacidade humana é limitada em analisar e avaliar muitas variáveis e métricas de grandes quantidades de opções de projeto. Mesmo ao usar modelagem generativa para a geração de formas, é necessário estabelecer regras e modelos de desempenho que possam orientar o projetista a analisar criticamente a performance do projeto como um todo (SHEA; AISH; GOURTOVAIA, 2005). Isso é possível através de um fluxo de trabalho que considere concepção, simulação e otimização.

Para tomar decisões, a equipe envolvida no processo precisa entender as metas e preferências do projeto, além de reunir, pesar e documentar a lógica deste processo projetual de forma eficiente e confiável: “atualmente, as equipes de projeto lutam para coletar e comunicar todas essas informações, gerando uma exploração ineficiente e ineficaz do projeto e uma perda de valor potencial” (PERKINS+WILL, 2017, p.4, tradução nossa²¹).

O fluxo de trabalho orientado ao desempenho oferece uma maneira de gerar e analisar alternativas, para que as decisões sejam pautadas em dados de performance entre as variáveis analisadas. A Figura 3 descreve esse conceito em um gráfico de coordenadas paralelas. As alternativas de projeto podem ter desempenhos diferentes em cada uma das variáveis de projeto, na maioria das vezes variáveis concorrentes. Uma função atribui um valor final de desempenho às alternativas de projeto.

Figura 3: Esquema conceitual do desempenho entre alternativas de projeto. Fonte: Haymaker et al. (2018)

O termo Projeto Orientado ao Desempenho (*Performance-driven*²²) (SHEA; AISH; GOURTOVAIA, 2005; OXMAN, 2007; SHI, 2010; SHI; YANG, 2013) se refere ao uso de otimização integrada de vários requisitos de desempenho quantificáveis de edifícios (SHI; YANG, 2013). Esse fluxo de trabalho

²¹ Do original em inglês: “Today, design teams struggle to gather and communicate all of this information, causing inefficient and ineffective design exploration, and a loss of potential value.”

²² Os termos geralmente utilizados são: *Performance-driven Generative Design Methods* (Shea, Aish e Gourtovaia, 2005); *Performance-driven Computational Geometry* (Oxman, 2007), *Performance-driven Architectural Design* (Shi, 2010; Shi e Yang, 2013) e *Performance-oriented Architecture* (Hensel, 2013).

ajuda os arquitetos a explorar muitas possibilidades de projeto que performam melhor em objetivos definidos, mantendo explícito os resultados do desempenho de cada opção.

Um dos impulsos para avaliação de desempenho são os padrões de construção ecológica, como LEED nos EUA, BREEAM no Reino Unido, Passive House na Alemanha, entre outros:

Nos últimos anos, a simulação de desempenho de edifícios tornou-se um campo de pesquisa muito ativo. Por um lado, pesquisadores e especialistas, estão estudando e produzindo ferramentas de simulação mais poderosas. Por outro lado, os profissionais começam a perceber o valor de incorporá-los ao processo de projeto. Essa força combinada impulsiona o projeto arquitetônico voltado para o desempenho. (SHI; YANG, 2013, p. 126, tradução nossa²³)

É este fluxo de trabalho que consideramos na escolha do Estudo de Aplicação utilizado nessa pesquisa. Portanto, analisaremos a comparação entre linguagens de programação visual e textual no contexto do Projeto Orientado ao Desempenho, que possui as fases de concepção, simulação e otimização.

Por demandar muitos requisitos de desempenho quantificáveis para guiar o projeto, este fluxo de trabalho necessita um controle maior do código durante o desenvolvimento dos algoritmos: “o pensamento mais racional e a análise científica são trazidos para o campo do projeto arquitetônico” (SHI; YANG, 2013, p. 125). Desta maneira, as análises realizadas durante e após a implementação do Estudo de Aplicação nos ajudam a ir em direção a um processo de projeto mais transparente, aberto e confiável, em resposta às lacunas e problemas apontados por Terzidis (2006), conforme apresentado na seção de introdução desta dissertação.

1.2 Intersecções entre Arquitetura e Programação

Apesar do crescente desenvolvimento da área de Design Computacional e aumento do número de arquitetos que produzem pesquisas e discussões, é frequente haver dúvidas e questionamentos sobre a relação entre o domínio da arquitetura e da programação, ou suas intersecções e limites. Reuniremos a seguir os principais aspectos encontrados na literatura atual que ajudam a esclarecer os graus de envolvimento do arquiteto com o domínio da programação.

Podemos constatar que esta discussão é presente no ensino da arquitetura desde a transição dos anos 1990 e 2000 (BURRY, 1997; GROSS, 2001). Gross (2001) indicava na época, que o argumento mais comum dos oponentes ao ensino da programação era a afirmação de que qualquer tarefa de programação deve ser deixada para programadores profissionais. Por outro lado, quem defende a prática da programação e sua introdução no ensino de arquitetura alega que é essencial para o processo de projeto entender como os computadores fazem e o que fazem, abrindo em certa instância a “caixa-preta” do uso de computadores, permitindo que arquitetos tenham outros conhecimentos e controles.

²³ Do original em inglês: “In recent years, building performance simulation has become a very active research field. On one hand, researchers and specialists are studying and producing more powerful simulation tools. On the other hand, practitioners start realizing the value of incorporating them into the design process. This combined force pushes performance-driven architectural design to the forefront.” (Shi, Yang, 2010, p. 126)

Burphy (1997) faz uma comparação interessante ao argumentar que introduzir programação básica no currículo do ensino possui um peso semelhante a introdução de análises estruturais básicas, que permite ao arquiteto, além de pré-dimensionar os sistemas estruturais das edificações, ter conhecimentos que o permita dialogar com engenheiros estruturais.

Da mesma maneira, o ensino da programação pode viabilizar um relacionamento mais eficaz com a aplicação da computação e com o funcionamento dos softwares, que por sua vez complementam a expressão das intenções de projeto e a interação com o programador profissional. É neste sentido que Santos, Lopes e Leitão (2012) também afirmam que: “a colaboração entre arquitetos e engenheiros de software deve ser apoiada por um vocabulário comum. Arquitetos sem noções de programação não podem efetivamente descrever suas ideias de projeto para um engenheiro de software” (SANTOS; LOPES; LEITÃO, 2012, p.90, tradução nossa²⁴).

Muitas vezes, o projetista é confrontado com a necessidade do apoio da programação para resolver alguma situação de sua própria concepção de projeto, tendo que resolver o problema com seus próprios conhecimentos de programação ou passar o trabalho para um programador (BURPHY, 1997). Nesta situação, principalmente quando a tarefa se torna muito complexa, o esforço do arquiteto em realizar a implementação de alguma solução pode ser tão custosa que torna o esforço aplicado na resolução do problema muito mais caro do que contornar a situação de outras maneiras (SANTOS; LOPES; LEITÃO, 2012). Da mesma maneira, as linguagens de programação textual mais usadas são geralmente muito complexas para quem não possui um conhecimento adequado na área de computação:

[...] como resultado, a maioria das tarefas de programação que são úteis na arquitetura são passadas para especialistas em informática que sabem muito pouco sobre arquitetura, criando uma incompatibilidade entre as ideias do arquiteto e sua implementação, [geralmente] em alguma linguagem de computador de baixo nível (LEITÃO; CABECINHAS; MARTINS, 2010, p.82, tradução nossa²⁵).

Gross (2001) também indicava que esta é uma preocupação mais acadêmica do que prática. Na prática da arquitetura, principalmente em projetos grandes ou complexos, é comum que o trabalho seja composto por equipes multidisciplinares.

O artigo “*Collaborative Digital Design: when architects meets the software engineer*” (SANTOS; LOPES; LEITÃO, 2012) explicita em detalhes a relação entre as duas áreas, os papéis de arquitetos e engenheiros de software (ou programadores profissionais) e os aspectos de colaboração entre eles:

Um engenheiro de software pode contribuir para o processo de projeto de várias maneiras, incluindo [o desenvolvimento de] ferramentas de modelagem e análise personalizadas, gerenciamento e otimização de scripts feitos por arquitetos ou outros profissionais, mediação ou tradução de programas escritos em uma variedade de idiomas e suporte ao desenvolvimento de programas complexos e algoritmos quando necessário. (SANTOS; LOPES; LEITÃO, 2012, p. 88, tradução nossa²⁶).

²⁴ Do original em inglês: “*Collaboration between architects and software engineers must be supported by a common vocabulary. Architects without programming notions cannot effectively describe their design ideas to a software engineer.*”

²⁵ Do original em inglês: “*As a result, most programming tasks that are useful in architecture are passed on to computer science specialists who know very little about architecture, creating an impedance mismatch between the architect’s ideas and its implementation in some low-level computer language.*”

²⁶ Do original em inglês: “*A software engineer can contribute to the design process in a number of ways, including, custom*

Na maioria das vezes, arquitetos aprendem a programar em linguagens que estão disponíveis dentro de ambientes CAD, como Rhinoscript (disponível no programa Rhinoceros) e AutoLisp (disponível para AutoCAD), que incorporam métodos para trabalhar com geometrias e sólidos, além de algumas outras linguagens visuais como Grasshopper e DynamoBIM e seus respectivos ambientes de linguagens textuais como VBA, Python, C#. Por esta experiência direcionada, pode ser difícil que arquitetos aprendam e programem fluentemente em outras linguagens que possuem sintaxe muito diferentes. Engenheiros de software conhecem uma gama mais ampla de linguagens, e quando houver necessidade de usar uma nova, eles têm muito menos esforço em aprendê-las:

[...] o esforço necessário para um engenheiro de software aprender um algoritmo escrito em uma linguagem ‘estrangeira’ e reescrevê-lo na linguagem exigida pela ferramenta de CAD usada pelo arquiteto, deve ser consideravelmente menor do que o esforço equivalente exigido para a maioria dos arquitetos quando executam a mesma tarefa (SANTOS; LOPES; LEITÃO, 2012, p.92, tradução nossa²⁷).

Em alguns casos, arquitetos podem tornar a escrita de seus programas desnecessariamente complicada, pois tentam contornar as limitações que encontram em softwares CAD, quando não conseguem implementar uma resolução do problema para uma determinada tarefa. Já engenheiros de software possuem metodologias adequadas para resolver problemas complexos de maneira eficiente, produzindo, por exemplo, pequenas aplicações, plugins ou programas, que podem tornar o fluxo de trabalho mais eficiente e permitir que os arquitetos se concentrem em tarefas de sua área (SANTOS; LOPES; LEITÃO, 2012).

Outro fator importante desta colaboração se apresenta durante processos de Projeto Orientado ao Desempenho, que integram simulações e otimização à concepção do projeto. Na prática, muitas vezes é necessário associar aos sistemas CAD outras aplicações, como simuladores diversos e ferramentas de análise (conforme veremos na seção 4.3.1 Substituição de Grasshopper por Rosetta), assim como métodos de otimização que são comuns nas áreas de engenharia. Isso é feito implementando componentes de conexão especializados de acordo com cada programa:

[...] enquanto os arquitetos estão melhor preparados para tomar decisões de arquitetura e projeto, engenheiros de software estão bem preparados para escolher uma tecnologia adequada para um determinado problema de programação ou conjunto de requisitos de software (SANTOS; LOPES; LEITÃO, 2012, p. 94, tradução nossa²⁸).

Ao observar os recentes avanços tecnológicos em diversas áreas, é possível identificar uma tendência no trabalho colaborativo entre arquitetos e profissionais da área das ciências da computação. Porém, para que isso permaneça, é necessário que continue se desenvolvendo uma forte cultura de comunicação entre as áreas (SANTOS; LOPES; LEITÃO, 2012). Uma

modeling and analysis tools, management and optimization of scripts made by architects or other professionals, mediation or translation of programs written in a variety of languages, and support to the development of complex programs and algorithms when necessary.

²⁷ Do original em inglês: “[...] the necessary effort for a software engineer to learn an algorithm written in a “foreign” language and rewrite it in the language required by the CAD tool used by the architect should be considerably smaller than the equivalent effort required for the majority architects for the same task.”

²⁸ Do original em inglês: “While architects are better prepared to make architectural and design decisions, software engineers are well prepared to choose an adequate technology for a given programming problem or set of software requirements.”

progressão nas habilidades de programação de arquitetos também é desejável, para que possam formular e modelar problemas que pretendem abordar de maneira mais eficiente:

No entanto, acreditamos que não é necessário que os arquitetos se tornem especialistas em programação. A ciência da computação é um campo de conhecimento independente com uma cultura, preocupações e objetivos diferentes da arquitetura. Arquitetos podem dominar técnicas de programação até o nível que é suficiente para suas necessidades imediatas (SANTOS; LOPES; LEITÃO, 2012, p. 95, tradução nossa²⁹).

Burry (1997) observou que uma vez que arquitetos estiverem em melhor posição para trabalhar criativamente com o computador, não serão mais dependentes de uma solução genérica de CAAD e terão assumido um grau de simbiose com o engenheiro de software.

1.3 Considerações Parciais

O Design Computacional supostamente melhora a capacidade do projetista em automatizar tarefas, estender funcionalidades dos softwares de modelagem, fazer alterações e negociar a inter-relação de conjuntos de dados de informação que são relevantes para o projeto arquitetônico, melhorando assim sua capacidade de projetar.

É comum que o projeto se adapte as ferramentas disponíveis dentro de softwares de desenho CAD e BIM, uma vez que as funcionalidades disponíveis nesses programas são limitadas. Utilizar programação no processo de projeto significa fazer o software se adaptar mais à metodologia estabelecida pela equipe e às diretrizes de projeto (BURRY; DATTA; ANSON, 2000).

No entanto, arquitetos passaram a transitar em um campo de conhecimento que muitas vezes não faz parte de sua formação profissional, a programação. A discussão da relação entre as áreas de arquitetura e computação sugere que uma progressão nas habilidades de programação de arquitetos é desejável, para que possam formular e modelar problemas que pretendem abordar de maneira mais eficiente. Assim, facilita-se a autonomia do arquiteto no desenvolvimento de projetos que sigam o paradigma de Projeto Orientado ao Desempenho.

Para que arquitetos possam dominar melhor as técnicas de programação para o desenvolvimento de projetos, é necessário aprofundar a investigação e conhecimento sobre o funcionamento dos programas de modelagem paramétrica e linguagens de programação utilizadas neste domínio. A seguir, desdobraremos especificidades dos ambientes de desenvolvimento utilizados em Design Computacional.

²⁹ Do original em inglês: "Nevertheless, we believe that it is not necessary for architects to become programming experts. Computer science is an independent field of knowledge with a different culture, concerns, and objectives, than architecture. Architects can master programming techniques up to the level that is sufficient for their immediate needs."

102

Programação para Arquitetura

“Como as ferramentas computacionais podem invocar uma mentalidade computacional e contribuir para o *design thinking*?”

A partir desta questão central, Robert Aish (2012, p.8³⁰) aponta três temas relevantes que compõem a discussão quando se investiga mais a fundo o Design Computacional³¹:

- (1) O tema do processo de projeto: o uso das linguagens de programação como um conjunto de ferramentas de projeto, qual discutimos principalmente no Capítulo 1.
- (2) O tema da linguagem de programação: a partir de especificidades técnicas, como taxonomias e tipos, qual discutiremos em mais detalhes neste capítulo.
- (3) O tema pedagógico: como arquitetos aprendem a projetar e programar, qual discutiremos mais atentamente no Capítulo 3, com foco nos processos cognitivos da programação.

Durante o desenvolvimento deste capítulo, também transitaremos entre aspectos destes temas. Entendemos que discutir a programação para arquitetura mantendo estas três esferas interligadas contribui para melhorar a compreensão sobre o processo de projeto em Design Computacional e seus possíveis desdobramentos futuros.

2.1 Sistemas de Modelagem Paramétrica (Campo Teórico)

Diversas classificações de tipos de modelagens paramétricas e diferentes abordagens do Design Computacional já foram propostas, sendo que algumas que julgamos referenciais serão retomadas aqui.

Por sua relevância no campo teórico, podemos citar as classificações estabelecidas por Oxman (2006), que com base na análise de modelos existentes na arquitetura propõe uma classificação conceitual para a modelagem do que chama de “design digital”. São eles: (1) Modelos CAD, que consistem em modelo descritivo e modelo geração-avaliação; (2) Modelos de formação, que

³⁰ Do original em inglês: “How can a computational tool invoke a computational mindset and in turn contribute to design thinking?”

³¹ Aish discute especificamente estes temas a partir da linguagem de programação DesignScript.

consistem em modelos de formação topológicos, modelos associativos e modelos baseados em movimento; (3) Modelos Generativos, que consistem em modelos de gramática da forma e modelos evolucionários; (4) Modelos de desempenho, que consistem em modelos de formação baseados em desempenho ; e (5) Modelos compostos.

Hernandez (2006) classifica os tipos de modelagem paramétrica em três categorias: (1) modelo de variação paramétrica; (2) combinação paramétrica e (3) modelo híbrido. Modelo de variação paramétrica é baseado na natureza declarativa dos parâmetros usados para construir a forma, e é a forma mais básica do design paramétrico, pois os componentes geométricos são controlados através da alteração dos valores (dados) dos parâmetros sem alterar a topologia da forma, ou seja, o número de componentes e suas relações. Os modelos de combinação paramétrica, também conhecidos como modelos de geometria associativa, oferecem, segundo o autor, um grau de complexidade maior, ao permitir que componentes geométricos possam ser combinados de diferentes maneiras a fim de criar novas composições de projeto. Já um modelo híbrido é categorizado como uma junção das duas abordagens anteriores, oferecendo uma maneira mais robusta de projetar, embora mais difícil de construir, por exigir uma forte estrutura de dados.

Kotnik (2010) define três níveis principais de utilização computacional: representacional, paramétrico e o algorítmico. Representacional diz respeito a utilização do computador como uma ferramenta de desenho. O nível paramétrico é caracterizado pela utilização de relações dada pelo espectro de possibilidades por meio de variação de parâmetros, ou seja, a variação em um dado de entrada gera um resultado variado na saída. O nível algorítmico vai além dessa manipulação de entrada e saída de dados, e diz respeito à codificação da função que executa os comandos, desta maneira, essa abordagem é “uma sequência de operações algébricas, analíticas e geométricas para a manipulação de dados e sua tradução em propriedades arquitetônicas” (KOTNIK, 2010, p. 9, tradução nossa³²).

Celani e Vaz (2012) reúnem as categorizações definidas por William Mitchell (1978), que consistem em três tipos de modelos: modelos icônicos, analógicos e simbólicos. Os autores explicam que modelos icônicos estão próximos da realidade, enquanto os simbólicos são mais abstratos, o modelo analógico é uma categoria intermediária. Modelos icônicos estão mais próximos da realidade no sentido da representação dos elementos construtivos, por exemplo, as catenárias de Gaudí. Modelos analógicos podem ser representados por linguagens de programação visual, onde os nós são usados para representar e manipular entidades de desenhos. Modelos simbólicos usam símbolos, como números, operadores matemáticos e palavras para descrever e executar operações em entidades de desenho.

As classificações conceituais acima se mostraram importantes e necessárias para o entendimento das novas ideias que surgem na práxis digital, sendo que o fornecimento de um quadro teórico oferece bases mais sólidas para o desenvolvimento da área de Design Computacional.

Em comparação com as classificações da primeira década dos anos 2000, que de maneira geral, são mais conceituais e teóricas, essa pesquisa se baseia nos tipos de modelagem paramétrica que estabelecem um diálogo mais técnico, aproximando-se do funcionamento do programa e da estrutura de dados que opera nos modelos. Desta maneira, focaremos a discussão a partir das Linguagens de Programação disponíveis no domínio da arquitetura.

³² Do original em inglês: “[...] a sequence of algebraic, analytic, and geometric operations for the manipulation of data and its translation into architectural properties.”

2.2 Linguagens de Programação

Uma linguagem de programação (*Programming Language*, PL³³) é um meio formal para traduzir um conceito ou expressar ideias em uma linguagem que o computador entenda e possa executar essas operações (LEITÃO; SANTOS, 2011).

Embora existam uma série de conceitos e paradigmas de programação, este capítulo se concentrará na comparação entre linguagens de programação visual (*Visual Programming Languages*, VPLs) e textuais (*Textual Programming Languages*, TPLs) e focará em discuti-las no domínio do desenvolvimento do projeto de arquitetura.

	Programa/IDE	Linguagem/ Biblioteca	Síntaxe
BIM		Dynamo	nodes
	Revit	Revit Macros	VB.NET
			C#
	ArchiCAD	RevitPythonShell	Python
CAD		GDL	BASIC-like
	Marionette		nodes
	Vectorworks	Vectorscript	Pascal
		Python	Python
	Rhinoceros	Grasshopper	nodes
		Python	Python
		Rhinoscript	VBScript
	Autocad	AutoLISP & VisualScheme	LISP
		VB.NET	BASIC-like
	FreeCAD	Python	Python
	SketchUp	Ruby	Ruby
	Blender	Python	Python
	Bentley	Generative Components	nodes
	Houdini	Houdini	nodes
			Java
	Processing	Processing	Python
	P5*JS (Processing Foundation)	P5*JS	JavaScript
	NodeBox 3	NodeBox	nodes
	NodeBox for OpenGL	Python	Python
		Python	Python
3D	3D Max	MAXScript	MAXScript
	Maya	Python	Python
	Fusion360	Python	Python
		C++	C-like
	Cinema 4D	Python	Python
		Coffee	Coffee
	Unity	C#	C
	Unreal Engine	Blueprints	blocks
		C++	C
	QGIS	Python	Python
	ArcGIS	Python	Python
GIS		Racket	Lisp-like
		Python	Python
	Rosetta	Processing	Java
		Java	Java
OUTRO			

Figura 4: Principais programas de modelagem tridimensional e suas linguagens de programação e sintaxes disponíveis para desenvolvimento. Fonte: adaptado de Villares (2018).

Legenda	
Tipo de modelagem	
	BIM
	CAD
	3D
	QGIS
	OUTRO
Tipo de linguagem	
	visual
	textual

³³ Utilizaremos a sigla PL (*programming languages*), VPL (*Visual Programming Languages*) e TPL (*Textual Programming Languages*) em inglês, por seu uso comum em artigos e pesquisas da área.

Leitão (2013) reúne alguns dos propósitos mais comuns do uso de PLs em arquitetura: (a) automatização de tarefas repetitivas (considerado um dos primeiros aspectos usuais da programação geral (BLACKWELL, 2006); (b) quando o usuário deseja formas geométricas e funcionalidades que não estão disponíveis nas ferramentas padrão dos softwares; (c) quando o usuário deseja usar conceitos de aleatoriedade ou outras regras e variações do mesmo projeto (aspecto fortemente relacionado com o Design Generativo) e (d) quando o projeto precisa responder a uma performance/desempenho.

No desenvolvimento de projetos, arquitetos partem do uso de ferramentas CAD e BIM, pois todo o trabalho depende da modelagem de geometrias e sólidos. A disponibilidade de PLs nesses softwares são conhecidas como Interfaces de Programa de Aplicativos (APIs) e oferecem a possibilidade de trabalhar com programação diretamente no ambiente de trabalho que arquitetos sabem utilizar.

A Figura 4 mostra os principais programas de modelagem tridimensional, as PLs disponíveis em cada um deles e suas respectivas sintaxes³⁴.

A primeira coluna indica que os programas estão separados por grupos genéricos de modelagem que suportam, como CAD, BIM, 3D, QGIS e outros. Existem muitas outras nuances de tipos de modelagem dentro desses programas. Por exemplo, os programas CAD Rhinoceros e AutoCAD diferem em suas capacidades de modelagem. O primeiro suporta a criação de NURBS (*Non-uniform rational basis spline*) e *Mesh* (malha), enquanto os sólidos do AutoCAD são modelados apenas em *Mesh*³⁵. Por sua vez, Blender, que não é um software CAD, suporta ambos os tipos de modelagem NURBS e *Mesh*. No grupo Modelagem 3D, Maya e 3DS Max (Autodesk) suportam o tipo de modelagem baseado em cena (*scene-based*) (JANSSEN; STOUFFS, 2015) e diferem de Houdini, Bentley e Nodebox, programas stand-alone baseados em modelagem processual.

A segunda coluna indica se a linguagem de programação disponível dentro do programa é visual ou textual. A maioria dos softwares disponibiliza o desenvolvimento de programas através das TPLs (Figura 5), o público principal são desenvolvedores de APIs que desejam expandir ou personalizar funcionalidades.

No entanto, as PLs mais utilizadas pelos arquitetos são as de sintaxe “nó” (*node*), as quais chamamos de VPLs. Esses grupos de programas com sintaxe VPL são geralmente utilizados quando se referem a Sistemas de Design Paramétrico (*Parametric Design Systems*, PDS) (AISH; WOODBURY, 2005; AISH; HANNA, 2017).

Já as TPLs, embora atualmente menos utilizadas por arquitetos, fazem parte do tipo de linguagem mais disponível nesses softwares.

O desenvolvimento em Python aparece em 14 das 37 linguagens de programação disponíveis para o desenvolvimento de programas, *plug-ins* e *scripts* dentro destes softwares (Figura 6), um bom indicativo da popularidade de Python.

Alguns dos motivos para essa popularidade podem estar atrelados às definições de uma

³⁴ A sintaxe de uma PL define como um conjunto de palavras e regras podem ser combinados. Um programa sintaticamente válido é aquele que pode ser executado. O que está escrito não precisa ter necessariamente significado: “o programa não precisa fazer nada útil para ser sintaticamente válido, ele só precisa ser interpretável como um programa” (Kaurel, 2016). O significado do código, por sua vez, é atribuído à semântica da linguagem.

³⁵ Mais informações sobre diferenças nos tipos de modelagem NURBS e MESH estão disponíveis no vídeo do arquiteto Milos Dimcic: ProArchitect #004 - Mesh vs NURBS. Disponível em: www.youtube.com/watch?v=50TeShluPQU

Figura 5: Quantidade de tipos de linguagens disponíveis, entre as apresentadas na tabela acima.

Figura 6: Quantidade de sintaxes disponíveis como linguagens de programação nos softwares de modelagem 3D.

linguagem “ideal” para introduzir alunos no aprendizado de PLs. Alguns dos elementos importantes são (GUPTA apud NOONE; MOONEY, 2018): uma sintaxe clara e intuitiva; cobrir todas as construções sintáticas e semânticas³⁶ comuns; ser uma linguagem consistente para manutenção de erros; ser equilibrada em não ter muita abstração e nem muito excesso de significados; ser personalizável para mudanças de necessidades. Ateeq et al. (2014 apud NOONE; MOONEY, 2018) verificaram que Python atende muitos desses requisitos.

A seguir reuniremos detalhes sobre o uso das VPLs no desenvolvimento de projeto. É possível observar porque seu uso se popularizou na área e como as características encontradas nos tipos de modelagem que elas suportam oferecem limitações em alguns tipos de fluxo de trabalho.

2.3 Linguagens de Programação Visual (VPLs)

O uso de linguagens de programação visual (VPLs) na arquitetura e design permitem que tarefas computacionais possam ser realizadas sem a necessidade que o projetista saiba escrever códigos textualmente. Também permitem que geometrias complexas possam ser modeladas sem conhecimentos detalhados sobre o funcionamento interno do software (STEINFELD; KO, 2013), uma vez que esses ambientes fornecem operações geométricas pré-definidas ao usuário.

³⁶ A semântica de uma PL é o significado que as sentenças têm. “É a semântica de uma linguagem de programação que decide o que acontecerá quando um programa sintaticamente válido for executado” (Kaurel, 2016). As semânticas definem o que pode ser expresso em uma linguagem, enquanto a sintaxe define como ela pode ser expressa

Figura 7: Principais programas de modelagem que disponibilizam linguagens de programação visual. Fonte: adaptado de Villares (2018).

Programas de Modelagem 3D e VPLs disponíveis			Legenda
Programa/IDE	Linguagem/ Biblioteca	Sintaxe	
Revit	Dynamo	nodes	Tipo de modelagem BIM CAD 3D
Vectorworks	Marionette	nodes	
Rhinoceros	Grasshopper	nodes	
Bentley	Generative Components	nodes	
Houdini	Houdini	nodes	
NodeBox 3	NodeBox	nodes	

O uso de VPLs são geralmente reconhecidos como bem-sucedidos quando lidam com objetos espaciais concretos e em projetos de simulação, como quando objetos no ambiente digital estão sujeitos a restrições projetadas para imitar o comportamento de leis naturais (responder à gravidade, por exemplo) (WHITLEY, 1997; BOSHERNITSAN; DOWNES, 2004).

As VPLs se tornaram especialmente populares em ambientes de programação específica de domínio (*domain-specific language*), como no caso da arquitetura.

Uma das características que mais atraem e viabilizam sua popularidade é o feedback visual simultâneo dos resultados, tanto de dados como de geometria, com todo o fluxo de operações recalculado e atualizado a cada nova variação de parâmetros (CELANI; VAZ, 2012). Essa exibição visual imediata auxilia o programador no entendimento geral das relações explícitas do código, isso o ajuda a encontrar erros ou detectar comportamentos inesperados mais cedo (BURNETT, et al. 1995). Myers (1990) afirma que em sistemas que precisam de atualização em tempo real, as especificações e manipulações gráficas podem ser mais apropriadas.

Ao analisar as vantagens das VPLs, Celani e Vaz (2012) citam os aspectos psicológicos das Dimensões Cognitivas estudados por Green e Petre (1996), que atribuem a facilidade de projetar nesses ambientes pela menor quantidade de planejamento sintático dessas linguagens. Essa também é a primeira característica de uma VPL definida por Burnett et al. (1995), pois o programador não necessita lidar com ponteiros, declarações, escopo ou variáveis, como geralmente teria de fazer ao programar em uma TPL. Desta maneira, as VPLs permitem que o programador possa demonstrar relações ou transformações de dados e geometrias, em vez de escrevê-las em sequências de comandos e símbolos abstratos, como acontece nas linguagens textuais. Isso atribui uma característica importante à linguagem: a representação explícita de relacionamentos (BURNETT et al., 1995).

Diversos autores argumentam que a programação visual apresenta um formato mais próximo das representações mentais dos problemas formalizados por humanos, principalmente de projetistas, acostumados em criar e organizar informações visuais como fluxogramas, diagramas e layouts (CLARISSE; CHANG, 1986; MYERS, 1990; ERHAN; FLEMMING, 2005). O raciocínio espacial desse tipo de linguagem ajuda o desenvolvimento de ideias abstratas, auxiliando o projetista a organizá-las ao planejar uma tarefa que precisa ser programada (CELANI, VAZ, 2012). Myers (1990)

explica que isso acontece porque o processamento de informações visuais em humanos é otimizado para dados multidimensionais³⁷, ao contrário de sistemas de visualização unidimensionais, como linguagens em forma textual.

No entanto, como afirma Blackwell (2001, 2006), muitos debates na área da ciência da computação apelam para a afirmação de que algum paradigma de programação é mais natural e, portanto, mais adequado para programadores inexperientes do que outros. Existem estudos que demonstram que não há justificativas cognitivas que possam afirmar que algum tipo de programação é universalmente superior por ser mais intuitivo ou natural (GREEN, 1989; MENZIES, 2002; BLACKWELL, 2006). Por exemplo em Whitley (1997), após realizar uma revisão sobre evidências empíricas a favor e contra VPLs, conclui que são benéficas para alguns propósitos ao mostrar melhorar o desempenho humano em algumas tarefas específicas (como o exemplo do sucesso do uso de planilhas), mas conclui que até o momento, não foi possível verificar se o efeito do uso da linguagem é suficientemente grande para ser de interesse prático, uma vez que existe distinção entre significância estatística dos efeitos medidos e benefício percebido (magnitude de um impacto). Conclusões similares ainda são encontradas em literatura mais atual, como em Blackwell (2006).

Outra característica comumente citada sobre VPLs é sua curva de aprendizado menor em relação a outras linguagens de programação. Programar visualmente requer pouca experiência do usuário ou mesmo nenhuma habilidade ou compreensão sobre conceitos de programação (AISH, 2013).

Whitley (1997) também apresenta uma hipótese comumente relatada na computação, que VPLs permitem que usuários concluam suas tarefas mais rapidamente do que notações puramente textuais. O autor refuta esta afirmação ao indicar que se trata apenas de uma especulação geral. Dados empíricos encontrados em Leitão, Santos e Lopes (2012) e Leitão, Fernandes e Santos (2013) mostram que é necessário mais tempo para desenvolver programas complexos em VPLs, uma vez que o tempo gasto aprendendo uma TPL, conhecidas por serem linguagens mais flexíveis, é rapidamente recuperado quando a complexidade do problema se torna suficientemente grande. Neste sentido, Burnett et al. (1995) argumenta que uma resolução possível seja tornar as VPLs adequadas para resolver grandes problemas de programação, no entanto isso exigiria que VPLs passassem a conter complexidades de linguagens que elas mesmas tentam remover ou simplificar. Isso é conhecido como problema de escalabilidade ou problema de aumento de escala.

Escalabilidade diz respeito à capacidade que um sistema tem de estar preparado para lidar ou aceitar situações maiores ou mais complexas mantendo sua eficiência (HILL, 1990). Em projetos de arquitetura, um sistema (programa) é considerado escalável quando é adequado para lidar com problemas que, mesmo iniciando de maneira simples, pode se tornar complexo em diversos domínios:

Conceitos de projeto muitas vezes começam como ideias deselegantes e simples que podem ser facilmente exploradas com modelos leves ou scripts. Mas para serem realizadas como um edifício físico, essas ideias necessariamente têm de ser desenvolvidas em modelos complexos de construção com centenas de milhares de componentes individualmente

³⁷ VPLs são formalmente especificadas como linguagens bidimensionais (KIPPER; HOWARD; AMES, 1997; BURNETT et al., 1995; BOSHERNITSAN, DOWNS, 2004), compostas como fluxogramas ou diagramas de relações.

detalhados. Em alguns casos, o ‘conceito’ é a forma geral do edifício, no entanto, passa-se a ser mais comum que o conceito do projeto possa ser um princípio de engenharia ou um objetivo para se conseguir uma determinada combinação de métricas de desempenho, ou ainda mesmo o uso de um algoritmo generativo distribuído nos componentes individuais. (AISH, 2013, p. 90, tradução nossa³⁸)

As limitações das VPLs também são reconhecidas, segundo Miller (1990) desde o momento em que a variação paramétrica surgiu em software proprietário (MILLER, 1990 apud BURRY, 1997). Há um consenso sobre essas limitações, pois alguns tipos de lógica são difíceis de expressar em sistemas de modelagem associativa (MYERS, 1990; BURRY, 1997; AISH, 2012; DINO, 2012; DAVIS, 2013; AISH, 2013). Na seção a seguir serão discutidos os tipos de limitações geralmente encontradas no desenvolvimento de projetos.

2.3.1 Limitações e escalabilidade

Daniel Davis (2013) reúne e reporta algumas das limitações sob a perspectiva da prática. O autor parte da crítica explícita do texto de Rick Smith (2007) intitulado “*Technical Notes from experiences and studies in using Parametric and BIM architectural software*”.

Rick Smith trabalhou no escritório do arquiteto Frank Gehry no início da década de 1990. Por sua experiência anterior em desenvolvimento de projetos para a indústria aeroespacial, atuou no projeto Barcelona Fish de Gehry usando CATIA, um dos primeiros softwares CAD/CAE/CAM³⁹ usados em projetos de arquitetura (DAVIS, 2013). Smith recebeu o Prêmio de Contribuição de Honra pelo AIA (*American Institute of Architects*) devido à sua contribuição e influência para a transformação na prática da arquitetura.

Davis parte das cinco principais deficiências da modelagem paramétrica estabelecidas por Smith (SMITH, 2007 apud DAVIS, 2013):

1. Modelos paramétricos requerem um grau de planejamento imediato (*front-loading*)

“*Front-loading*” é um termo usado em engenharia de software que significa organizar ou planejar um projeto ou processo, de modo que uma grande parcela da atividade geral ocorra concentrada no início deste processo, ou seja, um planejamento imediato.

Embora isto seja comum em planejamento de projetos que precisam seguir etapas lógicas com programação, é necessário primeiro conceituar o que será modelado, antes mesmo da lógica de construção do algoritmo, seguido dos passos de programar, depurar e testar onde o algoritmo da programação visual pode falhar (DAVIS, 2013).

O problema com este grau de planejamento no início do desenvolvimento do código, não é apenas o investimento cognitivo inicial: “[...] mas sim o risco de que o projetista possa investir tempo na abordagem errada” (SMITH, 2007, p. 2 apud DAVIS, 2013, p. 39, tradução nossa⁴⁰).

³⁸ Do original em inglês: “Design concepts often start as disarmingly elegant and simple ideas which can easily be explored with lightweight models or scripts. But to be realised as a physical building, these ideas necessarily have to be developed into complex building models with hundreds of thousands of individually detailed components.”

³⁹ Computer-Aided Design/ Computer-Aided-Engineering/ Computer-Aided-Manufacturing

⁴⁰ Do original em inglês: “[...] is not the upfront cognitive investment but rather the risk that the designer may invest time on “the wrong approach”

Davis afirma que o planejamento exige que o projetista antecipe (até certo ponto) os parâmetros e a hierarquia de relação entre as funções do modelo, uma vez que a VPL requer uma rigidez lógica das funções, que devem ser explícitas: “Esse planejamento antecipado pode ser um desafio, particularmente em um processo tão notoriamente difícil de antecipar como o processo de projeto (DAVIS, 2013, p. 40, tradução nossa⁴¹).

2. A antecipação da flexibilidade pode ser difícil.

A flexibilidade de um modelo paramétrico descrito em VPL diz respeito a maneira como ele pode acomodar mudanças.

O projetista deve estruturar o modelo considerando possíveis mudanças ou futuras solicitações que poderão ser feitas no projeto. Porém, se esta mudança não tiver sido planejada ou antecipada durante a construção do código, o projetista precisará acomodar esta alteração adicionando funções no modelo (DAVIS, 2013, p. 41). Por exemplo, na adição de uma nova ramificação no modelo paramétrico, o projetista deve se preocupar em extrair do modelo existente um dado ou parâmetro que não estava previsto inicialmente, o que pode acarretar em um algoritmo mais confuso e com maior tempo de processamento. Isso é significativo em modelos complexos e com muitos parâmetros e dados associados.

Isso significa que na teoria, um modelo paramétrico ajuda a reduzir o custo da mudança do projeto, uma vez que ter os parâmetros e funções explícitas exige menos esforço para mudar o projeto do que outros métodos de modelagem:

Ao manter um modelo flexível, o projetista pode se dar ao luxo de fazer mudanças, o que é importante, dada a inevitabilidade da mudança em um projeto de arquitetura. Embora algumas mudanças possam ser planejadas antecipadamente, muitas delas vêm de forças de fora da esfera de influência do projetista. Por exemplo, o cliente pode mudar o escopo de projeto, os políticos podem mudar a legislação, e as forças do mercado podem mudar o preço dos materiais. Outras mudanças podem ocorrer porque o projeto é um processo de geração de conhecimento (DAVIS, 2013, p.36, tradução nossa⁴²).

Antecipar a flexibilidade também pode resultar num paradoxo para o projetista, uma vez que é preciso balancear quais parâmetros e dados precisam estar disponíveis no modelo. Davis alega que todos os parâmetros têm um custo computacional, pois exigem trabalho antecipado para implementar e ainda mais trabalho para manipulá-los. Caso esse parâmetro aberto não seja relevante para o projeto, é muito provável que o esforço concentrado tenha sido em vão: “Assim, um modelo paramétrico ideal engloba todas as variações que o projetista deseja explorar com o conjunto mais conciso de parâmetros possíveis” (DAVIS, 2013, p. 41, tradução nossa⁴³).

⁴¹ Do original em inglês: “This upfront planning can be challenging, particularly in a process as notoriously hard to anticipate as the design process.”

⁴² Do original em inglês: “By maintaining a flexible model the designer can afford to make changes, which is important given the inevitability of change on an architecture project. While some changes can be anticipated and perhaps even front-loaded, many changes come from forces outside the designer’s sphere of influence. For instance, the client can change the brief, politicians can change the legislation, and market forces can change the price of materials. Other changes occur because design is a knowledge generating process.”

⁴³ Do original em inglês: “Thus, an ideal parametric model encompasses all the variations the designer wants to explore with the most concise set of parameters possible.”

Chaszar e Joyce (2016) também afirmam que é a flexibilidade dos PDS que atribuem o sucesso das atividades de concepção de projeto.

3. Mudanças importantes quebram modelos paramétricos

Existem situações que quebram o modelo e requerem que o código seja reconstruído do zero. Davis (2013) considera que uma das principais mudanças que forçam o projetista a refazer o código é a mudança da topologia da modelagem, ou seja, a mudança das propriedades fundamentais da forma.

Embora dependendo da situação essa reconstrução possa ser evitada, é possível gastar muito tempo ajustando os modelos, pois modificar as funções explícitas em uma VPL pode ser mais custoso do que escrever um novo código.

4. As mudanças podem ser difíceis de visualizar.

Um código em VPL é composto principalmente por nós e fios que indicam a direção do fluxo de dados. Em modelos complexos, o usuário muitas vezes pode não perceber visualmente mudanças no código. Davis argumenta que mesmo quando a parte do código que sofreu mudanças está desobstruída e visível na tela, o usuário pode sofrer de cegueira à mudança (*change blindness*) (DAVIS, 2013, p. 44). *Change blindness* é um mecanismo conhecido e estudado na psicologia e neurociência. Davis afirma que em geral, as pessoas não são capazes de detectar mudanças em uma cena visual, ao menos que ela ocorra em uma posição específica da retina onde a cena já estava localizada (DAVIS, 2013, p. 44).

Isso é relevante quando alguma parte do código ou parâmetro foi alterado por outro usuário, no caso de equipes que trabalham em equipe no mesmo código. Caso a alteração feita no arquivo seja um erro ou esteja em desacordo com o projeto, a mudança pode vir à tona numa fase mais tardia, onde o custo de mudança do modelo pode ser ainda mais caro ou acarretar problemas maiores.

5. Reutilizar e compartilhar códigos é problemático.

É comum que o programador principal do código se torne, em certa medida, a pessoa mais familiarizada com a lógica e a construção do modelo, sendo muitas vezes o único capaz de realizar mudanças quando este se torna muito complexo. Isso significa que alguns códigos VPL se tornam difíceis de reutilizar e compartilhar (DAVIS, 2013, p. 45).

Davis cita Kolaricvic et al. ao afirmar que “[...] mesmo as tarefas simples tornam-se cognitivamente exigentes quando um modelo paramétrico não é familiar” (KOLARIC et al., 2010 apud DAVIS, 2013, p. 46, tradução nossa⁴⁴).

A complexidade do modelo, sua edição e reutilização são consideradas um desafio fundamental nas VLPs pela desordem visual inevitável do modelo, resultado da representação complexa de numerosos nós e fios de fluxo de dados (LABELLE; NEMBRINI; HUANG, 2009; DINO, 2012). Isso dificulta a exploração efetiva do projeto e pode tornar o desenvolvimento do código contra-producente (AISH, 2013):

[...] à medida que os programas crescem, a quantidade de conexões entre os componentes torna o programa difícil de entender e até mesmo

⁴⁴ Do original em inglês: “[...] even simple tasks becomes cognitively demanding when a parametric model is unfamiliar.”

recursos como sliders param de funcionar como esperado, devido a problemas de desempenho. (FERREIRA; LEITÃO, 2015, p.636, tradução nossa⁴⁵)

De maneira geral, essas limitações encontradas no uso de VPLs são facilmente identificadas por projetistas que já precisaram desenvolver um projeto usando qualquer das VPLs mais utilizadas como Grasshopper, Dynamo e GenerativeComponents.

Entretanto, em projetos mais complexos que consideram que a concepção do modelo deve estar atrelada a parâmetros e dados de desempenho e performance, na qual a modelagem precisa, por exemplo, suportar a geração iterativa (*loops*) de análises e utilizar algum método de otimização, o problema da falta de flexibilidade se caracteriza como a questão-chave dessa limitação (JANSSEN et al., 2016).

Uma compreensão mais aprofundada sobre o funcionamento das VPLs é necessária. A investigação sobre as classificações das VPLs no processo de projeto de arquitetura é realizada a seguir.

2.3.2 Taxonomia dos Sistemas de Modelagem Paramétrica

Considera-se Sistemas de Design Paramétrico (*Parametric Design Systems*, PDS) (AISH; WOODBURY, 2005; AISH; HANNA, 2017) os programas utilizados para o desenvolvimento de projetos paramétricos e generativos. Manteremos a discussão sobre a taxonomia nos referindo ao tema como PDS nesta seção, a fim de manter a nomenclatura a qual os autores Janssen e Stouffs (2015) se referem.

As VPLs são as sintaxes de linguagem que as PDS disponibilizam para programação. Portanto, posteriormente passaremos a nos referir à linguagem VPL diretamente, em vez dos programas ou sistemas que suportam esse tipo de linguagem.

Janssen e Stouffs (2015) distinguem os tipos de modelagem paramétrica com base na maneira em que suportam iteração. Iteração refere-se a mecanismos que repetem sub procedimentos, como funções recursivas, execuções em *loops*, etc. (JANSSEN; LI; MOHANTY, 2016). Segundo os autores, essa classificação é uma maneira clara e inequívoca de classificar os tipos de modelagem, por informar características específicas de cada método e por não agregar a maioria dos métodos dentro da mesma categoria.

Janssen e Stouffs (2015), Woodbury (2010), Davis (2013) e Aish (2013) adotam um Grafo Acíclico Dirigido (*Directed acyclic graph*, DAG) como uma representação genérica para um modelo paramétrico. O uso de grafos na arquitetura data da década de 1970 (AISH, 2013) a partir da proposta de Christopher Alexander de usar grafos como uma técnica descritiva para modelar a tomada de decisões de projeto. Inicialmente, os grafos ajudaram projetistas a entender a estrutura dos produtos ou sistemas resultantes de seu processo de projeto.

Um DAG consiste em um grafo (uma rede de nós e arestas) que possui uma ordem topológica, uma sequência de vértices, de maneira que cada aresta é direcionada de um vértice a outro e não há possibilidade de retorno de um vértice a ele mesmo. Um DAG é acíclico, portanto não pode ter ciclos ou *loops*.

⁴⁵ Do original em inglês: “[...] as programs grow, the amount of connections between components makes the program difficult to understand and even features such as sliders stop working as intended due to performance issues.”

Figura 8: Representação de um Grafo Acíclico Dirigido, onde os círculos representam os nós, as arestas as conexões e as setas o fluxo de conexão.

DAGs são usados para modelar tipos diferentes de informação e normalmente representam coleções de eventos e sua influência entre esses eventos, por exemplo a relação entre informações contidas numa planilha (SCHMITT, 1998).

Considerando que cada célula da planilha é um vértice (nó) e que a relação entre uma fórmula que usa dados de outra célula é uma aresta que representa uma relação direta entre os vértices, é possível estabelecer uma relação topológica entre as informações, de maneira que todos os valores das células se ajustem quando um dos valores da planilha é alterado. De maneira geral, esse é o mesmo comportamento encontrado em um modelo paramétrico usado em arquitetura, por isso a ciência da computação usa DAGs para descrever as operações em linguagens de programação de fluxo de dados.

Sendo assim, consideramos que um DAG descreve o que Janssen e Stouffs (2015) chamam de Modelo Paramétrico Generalizado (*Generalized Parametric Model*, GPM).

De acordo com os autores e de maneira geral, todos os Modelos Paramétricos Generalizados são compostos pelos itens abaixo. A Figura 9 ilustra esses elementos em um código em Grasshopper.

Figura 9: Elementos de um Modelo Paramétrico Generalizado, identificados na VPL Grasshopper.

a. Nós operacionais: a funcionalidade de um nó é determinada pela função implementada em uma linguagem de programação textual (LEE, MESSERSCHMITT, 1987). A função do nó é restringida por especificidades do programa base em que opera (por exemplo, softwares CAD e BIM específicos como Rhinoceros e Revit).

b. Nós de dados: um nó de dados contém um dado ou uma coleção deles, normalmente resultado de uma saída (output) de uma operação ou armazenado para entrar (input) em uma operação (BURNETT et al., 1995). A

estrutura desses dados pode variar dependendo do sistema ou software base, mas de maneira geral são organizadas como listas planas, listas aninhadas (ou matrizes multidimensionais) e estruturas de dados topológicos (árvores). Em alguns sistemas, os usuários podem manipular a estrutura de dados usando operações ou ferramentas que permitam essa personalização.

c. Arestas ou fios: as arestas representam a direção que o fluxo de dados percorre entre os nós.

d. Execução: a ordem de execução é estabelecida pelo sistema em que o grafo de fluxo de dados está sendo executado. De maneira geral, cada vez que o grafo é executado, os conjuntos de nós operam e resultam em uma saída final. Caso os dados de entrada sejam alterados, o grafo é reexecutado e gera uma nova saída.

e. Iteração (Figura 10): na computação, iteração é o processo de repetição de uma ou mais ações. Em cada repetição (de uma ação), uma ou mais instâncias de iteração podem ser reproduzidas.

Janssen e Stouffs (2015) definem três tipos de iteração em um Modelo Paramétrico Generalizado:

Figura 10 (à esquerda): Tipos de iteração permitidos em ambientes de programação VPL. Fonte: adaptado de Janssen e Stouffs (2015).

Figura 11 (acima): Tipos de modelagem e as iterações que suportam.

Iteração simples: considerado o tipo mais simples de iteração, onde a mesma operação é aplicada sequencialmente em várias entidades. Os

autores dão como exemplo uma operação geométrica de “extrusão” sobre um nó de dados que contém uma lista de polígonos. Quando executado, o nó de operação com a função extrusão itera sobre uma lista e extruda um polígono por vez.

Iteração multi-operação implícita: a iteração se torna mais complexa quando um nó de operação precisa iterar vários valores de entradas. Por exemplo, usando a mesma operação de extrusão com um nó de dados adicional que fornece uma lista com as distâncias em que cada extrusão deve acontecer. Neste caso, a operação itera sobre ambas as listas, relacionando a posição de cada índice, qual contém um dado com o dado da lista correspondente. Neste caso, a maneira como os dados estão estruturados dentro do nó de dados define se a operação irá iterar de maneira correta e resultar em uma saída confiável. É neste sentido que o conhecimento de estrutura de dados e tipos de iteração em linguagens de fluxo de dados importa para a saída correta e verificação das informações usadas no projeto de arquitetura.

Iteração multi-operação explícita: segundo os autores, essa operação “representa explicitamente o processo iterativo usando nós adicionais com semântica especializada que modificam o fluxo de controle” (JANSSEN; STOUFFS, 2015, p. 160, tradução nossa⁴⁶)

Após a identificação dos elementos básicos (a, b, c, d, e) e a divisão de três tipos de iteração suportadas, os autores dividem as modelagens utilizadas na arquitetura em quatro categorias, e as distinguem pela maneira que suportam estas iterações (Figura 11):

Modelagem de objetos (*object modelling*): As modelagens desta categoria não suportam iteração, por exemplo os “componentes dinâmicos” do programa SketchUp (Timble), pois são grupos de geometria com parâmetros definidos.

Modelagem associativa (*associative modelling*): suportam apenas iteração simples (ou iteração de operação única).

Modelagem de fluxo de dados (*dataflow modelling*): suportam apenas iteração multi-operação implícita (ou iteração implícita de múltiplas operações).

Modelagem processual (*procedural modelling*): suportam apenas iteração multi-operação explícita (ou iteração explícita de múltiplas operações).

Cada plataforma disponível para programação visual em arquitetura suporta um tipo de iteração. Os programas de modelagem 3D gerais suportam a Modelagem de Objetos. Os programas BIM, como Revit e ArchiCAD suportam Modelagem Associativa, pois permitem que os objetos modelados possam ser modificados através da mudança de seus parâmetros, sem a necessidade de remodelá-los, uma vez que as famílias e geometrias possuem parametria.

As interfaces de programação visual GenerativeComponents (Bentley) e Grasshopper (para Rhinoceros, da McNeel), suportam iteração multi-operação implícita e podem ser classificados como sistemas que suportam Modelagem de Fluxo de Dados.

As interfaces de programação visual Houdini (SideFX) e DynamoBIM (para Revit e sua versão stand-alone da Autodesk) suportam iteração multi-operação explícita, portanto considerados sistemas de Modelagem Processual, sendo que Houdini suporta iteração usando data sinks e Dynamo suporta recursão.

⁴⁶ Do original em inglês: “[...] explicitly represents the iterative process using additional nodes with specialized semantics that modify the control flow.”

Janssen e Stouffs (2015b) afirmam que as modelagens de Fluxo de Dados e Processual são de fato as mais poderosas e exemplificam que esta última pode ser mais sucinta na resolução de problemas.

A partir deste detalhamento sobre as VPLs em funcionamento nos programas paramétricos mais utilizados em arquitetura, é possível compreender que existem diferenças entre os níveis de programação que é possível atingir dentro de cada um deles. Davis (2013) afirma que quando o projetista encontra problemas com a flexibilidade do código VPL, geralmente precisa escolher entre reconstruir o modelo ou “comprometer suas intenções de projeto para atender às limitações de um modelo paramétrico existente” (DAVIS, 2013, p. 42, tradução nossa⁴⁷).

É reconhecido que alguns tipos de lógica são difíceis de expressar e implementar, e é através do uso de TPLs que as limitações das VPLs podem ser superadas (LEITÃO; SANTOS, 2011; AISH, 2012; CELANI; VAZ, 2012; AISH, 2013; DAVIS, 2013). Além disso, apesar da grande atratividade visual da VPL, Boshernitsan e Downes (2004) afirmam que não vale a pena evitar totalmente o desenvolvimento de programa usando textos, uma vez que se torna mais difícil de ler e trabalhar em programas escritos em VPL do que TPLs.

Também é geralmente reconhecido que VPLs não são linguagens de programação completas (BOSHERNITSAN; DOWNES, 2004), ou seja, não é possível escrever e manter um programa inteiro em tal idioma. Inevitavelmente, será necessário estender funcionalidades com TPL em algum momento do projeto para permitir expressar, por exemplo, mais estruturas de controle como iteração ou recursão (BURNETT et al., 1995; LEITÃO; SANTOS, 2011).

Na grande maioria dos programas, é possível estender as funcionalidades escrevendo códigos em linguagens de programação textual como VB.NET, C, C#, Python, entre outros (PAYNE; ISSA, 2009), porém, existem dimensões dentro de um software que ainda limitam as funcionalidades que podem ser alcançadas, mesmo usando uma linguagem de programação mais poderosa. Por exemplo, ao programar um nó em VPL, usando as linguagens como VB, C# ou Python, ao tentar, por exemplo, implementar um código recursivo, este código ainda sofrerá restrição de funcionamento, uma vez que Grasshopper só aceita o tipo de modelagem de fluxo de dados (devido à natureza do Grafo Acíclico Dirigido).

É nesse sentido que buscamos descrever mais detalhes em relação ao uso das TPLs no desenvolvimento de projetos de arquitetura.

2.4 Linguagens de Programação Textual (TPLs)

O uso de linguagens de programação textual (TPLs) na arquitetura antecede o uso de VPLs. Celani e Vaz (2012) apontam que um dos primeiros livros práticos para o desenvolvimento de algoritmos em projetos de arquitetura foi o livro *“The Art of Computer Graphics Programming”* de Mitchell, Ligget e Kvan, publicado em 1987. Porém, mesmo que o desenvolvimento de algoritmos em TPLs tenha, naturalmente, antecedido o uso das VPLs e possua mais confiabilidade, foi o acesso às ferramentas de desenvolvimento de projetos em VPLs que impulsionaram o interesse atual dos arquitetos pela programação: “com a recente introdução de ferramentas paramétricas e

⁴⁷

Do original em inglês: “*compromising your design intentions to fit the limitations of an existing parametric model.*”

programação visual em software CAD o interesse em programação textual para o ensino de design generativo está diminuindo (CELANI, VAZ, 2012, p.123, tradução nossa⁴⁸).

A Figura 12 apresenta as linguagens TPL disponíveis para programação nos principais programas de modelagem utilizados em projetos de arquitetura.

Programa/IDE	Linguagem/ Biblioteca	Síntaxe
BIM	Revit Macros	VB.NET
	RevitPythonShell	C#
	ArchiCAD	Python
	GDL	BASIC-like
CAD	Vectorworks	Pascal
	Python	Python
	Rhinoceros	Python
	Rhinoscript	VBScript
	Autocad	AutoLISP & VisualScheme
		LISP
	FreeCAD	VB.NET
		BASIC-like
3D	SketchUp	Python
	Blender	Python
	Ruby	Ruby
	Processing	Python
	Processing	Java
	P5*JS (Processing Foundation)	JavaScript
	NodeBox for OpenGL	Python
	Python	Python
	3D Max	Python
	MAXScript	MAXScript
	Maya	Python
	Python	Python
	Fusion360	Python
	C++	C-like
	Cinema 4D	Python
	Coffee	Coffee
GIS	Unity	C#
		C
	Unreal Engine	Blueprints
OUTRO		blocks
	QGIS	C++
		C
	ArcGIS	Python
		Python
	Rocket	Lisp-like
	Python	Python
	Rosetta	Python
	Processing	Java
	Java	Java

Legenda

Tipo de modelagem

- BIM
- CAD
- 3D
- QGIS
- OUTRO

Figura 12: Softwares de modelagem tridimensional e TPLs disponíveis. Fonte: adaptado de Villares (2018)

Conforme apontam Leitão, Cabecinhas e Martins (2010), existem alguns problemas gerais nas TPLs dentro dessas ferramentas. Por exemplo: no programa BIM ArchiCAD, a linguagem GDL promove maus hábitos de programação, uma vez que herda diretamente do BASIC. No programa Rhinoceros, VBA, VBScript e RhinoScript (versões do VisualBasic), assim como VisualBasic para Revit são consideradas linguagens que escondem muitos conceitos importantes de ciência da computação ao custo de serem rápidas para o desenvolvimento de aplicativos orientados a eventos. No programa Maya, MEL possui uma sintaxe confusa e difícil de aprender e usar por usuários novatos. No programa AutoCAD, AutoLISP (da família LISP) é considerada uma linguagem antiga e sem atualizações, oferecendo características obsoletas.

É fato que as linguagens de script disponíveis para

⁴⁸ Do original em inglês: “[...] with the recent introduction of parametric tools and visual programming in CAD software the interest in textual programming for teaching generative design is decreasing”.

sistemas CAD foram projetadas para usuários que apenas querem mecanizar algumas tarefas e que não têm muito tempo disponível para aprender conceitos de programação ou até mesmo a vontade de fazê-lo. Além disso, muitas dessas linguagens de script foram desenvolvidas há muito tempo, num momento em que o projeto de linguagem de programação ainda estava em sua infância. A consequência é que as linguagens atuais de scripting para CAD não possuem as qualidades pedagógicas necessárias. (LEITÃO; CABECINHAS; MARTINS, 2010, p. 82, tradução nossa⁴⁹)

No entanto, algumas PLs modernas que avançam os problemas de obsolescência e/ou possuem sintaxe mais fáceis de compreender também estão disponíveis, como é o caso de: VisualScheme (LEITÃO; CABECINHAS; MARTINS, 2010) e C++ para AutoCAD; Python para Grasshopper; e C# para Dynamo.

Mesmo assim, o uso de uma PL moderna não é um aspecto suficiente para ajudar o fluxo de trabalho de projeto. O panorama geral sobre o acesso às diferentes linguagens de programação e suas especificidades dentro dos programas de modelagem nos dá pano de fundo para compreender e discutir alguns aspectos do uso de TPLs. Por exemplo:

Steinfeld e Ko (2013) definem que para suportar efetivamente a diversidade de representações e operações exigidas no desenvolvimento do projeto, um ambiente de programação precisa atender os seguintes requisitos:

(1) Possuir um rico conjunto de elementos primitivos: construções geométricas, sólidos e algoritmos específicos do domínio da arquitetura.

A maioria das PLs utilizadas por arquitetos opera dentro de programas CAD e BIM, que por sua vez já possuem operações de modelagem disponível nativamente. Isso significa que as TPLs acessam essas operações geométricas do software diretamente. Quando ambientes de programação não estão atrelados a esses softwares de modelagem, é necessário construir ou utilizar bibliotecas que constroem os conjuntos de elementos primitivos (como é o caso de Rosetta, que será abordado na seção 4.2.4 Rosetta).

(2) Fornecer convenções, métodos de abstração e estruturas de código adequadas que permitam que o usuário manipule, modifique e estenda o conjunto de elementos primitivos.

(3) Disponibilizar a programação em linguagens modernas: PLs de alto nível geralmente contêm uma série de recursos que aproximam a sintaxe e semântica à linguagem entendida por humanos, como compreensão de listas, digitação dinâmica e suporte para múltiplos paradigmas de programação.

Por exemplo, compreensão de listas é uma abstração importante para arquitetos (ou usuários sem formação em ciências da computação). Leitão e Santos (2011) mostram a quantidade de recursos sintáticos que RhinoScript necessita no desenvolvimento de programas, em comparação com Haskell ou Python, que por sua vez utilizam compreensão de listas. Isso ilustra o potencial que PLs modernas tem ao simplificar seu uso, assim, usuários podem se concentrar mais ao problema que desejam resolver, ao invés de dominarem um conjunto grande de conceitos de implementação do idioma.

⁴⁹ Do original em inglês: *"It is a fact that the scripting languages available for CAD systems were designed for users that just want to mechanize some tasks and that do not have a lot of time available to learn programming concepts or even the will to do so. Moreover, many of those scripting languages were developed a long time ago, at a time when programming language design was still in its infancy. The consequence is that current CAD scripting languages do not have the necessary pedagogical qualities."*

Figura 13: sequência de pontos em uma espiral cônica.

A comparação entre a diferença de sintaxe entre RhinoScript e a linguagem moderna Haskell oferecida por Leitão e Santos (2011) pode ser visualizada a seguir. Todos os códigos produzem uma sequência de pontos em uma espiral cônica.

RhinoScript:

```
Function ConicSpiral(Length,N)
  Dim points()
  ReDim points(N-1)
  Dim t, i
  For i=0 To N-1
    t=i*Length/N
    points(i)=Pt(t*Cos(5*t), t*Sin(5*t), t)
  Next
  ConicSpiral=points
End Function
```

Haskell:

```
conicSpiral length n =
  [(t*(cos 5*t), t*(sin 5*t), t) |
   t <- [i*length/n | i <- [0..n]]]
```

Um exemplo equivalente em Grasshopper também ajuda na compreensão do código:

Figura 14: o código VPL em Grasshopper que produz uma sequência de pontos em uma espiral cônica. Fonte: (Leitão, Santos, 2011)

Por este motivo, a pesquisa sobre o uso de TPLs pode ser útil ao arquiteto que necessita entender e decidir sobre quais linguagens e ferramentas podem ser mais adequadas no desenvolvimento de projetos de arquitetura, assim como ao tipo de resolução de problemas que pretende atingir.

2.4.1 TPL no ensino de programação para arquitetura

Na prática dos arquitetos, a programação é uma tarefa que requer um esforço considerável. Durante o ensino de arquitetura, Leitão (2013) observa que os alunos tendem a valorizar o esforço que aplicaram aprendendo uma PL ao utilizar sempre linguagens semelhantes a ela. Da mesma maneira, como as PLs são geralmente específicas para o software que escolheram utilizar, se torna mais difícil que eles programem em uma ferramenta CAD diferente (LEITÃO, 2013). Isso cria o problema de migração, qual o usuário não pode migrar facilmente entre ferramentas de desenho diferentes:

Há pelo menos três aspectos para este problema de migração. O primeiro aspecto é que os alunos tendem a tornar-se viciados na primeira língua que aprendem. Isso significa que os alunos que aprendem, por exemplo, o RhinoScript, não podem programar facilmente, por exemplo, em AutoLISP. Este problema se torna ainda maior para linguagens de programação que têm diferentes paradigmas sintáticos, como acontece entre linguagens de programação visual, como Grasshopper, e linguagens de programação textual, como o RhinoScript. (LEITÃO, 2013, p.96, tradução nossa⁵⁰)

Embora a programação textual pareça dificultar a programação, Saito, Washizaki e Fukazawa (2017) verificaram em um estudo controlado que a diferença de dificuldade em ensinar uma VPL ou TPL a alunos iniciantes não é estatisticamente significativa, e ambos os métodos podem ser aplicados a aprendizes. O que se verificou, no entanto, é que existe uma diferença no efeito da aprendizagem, onde a TPL pode afetar mais a taxa de respostas corretas para os problemas implementados pelos alunos. Ou seja, as VPLs ainda oferecem maior grau de compreensão para quem aprende a linguagem, enquanto as TPLs estão mais adequadas para o aprendizado de maior quantidade de operações.

Para Leitão (2013), durante o ensino de programação a alunos de arquitetura, pode ser mais vantajoso que o curso se concentre nos conceitos de computação e evite ser dependente de capacidades de softwares específicos. Como teste para esse cenário, durante um semestre de curso de programação para arquitetura, o autor e professor selecionou um subconjunto de conceitos da linguagem AutoLISP e um subconjunto dos recursos usados em AutoCAD, e os aplicou em tópicos como: Conceitos Fundamentais, Abstração de Dados, Abstração Funcional, Estado e Aleatoriedade, Estruturas Recursivas de Dados, Geometria sólida construtiva, Transformações Geométricas, Funções de Ordem Superior, Curvas Paramétricas e Superfícies (LEITÃO, 2013).

Após a experiência, a estrutura de aulas forneceu três vantagens importantes (LEITÃO, 2013):

- (1) Mais tempo no ensino de conceitos da ciências da computação e menos tempo em recursos de linguagem específicos;
- (2) Motivação para que os alunos possam estender e reutilizar as funções/códigos definidos,

⁵⁰ Do original em inglês: "There are at least three aspects to this migration problem. The first aspect is that students tend to become addicted to the first language they learn. This means that students that learn, for example, RhinoScript, cannot easily program in, for example, AutoLISP. This problem becomes even bigger for programming languages that have different syntactic paradigms, as it happens between visual programming languages, such as Grasshopper, and textual programming languages, such as RhinoScript."

uma vez que a linguagem LISP trabalha com mecanismos de abstração utilizados em programação funcional;

(3) Reduzir a incompatibilidade entre linguagens quando os alunos migrarem para uma PL diferente, uma vez que o curso se concentrou mais em conceitos da ciências da computação.

Como visto na discussão sobre as taxonomias dos PDS (JANSSEN; STOUFFS, 2015; JANSSEN, 2015) (descrito na seção 2.3.2 Taxonomia dos Sistemas de Modelagem Paramétrica), a maioria dos conceitos utilizados em VPLs baseado em grafos têm relevância limitada quando o usuário aprende uma TPL (JANSSEN; LI; MOHANTY, 2016). Para o autor, seria desejável que usuários especializados em sistemas VPL possam passar para um TPL sem ter que reiniciar completamente seu processo de aprendizagem, ou seja, que um sistema VPL permita que os usuários possam gradualmente aprender conceitos de TPL (JANSSEN; LI; MOHANTY, 2016).

Nesta mesma direção, Aish (2013) argumenta que um sistema que permite a transição “nó para o código” (VPL para TPL), possibilita que arquitetos permaneçam utilizando VPL como ponto de partida e possam migrar para o desenvolvimento de scripts TPL quando for necessário desenvolver lógicas mais complexas. No entanto, o autor também identifica que conceitos adicionais sobre programação serão necessários para programar em TPL.

2.4.2 Transição entre VPL e TPL

Em uma pesquisa que compara os efeitos da aprendizagem de programação com VPL e TPL, Zorn et al. (2013, apud SAITO; WASHIZAKI; FUKAZAWA, 2017), afirmaram que os insumos visuais da VPL aumentaram o interesse dos alunos. Além disso, Noone e Mooney (2018) verificaram que as abordagens *drag-and-drop*⁵¹ e *WYSIWYG*⁵² das VPLs promoveram a criatividade de uma forma que pode não ser possível alcançar com linguagens baseadas em texto. Esse fator é altamente influente para os fluxos de trabalho de arquitetos.

Essa diferença pode acontecer porque a entrada de texto é um método de programação mais realista que a entrada visual. Os alunos podem ter um preconceito de que o texto é mais representativo da programação e é percebido como mais difícil. (SAITO; WASHIZAKI; FUKAZAWA, 2017, p.222, tradução nossa⁵³)

Darling e White (2015 apud NOONE; MOONEY, 2018) realizaram um estudo no qual a VPL Scratch e a TPL Python eram ensinadas em conjunto por um período de dez semanas. Os autores verificaram que o entendimento da TPL ficou muito mais sedimentado, e o processo de transição aumentou a compreensão da Ciência da Computação.

No entanto, as VPLs são geralmente usadas em domínios específicos, enquanto TPLs são mais frequentes para a programação geral. Isso significa que aprender os dois tipos de linguagem ao mesmo tempo pode afetar o domínio da TPL pela grande diferença semântica entre elas, segundo Kaurel (2016). Para tentar mitigar esse problema, o autor indica o uso de linguagens

⁵¹ Arrastar e soltar: o ato de mover um nó para outra parte da tela.

⁵² Sigla para *What You See Is What You Get* (o que você vê é o que você tem). Descreve o fato de o usuário ver o resultado de sua manipulação ao mesmo tempo em que o programa está sendo criado.

⁵³ Do original em inglês: “*This difference may be because the text input is a more realistic programming method than the visual input. Learners may have a prejudice that text is more representative of programming and it is perceived as more difficult.*”

híbridas, que permitem que os usuários transitem entre VPL e TPL sem se preocupar com as diferenças semânticas.

As linguagens híbridas reúnem a navegabilidade dos ambientes visuais às linguagens de programação baseadas em texto. Esse tipo de linguagem permite utilizar as duas abordagens paralelamente. O primeiro estudo sobre o conceito de línguas híbridas foi realizado por Cilliers et al. (2005 apud NOONE; MOONEY, 2018). Koitz and Slany (2014 apud NOONE; MOONEY, 2018) também compararam o uso da PL híbrida Pocket Code com a VPL Scratch, e verificaram que os efeitos da utilização de Pocket Code foram mais benéficos do que a abordagem puramente visual do Scratch.

2.5 Cognição em programação

Nesta seção, alguns aspectos gerais relacionados à cognição em programação serão apresentados. Eles foram baseados em estudos sobre o ensino de programação e nos ajuda a compreender algumas diretrizes gerais sobre a prática de programar. A fim de partirmos de um contexto amplo, ressaltamos que a discussão apresentada nesta seção se refere à programação independentemente de domínio (WINSLOW, 1996).

Para aplicar corretamente uma PL, não é suficiente apenas estudar a sintaxe e semântica da linguagem (fatos básicos, características e regras) (SOLOWAY, 1986; SHACKELFORD, BADRE, 1993; WINSLOW, 1996). Aprender a PL permite que usuários possam gerar declarações sintaticamente válidas, no entanto, a dificuldade é saber onde e como combinar instruções para gerar o resultado desejado (SOLOWAY, 1986; WINSLOW, 1996). Mesmo quando especialistas não sabem mais sobre uma determinada PL do que estudantes, a habilidade que os diferencia no sucesso em escrever um programa é que especialistas tendem a saber mais sobre o domínio do problema e as possíveis resoluções (WINSLOW, 1996).

É amplamente reconhecido que é necessário construir uma representação mental do domínio do problema do mundo real na prática da programação: “[...] os especialistas pensam em termos de algoritmos e não de programas. A tradução real de um algoritmo em um programa em funcionamento é uma tarefa, não um problema” (WINSLOW, 1996). Isso significa que é importante focar em “estratégias de solução de problemas” ao invés de aspectos de sintaxe e semântica da escrita (SOLOWAY, 1986; SHACKELFORD, BADRE, 1993).

Segundo Soloway (1986), para aprender a programar, é necessário aprender a construir mecanismos e explicações. Winslow (1996) afirma que dominar a matemática do ensino médio é um bom indicador de que o programador aprendeu certas habilidades de solução de problemas. O autor também relembra que solucionar problemas através de analogias é uma tarefa comum em muitos níveis de programação, porém, o problema está em escolher a analogia apropriada para essas resoluções.

Há um consenso entre programadores que a construção de um conjunto sistemático de modelos de resolução ajuda a saber onde e como combinar instruções para gerar o resultado desejado.

Soloway (1986) e Winslow (1996) ressaltam a importância em utilizar modelos que possuam abstração - suficientes para uma grande categoria de problemas - para que haja reutilização da experiência: “Uma característica marcante da especialização é a capacidade de visualizar um

Modelos adequados do espaço do problema e os fatos e estratégias de programação são essenciais para transformar um novato em especialista (WINSLOW, 1996). A chave para essa transição - aprender conceitos e técnicas de uma PL e ser capaz de planejar conscientemente a solução de um problema em áreas específicas - está na prática (WINSLOW, 1996).

Apresentaremos a seguir, em ordem cronológica, um panorama geral das interfaces de programação que utilizam TPL ou propõem maneiras mais integradas para a mudança entre VPL e TPLs.

FormWriter (Figura 27) é uma linguagem de programação com ambiente de desenvolvimento que fornece acesso a bibliotecas de geometrias. Segundo o autor, permite aliar simplicidade de escrita de códigos sem tirar o poder da programação textual. A linguagem usa o paradigma de programação funcional. Nesta linguagem os tipos das variáveis não precisam ser declarados, o sistema também fornece feedback visual imediato e estruturas de controle completas, como: “iteração (repetição) e execução condicional (*if-then-else*), bem como recursão; e tipos de dados, incluindo números, símbolos, sequências (listas)” (GROSS, 2001, p.585, tradução nossa⁵⁴). Além disso, FormWriter usa geometria diferencial para executar transposições e rotação de geometrias no modelo: “o programador não precisa recorrer a coordenadas, trigonometria ou equação (cartesiana ou polar) de um círculo: apenas um conhecimento experiencial de senso comum do espaço” (GROSS, 2001, p.578, tradução nossa).

60 | Programação para Arquitetura

Rosetta (LOPES, LEITÃO, 2011)

Rosetta (Figura 16) é um ambiente de programação que permite que os usuários escolham diferentes linguagens de programação (visual e textual) (*front-ends*) e diferentes aplicações CAD e BIM para a modelagem tridimensional (*back-ends*), oferecendo certo grau de liberdade para que o projetista combine as melhores abordagens para o processo de projeto em questão sem perder a portabilidade do código. Isso significa que o algoritmo que descreve o projeto pode ser desenvolvido em uma linguagem única, pois ao invés de utilizar as bibliotecas de geometria específicas do programa base, como RhinoScript para Rhinoceros ou DesignScript para Dynamo, utiliza uma biblioteca implementada pelos autores para construção e edição de sólidos.

Rosetta possui uma biblioteca com os métodos necessários para a construção e edição de geometrias e alguns métodos adicionais que auxiliam o fluxo de trabalho de projeto. Segundo os autores, Rosetta é composto por:

Uma camada de abstração: essa camada engloba a funcionalidades comuns dos programas de modelagem, e incluem (a) construtores de formas, os procedimentos para criar formas geométricas, como círculos e superfícies; e (b) transformações, os procedimentos que aplicam transformações geométricas, incluindo traduções, lofts, extrusões e varreduras.

Uma linguagem de programação intermediária (intermediate programming language, IPL): A IPL utilizada é Racket, uma TPL suportada pelo Dr.Racket, um ambiente de programação pedagógica adequado para o desenvolvimento de linguagens de programação (Lopes, Leitão, 2011):

Como o IPL é usado como um destino de compilação para os front-ends, o foco principal dessa linguagem é fornecer a funcionalidade essencial para o design generativo. Portanto, certos recursos linguísticos, como apresentação (ou seja, sintaxe), não são abordados por essa linguagem. De fato, como a apresentação é um aspecto tão importante de uma linguagem e existem várias sintaxes possíveis, incluindo as visuais e textuais, a única maneira de acomodar todas as possibilidades é adiar a escolha da apresentação para os front-ends, mantendo uma sintaxe abstrata no IPL. A necessidade de uma sintaxe abstrata é uma das razões para apoiar a escolha do Racket como ponto de partida para o IPL. (LOPES; LEITÃO, 2011, p. 198, tradução nossa⁵⁵).

Front-ends: Os front-ends são as linguagens de programação que os usuários podem escolher para escrever os programas. Estão disponíveis, além da própria IPL Racket, outras TPLs como Python, JavaScript e Processing.

Back-ends: Nesta camada estão os programas de modelagem CAD, BIM e de simulação em que se pode mandar o modelo escrito na TPL escolhida. O mesmo código pode produzir saídas de projeto para programas como AutoCAD, Revit, ArchiCAD, Rhinoceros, Robot, entre outros.

O benefício de Rosetta é que o mesmo algoritmo escrito pode ser compilado para mais de um software, como Rhinoceros e AutoCAD, sem a necessidade de reescrever o código. Caso contrário, dois códigos diferentes precisariam ser escritos, cada um utilizando as bibliotecas

⁵⁵ Do original em inglês: “Because the IPL is used as a compilation target for the front-ends, the main focus of this language is to provide the core functionality essential for generative design. Therefore, certain linguistic features, such as presentation (i.e., syntax), are not addressed by this language. In fact, because the presentation is such an important aspect of a language and there are several possible syntaxes, including visual and textual ones, the only way to accommodate all possibilities is to defer the choice of the presentation to the front-ends, while maintaining an abstract syntax in the IPL. The need for an abstract syntax is one of the reasons for supporting the choice of Racket as a starting point for the IPL”.

das TPLs RhinoScript e AutoLISP, o que quebraria a interoperabilidade do algoritmo caso o projetista optasse por trabalhar com outros softwares em momento posterior.

Figura 16: Ambiente de programação Rosetta. Acima: programa AutoCAD (back-end) com algoritmo desenvolvido em JavaScript (front-end). Abaixo: programa Rhinoceros (back-end) com algoritmo em Dr.Racket (front-end). Fonte: Lopes, Leitão (2011, p. 200 e 202)

DesignScript (AISH, 2012)

DesignScript (Figura 17) é uma linguagem de programação desenvolvida por Robert Aish (2012) e atualmente integrada ao ambiente de programação visual Dynamo e DynamoBIM (disponível para Revit). DesignScript é multiparadigma (incluindo paradigmas orientados a objetos, funcionais e associativos); de domínio específico, pois oferece funcionalidades para a geração de modelos geométricos; e, segundo Aish (2012), apropriada ao aprendizado e a escalabilidade, por permitir uma aquisição progressiva de conceitos de programação aplicados ao projeto de arquitetura.

Figura 17: TPL DesignScript sendo executada por nós da VPL Dynamo. Fonte: Grupo Sfdug

Yeti (DAVIS, 2013)

Yeti é um ambiente de desenvolvimento textual interativo (Figura 18) desenvolvido por Davis (2013). A geometria produzida está constantemente em sincronia com a escrita do código, sendo assim, sempre que o código é alterado, a geometria também se altera automaticamente (live programming). Embora seja um ambiente de programação textual, Yeti parece oferecer um auxílio importante para o projetista que precisa migrar de uma VPL para um TPL, pois usa uma linguagem de programação por fluxo de dados para empregar um Grafo Acíclico Dirigido (DAG) para a geração de geometria.

Para realizar cálculos geométricos com rapidez suficiente para que se sintam interativos, Yeti emprega um DAG para coordenar o recalculamento da geometria. Este é essencialmente o mesmo conceito que alimenta a interatividade dos ambientes de programação visual Grasshopper, Houdini e GenerativeComponents. No entanto, o DAG do Yeti não é gerado através de uma interface visual, em vez disso, é definido textualmente através do YAML de formato de dados relacionais. (DAVIS, 2013, p. 168, tradução nossa⁵⁶)

Além disso, Yeti oferece alguns recursos em sua IDE como: (a) auto completar, ao sugerir chaves, nomes e objetos contextualmente relevantes e economizar tempo de pesquisa à extensa documentação da linguagem; (b) manipulação robusta de erros, ao destacar erros à medida que são escritos: “uma vez que o Yeti não usa uma sequência de instruções, mas sim uma linguagem de fluxo de dados, pode continuar a executar o código

⁵⁶ Do original em inglês: “In order to perform geometric calculations fast enough that they feel interactive, Yeti employs a DAG to coordinate recalculating the geometry. This is essentially the same concept powering the interactivity of the visual programming environments Grasshopper, Houdini, and GenerativeComponents. However, Yeti’s DAG is not generated through a visual interface, rather it is defined textually through the relational data format YAML.”

que contém erros, analisando apenas as partes livres de erros do código em um DAG” (DAVIS, 2013, p. 172, tradução nossa⁵⁷); (c) rastreabilidade, visualização da conexão entre código e a geometria; (d) controle de parâmetros, para que o usuário possa inspecionar todos os dados relevantes que controlam a geometria.

Figura 18: Interface Yeti.
Fonte: Davis (2013, P. 173)

Dexen (JANSSEN, 2015) (dexen.org).

Dexen é descrito como uma plataforma desenvolvida para apoiar pesquisadores e projetistas na experimentação de algoritmos na exploração de projeto (Figura 19). A plataforma foi desenvolvida no Design Automation Lab da Universidade Nacional de Cingapura. A principal característica do Dexen é permitir escalabilidade e extensibilidade através do desenvolvimento de algoritmos de otimização. Por ser uma plataforma baseada em MongoDB, suporta as linguagens C, C++, C #, Java, Perl, PHP, Python e Ruby.

Figura 19: A aplicação web Dexen mostrando o gráfico de Pareto através da otimização pelo Eddex.
Fonte: Janssen (2015, p. 449)

⁵⁷ Do original em inglês: “Since Yeti does not use a sequence of instructions but rather a data-flow language, Yeti can continue to run code that contains errors by only parsing the error-free portions of the code into a DAG.”

Möbius (JANSSEN; LI; MOHANTY, 2016)

Möbius é um ambiente de programação web para arquitetura, seu objetivo é superar as limitações existentes das VPLs. Para isso, a interface é compatível com modelagem processual. É possível usar os paradigmas associativos e imperativos. Segundo os autores:

O estilo imperativo de programação é suportado através de programação baseada em blocos, onde o usuário constrói procedimentos através da criação de sequências de blocos de código (Resnick et al., 2014). O estilo associativo de programação é suportado através da programação de fluxo de dados, com a qual o usuário constrói redes que consistem em nós e fios. Cada nó possui um procedimento imperativo que é definido usando a abordagem baseada em blocos. (JANSSEN; LI; MOHANTY, 2016, p. 159-160, tradução nossa⁵⁸).

A partir de uma programação VPL, gera automaticamente um programa textual equivalente, permitindo que os mecanismos de programação mais avançados das TPLs possam ser utilizados.

A interface do sistema permite uma visualização integrada entre linguagem textual (d) e visual (f), além de permitir escrever códigos que suportam loops e funções de ordem superior.

Figura 20: Interface Möbius. Fonte: Janssen, Li, Mohanty (2016, p. 160)

Luna Moth (ALFAIATE, 2017)

Luna Moth (Figura 21) é um ambiente de desenvolvimento Web para projetos generativos. A interface usa JavaScript, e possui capacidade de expansão para ser compatível com as PLs disponíveis em Rosetta (LOPES; LEITÃO, 2011).

⁵⁸ Do original em inglês: "The imperative style of programming is supported through blocks-based pro-gramming, where the user constructs procedures by creating sequences of code blocks (Resnick et al, 2014). The associative style of programming is supported through dataflow programming, where the user constructs networks consisting of nodes and wires. Each node has an imperative procedure that is defined using the blocks-based approach."

A interface oferece ambiente de programação “live programming”, conforme o usuário altera o código, é possível visualizar imediatamente as mudanças que isso ocasiona na geometria. Outra característica interessante do sistema é a Rastreabilidade, que destaca a geometria correspondente ao clicar na função ou expressão do código a qual se refere. O mesmo acontece se o usuário seleciona uma parte da geometria aparente, a Rastreabilidade resalta o trecho de código que descreve esta geometria selecionada (Figura 22).

Figura 21: Interface de programação de Luna Moth. Fonte: Alfaiate (2017, p. 24)

Figura 22: Exemplo da característica de Rastreabilidade de Luna Moth. Fonte: Alfaiate (2017, p. 29)

2.7 Considerações Parciais

O uso das linguagens de programação visual tem sido fundamental para impulsionar o interesse e a prática em Design Computacional. No entanto, mesmo para profissionais experientes, como nos casos relatados por Rick Smith apud Davis (2013), as VPLs revelam-se inflexíveis e com problemas de escalabilidade, o que acaba por restringir o desenvolvimento do projeto. Além das limitações por si só, a falta de controle e clareza do código podem ser ocasionadas, resultando muitas vezes em um projeto impreciso, de processo nebuloso ou com formas surgidas sob alguma lógica computacional que não é totalmente clara.

Além de ser base essencial para a comparação entre VPL e TPL, o conjunto de informações apresentadas neste capítulo também é importante para entender: (1) que o problema de escalabilidade se caracteriza como a questão-chave da limitação das VPLs (JANSSEN; LI; MOHANTY, 2016); (2) como os tipos de modelagem paramétrica possibilitadas pelas VPLs disponíveis em arquitetura influenciam no desenvolvimento do Projeto Orientado ao Desempenho.

Neste fluxo de trabalho a concepção do modelo deve estar atrelada às métricas desse desempenho; o sistema de modelagem precisa, por exemplo, suportar a geração iterativa (loops) de análises e simulações e utilizar algum método de otimização para estabelecer a performance do projeto.

Para superar as limitações encontradas nas VPLs o programador precisa, em algum momento do fluxo de trabalho computacional, programar em uma linguagem textual para obter escalabilidade. Isso coloca as VPLs no papel de suportar linguagens de programação textuais, em vez de serem linguagens de programação escaláveis (BURNETT et al., 1995). Para isso, Aish (2013) considera que é necessário nos afastarmos do foco de tornar as ferramentas de projeto computacional mais acessíveis e considerar apoiar uma progressão no desenvolvimento das habilidades computacionais dos arquitetos.

Para apoiar a discussão sobre a progressão necessária na habilidade que arquitetos devem ter em programar textualmente, discutiremos alguns aspectos sobre cognição em programação. Isso nos ajudará a investigar em qual direção arquitetos podem aplicar seus esforços e focar em práticas mais relevantes para a programação que necessitam desenvolver.

03

**Avaliação de Linguagens de Programação Visual
e Textual**

Avaliar ambientes e linguagens de programação (*Programming Languages*, PLs) é um desafio mesmo para a área de Interação Humano-Computador (*Human-Computer Interaction*, HCI) (GREEN, PETRE, 1996; BLACKWELL, 2006).

Neste capítulo cruzaremos alguns dos métodos de avaliação de PLs encontrados na literatura, que abordam tipos diferentes de avaliação: fatores de qualidade internos e externos (MEYER, 1997). Fatores de qualidade internos dizem respeito às características mais relevantes para a escrita da programação ou acesso aos pormenores do funcionamento do software. Por exemplo, se o código pode ser modular, legível ou se a entrada de um tipo de dado gera erros. Entendemos que os detalhes internos estão mais próximos de engenheiros de software, cientistas da programação e desenvolvedores de PLs. Fatores de qualidade externos dizem respeito às características que podem ser identificadas pelos usuários finais, como velocidade, facilidade de uso, adaptabilidade para mudanças no código. Consideramos “usuários” não só pessoas que interagem com o programa final (como no caso de usuários de software), mas como usuários especialistas em programação de domínios específicos, como no caso de arquitetos, engenheiros e cientistas que possuem experiência de programação limitada (BAROTH; HARTSOUGH, 1995; WHITLEY, 1997; MEYER, 1997).

É comum encontrar técnicas de avaliação de linguagens que se concentram em inspecionar tarefas simples da interação do usuário com a notação. Green e Petre (1996) apontam que muitas das tradições em avaliar PLs concentram-se no produto acabado, ou seja, avaliar detalhes físicos e de baixo nível⁵⁹ de interação entre o usuário e o sistema (suas qualidades internas). O problema em avaliar apenas através desta abordagem está na sobrecarga de detalhes sobre o funcionamento das linguagens e como cada tarefa poderia ser executada naquele ambiente.

Os autores propuseram uma abordagem alternativa chamada de Dimensões Cognitivas, como uma estrutura ampla para essa avaliação: “essa estrutura abrangente complementa as análises detalhadas e altamente específicas típicas dos modelos cognitivos contemporâneos em IHC e tem mais a dizer aos usuários que não são especialistas nessa área” (GREEN; PETRE, 1996, p. 3, tradução nossa⁶⁰).

⁵⁹ Em computação linguagens de baixo nível possuem instruções mais próximas da interpretação da máquina, mais próximas à arquitetura do computador. Já linguagens de alto nível possuem instruções com mais abstração, mais próximas da linguagem entendida por humanos.

⁶⁰ Do original em inglês: “This broad-brush framework supplements the detailed and highly specific analyses typical of

Desta maneira, para criar um background para avaliar VPLs e TPLs, consideramos como base as Dimensões Cognitivas. Realizamos o cruzamento de outras características de avaliação de PLs complementares, que consideram também fatores de qualidade internos, por serem características indispensáveis para o programador que deseja escalar entre VPLs e TPLs:

No final, apenas fatores externos são importantes [...] Mas a chave para alcançar esses fatores externos está nos fatores internos: para que os usuários aproveitem as qualidades visíveis, os projetistas e implementadores devem ter aplicado técnicas internas que garantam as qualidades ocultas. (MEYER, 1997, p.3, tradução nossa⁶¹)

As análises realizadas nesta pesquisa possuem generalizações. O ponto, no entanto, é que embora arquitetos compartilhem desafios das áreas da computação, nem todas as características da programação são relevantes para as circunstâncias do Design Computacional. Na seção a seguir, descrevo o corpo de conhecimento utilizado para realizar a comparação entre VPL e TPL e as possíveis características que são mais pertinentes para a prática da programação para arquitetura.

3.1 Critérios de Avaliação de PLs

Para podermos estabelecer uma base de comparação entre linguagens de programação diferentes, foi necessário recorrer à critérios de avaliação de linguagens disponíveis na literatura.

É possível constituir uma visão geral entre VPLs e TPLs. Embora seja plausível estabelecer suas vantagens e desvantagens, como fizemos no Capítulo anterior, parece mais interessante compará-las a fim de observar o impacto do uso dessas linguagens no processo de Projeto Orientado ao Desempenho. Isso se dá principalmente por ser reconhecido que não há uma conclusão sobre a melhor opção entre ambas (MENZIES, 2002 apud JANSSEN; LI; MOHANTY, 2016, p. 157).

Os critérios de avaliação de linguagens de programação utilizados nesta pesquisa partem de um contexto geral da cognição em programação e se direcionam ao domínio específico da programação em arquitetura.

Usamos como base as Dimensões Cognitivas (*Cognitive Dimensions*, CDs) (GREEN; PETRE, 1996) (Figura 23), uma estrutura utilizada como uma teoria geral para avaliar notações e linguagens de programação. As CDs discutem aspectos das PLs relacionados à Psicologia da Programação e

contemporary cognitive models in HCI and it has more to say to users who are not HCI specialists."

⁶¹ Do original em inglês: "In the end, only external factors matter.[...] But the key to achieving these external factors is in the internal ones: for the users to enjoy the visible qualities, the designers and implementers must have applied internal techniques that will ensure the hidden qualities."

Interação Humano Computador.

Green, Petre (1996)	Dimensões Cognitivas
1 Viscosidade	Resistência à mudança
2 Abstração	Tipos e disponibilidade de mecanismos de abstração
3 Comprometimento Prematuro	Restrições na ordem de fazer as coisas
4 Proximidade de Mapeamento	Proximidade entre representação e domínio
5 Dependências Ocultas	Links importantes entre entidades não são visíveis
6 Expressividade	A finalidade de uma entidade é prontamente inferida
7 Notação Secundária	Informações extras em outros meios que não a sintaxe formal
8 Visibilidade	Capacidade de visualizar facilmente os componentes
9 Operações Mentais Difíceis	Alta demanda em recursos cognitivos
10 Consistência	Semânticas similares são expressas em formas sintáticas similares
11 Difusão/Concisão	Verbosidade da linguagem
12 Propensão a Erros	A notação leva a erros e o sistema dá pouca proteção
13 Avaliação Progressiva	O trabalho pode ser verificado a qualquer momento

Figura 23: Dimensões Cognitivas definidas por Green e Petre (1996).

Considerando esse conjunto, utilizamos critérios de avaliação de linguagens de programação de outros autores e estabelecemos relações entre eles. Os critérios utilizados foram propostos pelos autores a seguir.

Sebesta (2011) propõe critérios para julgar TPLs de uso geral na computação. O autor oferece uma estrutura para avaliar as características que afetam TPLs nos quesitos legibilidade, facilidade de escrita e confiabilidade geral da linguagem. Os critérios definidos por Sebesta e a descrição resumida do que eles representam são sintetizados na Figura 24.

	Sebesta (2011)	Crítérios de avaliação para linguagens de programação
Legibilidade Facilidade de Escrita Confiabilidade	Simplicidade	A simplicidade geral da linguagem, como a quantidade de maneiras de realizar operações, de construções básicas, de significado dos operadores, etc.
	Ortogonalidade	A combinação entre uma quantidade pequena de conjuntos de construções primitivas e um número relativamente pequeno de formas para construir as estruturas de controle e de dados da linguagem.
	Tipos de Dados	A presença de mecanismos adequados para definir tipos e estruturas de dados.
	Projeto da Sintaxe	Sintaxe ou forma dos elementos da linguagem, como formato dos identificadores e palavras especiais.
	Suporte à Abstração	A habilidade em definir e usar estruturas ou operações complicadas de maneira que permita que muitos dos detalhes sejam ignorados.
	Expressividade	Uma linguagem expressiva especifica computações de uma forma conveniente, em vez de deselegante.
	Verificação de Tipos	A execução de testes para detectar erros de tipos em um programa, tanto por parte do compilador quanto durante a execução de um programa.
	Tratamento de Exceções	A habilidade de um programa de interceptar erros em tempo de execução.
	Apelidos	Apelidos são permitidos quando é possível ter um ou mais nomes para acessar a mesma célula de memória. São geralmente usados para resolver deficiências nos recursos de abstração de dados.

Figura 24: Critérios de avaliação de TPLs definido por Sebesta (2011)

Meyer (1997) propõe critérios para avaliar e orientar a qualidade de produtos desenvolvidos pela Engenharia de Software. A área possui contribuições importantes para o desenvolvimento de programas voltados à solução de problemas, muitas dessas contribuições podem ser emprestadas à arquitetura (DAVIS, 2013). Os critérios de avaliação definidos pelo autor são (Figura 25).

Figura 25: Critérios de avaliação para desenvolvimento de software proposto por Meyer (1997)

Meyer (1997)	Critérios de avaliação para desenvolvimento de software (POO)
1 Precisão	A capacidade dos produtos de software em executar suas tarefas exatas, conforme definido pela sua especificação.
2 Robustez	A capacidade dos sistemas de software de reagirem apropriadamente a condições anormais.
3 Extensibilidade	A facilidade de adaptar produtos de software a mudanças de especificação.
4 Reutilização	A capacidade dos elementos de software de servir para a construção de muitas aplicações diferentes.
5 Compatibilidade	A facilidade de combinar elementos de software com outros.
6 Eficiência	A capacidade que um sistema de software tem em colocar o menor número possível de demandas em recursos de hardware, como tempo do processador, espaço ocupado em memórias internas e externas, largura de banda usada em dispositivos de software.
7 Portabilidade	A facilidade de transferir produtos de software para vários ambientes de hardware e software.
8 Facilidade de Uso	A facilidade com que pessoas de várias origens e qualificações podem aprender a usar produtos de software e aplicá-los para resolver problemas. Ele também cobre a facilidade de instalação, operação e monitoramento.
9 Funcionalidade	A extensão de possibilidades oferecidas por um sistema.
10 Pontualidade	A capacidade de um sistema de software ser lançado quando ou antes seus usuários desejarem.

Kiper, Howard e Ames (1997) fornecem critérios para avaliar a qualidade de uma VPL (Figura 26). Os autores definem uma visão geral para considerar uma VPL “completa”. Esse conjunto de critérios servem para comparar e contrastar linguagens visuais ao fornecer “maior compreensão dos pontos fortes e fracos de cada um de um modo relativo ou absoluto” (KIPER; HOWARD; AMES, 1997, p. 180, tradução nossa⁶²).

Figura 26: Critérios de avaliação de VPLs definido por Kiper, Howard e Ames (1997)

Kiper, Howard e Ames (1997)	Critérios de avaliação para classificação de VPLs de uso geral
1 Natureza Visual	O grau em que a informação é representada em uma forma gráfica, como ícones, diagramas, gráficos, etc.
2 Funcionalidade	A aplicabilidade geral da linguagem e não sua restrição a um campo específico de aplicação.
3 Facilidade de Compreensão	A facilidade com que os programas nesta linguagem podem ser entendidos.
4 Suporte à Paradigmas	O grau em que a linguagem suporta o paradigma de programação pretendido.
5 Escalabilidade	Uma medida da capacidade de criar programas grandes.

Por fim, Aish e Hanna (2017) avaliam propriedades de linguagens visuais usadas especificamente em sistema de design paramétrico na arquitetura (Figura 27). Os critérios Barreiras de Abstração e Flexibilidade foram baseados diretamente em algumas das Dimensões Cognitivas.

Figura 27: Métricas para avaliação de VPLs utilizadas no ensino de arquitetura estabelecido por Aish e Hanna (2017)

Aish, Hanna (2017)	Métricas para avaliar as VPLs utilizadas no ensino de arquitetura
1 Barreiras de Abstração	As Barreiras de Abstração são determinadas pelo número mínimo de novas abstrações que devem ser dominadas antes de usar o sistema.
2 Interferência semântica	Ocorre quando há uma incompatibilidade entre um termo e o que significa ser transmitido.
3 Consistência entre Representações	Sistemas de modelagem paramétrica oferecem representações como representação gráfica de fluxo de dados baseada em gráfico, representação geométrica e às vezes uma representação do programa baseado em texto. Consistência entre essas representações ajuda o usuário a construir um modelo mental interno unificado.
4 Descoberta	Como a funcionalidade do sistema é apresentada e documentada para que possa ser descoberta pelo usuário sem ajuda.
5 Flexibilidade	Os autores utilizam três Dimensões Cognitivas de Green e Blackwell (1998) para compor o que entendem por Flexibilidade: (1) o sistema exige que o usuário pense e execute ações em um ordem inadequada? [Compromisso Prematuro]; (2) o sistema permite ao usuário tomar decisões tentativas que podem ser posteriormente alteradas? [Provisionalidade]; e (3) o quão difícil é que essa mudança aconteça posteriormente? [Viscosidade].
6 Efeitos Colaterais	Algumas mudanças mínimas do usuário (por exemplo, para os dados de entrada) têm um efeito abrangente e inesperado no comportamento do modelo ou programa (por exemplo, na saída).
7 Retrabalho	Modificações ou operações adicionais que o usuário é obrigado a fazer que não faz parte da intenção do projeto e nem parece ser logicamente necessário.
8 Fluxo de Trabalho Complicado	Para completar a tarefa, o usuário deve adotar um fluxo de trabalho tão complexo que todo o processo parece ser contraproducente e desencorajar ao ponto em que a tarefa pode ser abandonada.
9 Interatividade	O desempenho e o suporte do sistema para a interatividade.

⁶² Do original em inglês: “[...] giving a greater understanding of the strengths and shortcomings of each in either a relative or an absolute manner.”

Embora tratem-se de classificações distintas, identificamos que muitos dos aspectos encontrados nos critérios de avaliação dos autores possuem relações entre si. Cada autor utiliza uma nomenclatura diferente para discutir características semelhantes. Identificamos quais destes aspectos estão relacionados e os apresentamos na seção seguinte.

3.2 Dimensões Cognitivas

Dimensões Cognitivas (*Cognitive Dimensions*, CDs) são características de linguagens de computador que mostram a maneira como a informação é estruturada, representada e compartilhada por notações⁶³ diferentes. Também mostra como essa informação interage com o conhecimento cognitivo humano (GREEN, 1989; GREEN; PETRE, 1996).

A estrutura das CDs foi proposta inicialmente por Green (1989), ampliada por Green e Petre (1996)⁶⁴ e posteriormente atualizada por Blackwell et al. (2001), que analisa uma série de sugestões para ampliação das CDs sugeridas por outros autores, e providencia guidelines para que novas dimensões possam ser propostas.

A abordagem foi criada para auxiliar projetistas de sistemas de notação e informação a descrever, analisar, prever e avaliar o impacto cognitivo que esses sistemas terão em usuários humanos (BLACKWELL et al., 2001).

Sendo assim, as CDs são um método de avaliação de linguagens de programação, pois fornecem um background para comparar interfaces ou linguagens diferentes⁶⁵:

A análise de dimensão facilita a comparação de interfaces ou linguagens diferentes, e também ajuda a identificar a relação entre ferramentas de suporte e linguagens de programação: as ferramentas de suporte possibilitam o uso de planejamento oportunista com notações que de outra forma o inibiriam. (GREEN, 1989, p. 1, tradução nossa⁶⁶)

CDs foram amplamente aceitas e utilizadas por programadores e pesquisadores das áreas de Psicologia da Programação (*Psychology of Programming*, PoP) e Interação Humano-Computador (*Human-Computer Interaction*, HCI), pois foram capazes de fornecer um vocabulário para nomear e declarar os problemas surgidos durante o projeto de interação de usuários com esses sistemas, uma vez que muitas dessas dimensões só eram conhecidas pela experiência do projetista/usuário de maneira intuitiva. (BLACKWELL et al., 2001).

É importante notar que as CDs precisam ser consideradas no contexto do sistema em utilização pelo usuário: “o perfil de usabilidade de um sistema ou artefato depende do tipo de atividade em que o usuário estará envolvido e da estrutura das informações contidas na notação” (BLACKWELL et al., 2001). Isso se dá pelo fato de que nenhuma das dimensões é necessariamente boa ou ruim

⁶³ “Estamos mais acostumados a falar de linguagens de programação do que notações. Usarei o termo notação para manter a forma distinta do conteúdo”. (Green, 1989)

⁶⁴ Embora os autores reconheçam na altura que as CDs não é uma entidade acabada.

⁶⁵ Nesta dissertação apresentamos as dimensões cognitivas como um método de avaliação de linguagens de programação visual e seus ambientes, sejam elas de alto ou baixo nível, processual ou declarativa, de propósito especial ou de propósito geral (Green, 1989; Green and Petre, 1996)

⁶⁶ Do original em inglês: “The dimension analysis makes it easier to compare dissimilar interfaces or languages, and also helps to identify the relationship between support tools and programming languages: the support tools make it possible to use opportunistic planning with notations that would otherwise inhibit it.”

por si mesma. É preciso avaliar o contexto e as necessidades do usuário para que as decisões possam compensar perdas e ganhos (*trade-offs*) entre as dimensões:

Talvez o projeto da linguagem possa ser melhorado em um aspecto, mas apenas à custa de torná-lo pior em algum outro aspecto. Ou talvez ele possa ser mais apropriado para um determinado grupo de usuários, mas apenas às custas de se tornar menos utilizável para algum outro grupo de usuários. Ou, de maneira mais insidiosa, talvez o projeto da linguagem possa ser alterado de modo que seja adequado para os usuários quando eles estiverem realizando uma determinada tarefa, mas depois se tornem inutilizáveis para outra tarefa importante. (BLACKWELL et al. 2001, p. 327, tradução nossa⁶⁷)

Algumas pesquisas da área do Design Computacional utilizaram anteriormente parte da estrutura das CDs.

Celani e Vaz (2012) se referem às Dimensões Cognitivas de Green e Petre (1996) de maneira geral para discutirem alguns dos benefícios das VPLs em relação às TPLs.

Daniel Davis (2013) usa alguns aspectos encontrados nas CDs para discutir a flexibilidade de VPLs usadas em sistemas de design paramétricos ao longo de sua tese. O autor também se refere a alguns dos fatores estabelecidos por Meyer (1997) para orientar e avaliar a Programação Orientada a Objetos e os utiliza como estrutura para avaliar modelos paramétricos qualitativamente.

Aish e Hanna (2017) partem das CDs, principalmente das dimensões Barreira de abstração e Viscosidade⁶⁸ para comparar VPLs usadas em sistemas de design paramétrico durante o ensino de arquitetura. Os autores propõem dimensões adicionais que combinam e ampliam a pesquisa das CDs, as quais também serão consideradas e discutidas neste capítulo.

A seguir, discutiremos as Dimensões Cognitivas de Green e Petre (1996). Nos referiremos às dimensões como substantivos próprios, grafados com a primeira letra maiúscula, a fim de diferenciá-las de seu significado quando usamos as mesmas palavras como substantivo comum (por exemplo Abstração, enquanto referência à CD, e abstração, quando nos referirmos ao conceito geral da palavra).

3.2.1 Viscosidade (*Viscosity*)

How much effort is required to perform a single change? (GREEN; PETRE, 1996)

Green e Petre (1996) aplicam o termo Viscosidade para indicar quanto trabalho o usuário deve exercer para efetuar uma alteração no código, ou seja, a dimensão indica a resistência que o sistema tem à mudança local. A facilidade dessa alteração depende em parte do tipo da mudança. No entanto, algumas linguagens e/ou ambientes demandam mais trabalho para que isso aconteça.

⁶⁷ Do original em inglês: *"Perhaps the design can be improved in one respect, but only at the expense of making it worse in some other respect. Or perhaps it can be made more appropriate for a particular user group (e.g. the elderly), but only at the expense of becoming less usable for some other user group (e.g. those who have very little time). Or more insidiously, perhaps the design can be altered so that it is suitable for users when they are carrying out a certain task, but then becomes unusable for another important task."*

⁶⁸ Qual chamam de "Flexibilidade" de Green e Blackwell (1998)

A viscosidade normalmente é encontrada em sistemas que contém muitas dependências entre suas partes, onde uma pequena alteração requer muitos ajustes subsequentes (GREEN, 1989). Esse fator é geralmente relatado em códigos VPL (GREEN; PETRE, 1996; AISH, 2013; DAVIS, 2013).

Um exemplo fornecido por Green (1989) nos ajuda a compreender essa dimensão, imaginemos:

um texto contendo números de seção, números de figura e referências cruzadas; inserir ou excluir uma seção pode significar que muitos desses números devem ser alterados. As soluções são: evitar seções numeradas (ou seja, alterar a notação) ou usar um formatador de documentos com conhecimento, como Scribe ou TEX (ou seja, fornecer suporte de ambiente) (GREEN, 1989, p. 6, tradução nossa⁶⁹).

Um ambiente que disponibiliza abstrações adequadas pode reduzir a Viscosidade (GREEN; PETRE, 1996; BLACKWELL et al., 2001).

Similaridades entre Viscosidade e os critérios de avaliação dos demais autores

A maioria dos autores analisados (exceto Sebesta (2011)) tratam de aspectos que estão diretamente ligados ao conceito de Viscosidade (Figura 29). Aish e Hanna (2017) definem três categorias que entendemos estar intrinsecamente ligadas à esta dimensão, são elas: Retrabalho, Fluxo de Trabalho Complicado e Flexibilidade. Retrabalho trata de operações que o usuário é obrigado a adicionar ao código que não fazem parte da intenção do projeto e não são logicamente necessárias. Contudo, precisam ser inseridas para contornar problemas no código, principalmente em resposta a algumas limitações no sistema original. Isso leva a uma camada extra de esforço mental que o usuário precisa adicionar ao código para lidar com problema de Viscosidade e limitações da linguagem. Fluxo de

Figura 28: A dimensão Viscosidade e suas relações com as demais métricas de avaliação de PLs.

⁶⁹ Do original em inglês: “[...] a text containing section numbers, figure numbers, and cross-references; either inserting or deleting a section may mean that many such numbers must be changed. Solutions are to avoid numbered sections (i.e., change the notation) or to use a knowledgeable document formatter, such as Scribe or TEX (i.e., provide environmental support)”

Trabalho Complicado descreve o caso de um fluxo de trabalho complexo o suficiente para tornar o processo contraproducente, ou seja, o usuário pode abandonar a tarefa por perceber que a linguagem ou ambiente não parece apropriada para a resolução de seu problema. Flexibilidade trata de problemas de dimensionamento e usabilidade em VPLs. Os autores relacionam Flexibilidade explicitamente à dimensão Viscosidade:

Um nó em um gráfico de fluxo de dados combina: o nome, o 'tipo' e o método de cálculo usado para criar seu valor. Enquanto o tipo e o método podem ser interdependentes, em uma linguagem de programação baseada em texto regular, o usuário pode mudar qualquer um desses aspectos de forma independente. Em um sistema baseado em nó, essas opções geralmente não estão disponíveis, forçando o usuário a criar um novo nó e, em seguida, mover as conexões do nó antigo para o novo nó e, finalmente, excluir o nó antigo. Um exemplo claro de 'viscosidade' (AISH; HANNA, 2017, p. 7, tradução nossa⁷⁰)

Adicionalmente, revisitaremos Davis (2013) por também se referir a Flexibilidade ao discutir as limitações das VPLs. Sua relação com Viscosidade de Green e Petre é direta⁷¹, uma vez que a inflexibilidade dos modelos leva à reconstrução das partes mais frágeis do modelo (ou do modelo todo, nos casos mais graves de viscosidade). Isso geralmente causa atrasos inesperados no projeto. No pior dos casos o projetista acomoda os erros encontrados durante a programação e “termina com um projeto que não foi criado em um modelo paramétrico, mas foi criado para acomodar as limitações desse modelo paramétrico”⁷² (DAVIS, 2013, p. 6, tradução nossa⁷³). Para ajudar a compor a resolução deste problema na programação de códigos VPL em arquitetura, o autor sugere se basear nos critérios de Programação Estruturada da Engenharia de Software.

Meyer (1997) aborda o mesmo problema, do ponto de vista da Engenharia de Software, ao discutir Extensibilidade, a facilidade de adaptar os programas a mudanças de especificação: “Para programas pequenos, a mudança geralmente não é uma questão difícil; mas à medida que o software cresce, fica cada vez mais difícil se adaptar” (MEYER, 1997, p. 6, tradução nossa⁷⁴).

O autor sugere dois princípios para melhorar a Extensibilidade: (1) Simplicidade do Projeto, dado que uma estrutura simples será mais fácil de se adaptar a mudanças do que uma complexa e (2) Descentralização, ao propor mais autonomia dos módulos, para que uma mudança simples afete apenas uma quantidade pequena de módulos em vez de afetá-los por reação em cadeia.

Meyer argumenta que o método de Programação Orientada a Objetos ajuda na resolução de muitos problemas de extensibilidade ao auxiliar projetistas a produzir sistemas cuja estrutura permanece descentralizada e simples, mesmo para sistemas grandes.

⁷⁰ Do original em inglês: “[...] the name, the ‘type’, and the calculation method used to create its value. While type and method may be interdependent, in a regular text based programming language the user is free to change any one of these aspects independently. In a node based system these options are often not available forcing the user to create a new node, then to move the connections from the old node to the new node and finally to delete the old node. A clear example of ‘viscosity’.”

⁷¹ Embora Davis (2013) não forneça diretamente métricas de avaliação de PLs em sua tese, aproveitamos alguns ganchos entre as CDs e relatos sobre aspectos das VPLs citado pelo autor para compor nossa discussão.

⁷² Ressalta-se aqui que esse tipo de problema no processo de projeto da arquitetura é que embasa a introdução desta dissertação (seção Impacto das Linguagens de Programação no Processo de Projeto em Arquitetura (Problema de pesquisa)).

⁷³ Do original em inglês: “[...] ends up with a design that was not so much created in a parametric model as it was created for the limitations of a parametric model.”

⁷⁴ Do original em inglês: “For small programs change is usually not a difficult issue; but as software grows bigger, it becomes harder and harder to adapt.”

Além disso, o autor afirma que “o problema da extensibilidade é de escala” (MEYER, 1997, p. 6, tradução nossa⁷⁵).

Kiper, Howard e Ames (1997) também utiliza a categoria Escalabilidade em suas métricas para avaliar VPLs:

Métodos e ferramentas que funcionam bem em artefatos de software de pequeno ou médio porte não necessariamente funcionam tão bem com os maiores. [...] O argumento mais convincente em apoio a uma escalabilidade VPL é uma prova pelo exemplo: um desenvolvimento de um programa grande de alta qualidade com um nível razoável de esforço (KIPER; HOWARD; AMES, 1997, p. 184-188, tradução nossa⁷⁶).

Apontaremos a seguir em quais seções do estudo de aplicação a dimensão Viscosidade foi encontrada e analisada.

Na seção 4.4.5 Escalabilidade do código e modelo, a implementação do estudo de aplicação em uma TPL utilizando Rosetta mostrou avanços em relação à Viscosidade, Escalabilidade e Extensibilidade em relação a VPL.

Na seção 4.4.1 Mudança na topologia do projeto, apresentamos uma análise do problemas de Dependências Ocultas (descrito em detalhes na seção 3.2.7 Dependências Ocultas (Hidden-Dependencies)) no código VPL, a Viscosidade do sistema em manter a coerência do fluxo de dados é tamanha, que não conseguimos implementar uma solução em tempo hábil.

Na seção 4.4.3 Comunicação com simuladores, encontramos muita Viscosidade para contornar problemas na ordem de execução do código em sincronia entre a VPL Grasshopper e a TPL Rosetta.

3.2.2 Abstração (*Abstraction Gradient*)

What are the minimum and maximum levels of abstraction? Can fragments be encapsulated? (GREEN; PETRE, 1996, p. 11)

Abstração em Ciências da Computação é a habilidade em definir aspectos gerais de um contexto qualquer e ignorar características menos importante para a generalidade deste contexto. É um dos pilares da Programação Orientada a Objetos (POO), onde, por exemplo, uma classe abstrata fornece o modelo inicial genérico para outras classes derivadas que compartilham das funcionalidades iniciais e adicionam comportamentos específicos.

Green e Petre (1996, p. 18, tradução nossa⁷⁷) definem abstração como “um agrupamento de elementos a serem tratados como uma entidade”.

⁷⁵ Do original em inglês: “The problem of extendibility is one of scale.”

⁷⁶ Do original em inglês: “Methods and tools that work well on small or medium-size software artifacts do not necessarily work as well on larger ones. [...] The most convincing argument in support of a VPLs scalability is a proof by example: a development of a high quality, large program with a reasonable level of effort.”

⁷⁷ Do original em inglês: “[...] is a grouping of elements to be treated as one entity [...]”

Essa dimensão discute tipos e disponibilidades de mecanismos de abstração em PLs (BLACKWELL et al., 2001).

Do ponto de vista educacional, aprender a pensar em termos abstratos trata-se de uma grande conquista (GREEN; PETRE, 1996). No entanto, PLs que permitem muitas abstrações são mais difíceis de aprender (BLACKWELL et al., 2001; SEBESTA, 2011) uma vez que levam mais tempo para serem configuradas, pois necessitam que as abstrações sejam definidas antes que os objetivos do possam ser declarados (GREEN; PETRE, 1996).

Similaridades entre Abstração e os critérios de avaliação dos demais autores

Figura 29: A dimensão Abstração e suas relações com as demais métricas de avaliação de PLs.

Sebesta (2011) discute a mesma característica no tópico Simplicidade. Uma linguagem com muitas construções básicas (muitos mecanismos de abstração) é mais difícil de aprender e pode forçar o programador a se concentrar apenas em um subconjunto de recursos e ignorar outros. Sendo assim, a simplicidade geral de uma linguagem também afeta sua legibilidade (SEBESTA, 2011).

A característica Suporte a Abstração de Sebesta (2011) divide o mesmo significado que a dimensão, a habilidade em definir estruturas complicadas e ignorar detalhes destas operações: “É uma simples questão de o domínio da solução do problema da linguagem ser mais próximo do domínio do problema” (SEBESTA, 2011, p. 33). O suporte para abstrações é um fator importante na facilidade de escrita de uma linguagem.

De maneira geral, esses conceitos estão de acordo com a categoria de Meyer (1997) Facilidade de Uso, a facilidade com que programadores de vários níveis de conhecimento podem utilizar o sistema para resolver problemas. Uma das chaves para facilitar o uso do sistema é sua simplicidade estrutural (MEYER, 1997).

Kiper, Howard e Ames (1997) também define a categoria Facilidade de Compreensão nas características de avaliação de linguagens (neste caso,

de VPLs). Quando a linguagem falha neste requisito, ela se torna apenas outra nova linguagem para criar programas e não ajuda a resolver os problemas de desenvolvimento em programação (software crisis⁷⁸).

Na mesma medida, Green e Petre (1996) apontam que utilizar abstrações bem escolhidas pode aumentar a compreensão de uma PL. Sebesta (2011) também aborda a mesma perspectiva na categoria Suporte para Abstração. O autor define que “o grau de abstração permitido por uma linguagem de programação e a naturalidade de sua expressão são importantes para sua facilidade de escrita” (SEBESTA, 2011, p. 33).

No caso de uma VPL, o usuário não precisará pensar em muitas das regras para programar, como “variáveis, declarações, sequenciamento, fluxo de controle, ponteiros, modificação de estado, *loops* de evento, herança ou escopo hierárquico” (BURNETT; AMBLER, 1994 p. 59 apud GREEN; PETRE, 1996, p. 19, tradução nossa⁷⁹).

O uso de abstrações em programação possui um retorno e gratificação menos imediato, porém se torna essencial para resolver os problemas de flexibilidade geralmente relatados na programação com VPLs em arquitetura⁸⁰.

O cruzamento das informações acima nos indica que é necessário levar em conta o balanceamento dos níveis de abstração da linguagem, assim como entendido por Aish e Hanna (2016), ao medir as Barreiras de Abstrações em VPLs utilizadas na arquitetura:

Idealmente, novas abstrações devem ser descobertas e aplicadas na ordem em que parecem ser relevantes para o interesse do usuário. É necessário que o delta na compreensão entre uma abstração conhecida e uma abstração desconhecida esteja dentro da capacidade de compreensão do usuário. Em um contexto educacional, o objetivo não é evitar abstrações, mas evitar que abstrações se tornem uma barreira. (AISH; HANNA, 2017, p. 4, tradução nossa⁸¹)

Apontaremos a seguir em quais seções do estudo de aplicação a dimensão Abstração foi encontrada e analisada.

Na seção 4.4.1 Mudança na topologia do projeto, os mecanismos de abstrações disponíveis na TPL, conhecidos por melhorar a Viscosidade, de fato mostraram características que avançaram nesta dimensão.

⁷⁸ “O termo expressava as dificuldades do desenvolvimento de software frente ao rápido crescimento da demanda por software, da complexidade dos problemas a serem resolvidos e da inexistência de técnicas estabelecidas para o desenvolvimento de sistemas que funcionassem adequadamente ou pudessem ser validados” (WIKIPEDIA)

⁷⁹ Do original em inglês: “The user does not have to think about variables, declarations, sequencing, control flow, pointers, state modification, event loops, inheritance trees or hierarchical scope rules in order to program.”

⁸⁰ Como os problemas relatados em Davis (2013) e Chaszar e Joyce (2016).

⁸¹ Do original em inglês: “Ideally new abstractions should be discovered and applied in the order in which they appear to be relevant to the user’s interest and where the delta in understanding between a known abstraction and an unknown abstraction is within the user’s ability to comprehend. In an educational context the aim is not to avoid abstractions but to avoid abstractions becoming a barrier.”

3.2.3 Comprometimento Prematuro (*Premature Commitment*)

Do programmers have to make decisions before they have the information they need? (GREEN; PETRE, 1996)

Esta dimensão descreve quando a natureza do sistema impõe que o usuário tome decisões iniciais antes que as metas e objetivos do programa estejam completamente definidos. Segundo Green e Petre (1996), o problema surge quando: (a) a notação contém muitas dependências internas; (b) o meio ou o ambiente de trabalho restringe a ordem de fazer as coisas, e (c) a ordem é inadequada. Além disso, o problema desta antecipação forçada ocorre quando: (i) existem dependências internas entre os componentes e (ii) existem restrições de ordem que restringem quais componentes podem ser criados primeiro.

Similaridades entre Comprometimento Prematuro e os critérios de avaliação dos demais autores

Não foram encontradas relações diretas entre esta dimensão e os autores analisados.

No entanto, existem relatos relevantes de características que correspondem à esta dimensão. Em sua tese sobre a flexibilidade de modelos paramétricos, Davis (2013) aborda dois aspectos do que entendemos como Comprometimento Prematuro: grau de Planejamento Inicial e Antecipação de Flexibilidade de um modelo.

Planejamento Inicial (*front-loading*) é um termo usado em engenharia de software que significa organizar ou planejar um projeto ou processo, de modo que uma grande parcela da atividade geral ocorra concentrada no início deste processo, ou seja, um planejamento imediato. Este é um processo comum no planejamento de projetos que precisam seguir etapas lógicas com programação. No entanto, é necessário primeiro conceituar o que será modelado, antes mesmo da lógica de construção do algoritmo (DAVIS, 2013).

Embora essa carga inicial seja necessária, Smith afirma que a dificuldade deste processo “[...] não é o investimento cognitivo inicial, mas sim o risco de que o projetista possa investir tempo na abordagem errada” (SMITH, 2007, p. 2 apud DAVIS, 2013, p. 39, tradução nossa).

Davis afirma que o planejamento exige que o projetista antecipe (até certo ponto) os parâmetros e a hierarquia de relação entre as funções do modelo, uma vez que a modelagem paramétrica requer uma rigidez lógica das funções, que devem ser explícitas. “Esse planejamento antecipado pode ser um desafio, particularmente em um processo tão notoriamente difícil de antecipar como o processo de projeto (DAVIS, 2013, p. 40, tradução nossa).

A flexibilidade de um modelo paramétrico diz respeito à maneira como ele pode acomodar mudanças.

O projetista deve estruturar o modelo considerando possíveis mudanças ou futuras solicitações que poderão ser feitas no projeto. Porém, se esta mudança não tiver sido planejada ou antecipada durante a construção do algoritmo, o projetista precisará acomodar esta alteração adicionando funções no modelo (DAVIS, 2013, p. 41). Por exemplo, na adição de uma nova ramificação no modelo paramétrico, o projetista deve se preocupar em extrair do modelo existente um dado ou parâmetro que não estava previsto inicialmente, o que pode levar a um algoritmo

mais confuso e com maior tempo de processamento. Isso é significativo em modelos complexos com muitos parâmetros e dados associados.

Antecipar a flexibilidade também pode resultar num paradoxo para o projetista, uma vez que é preciso balancear quais parâmetros e dados podem estar disponíveis no modelo. Davis (2013) alega que todos os parâmetros têm um custo computacional, pois exigem trabalho antecipado para implementar e ainda mais trabalho para mudá-los. Desta maneira, o Comprometimento Prematuro é uma dimensão de grande protagonismo em VPLs em arquitetura.

Green e Petre (1996) consideram que VPLs possuem menos comprometimento com a ordem de criação do código do que TPLs. Por serem representadas por conexões de nós e fios, VPLs permitem que as etapas de programação estejam mais compatíveis com a ordem conceitual que vêm à mente do programador.

No entanto, os autores definem quatro motivos que podem influenciar no comprometimento antecipado em VPLs: (1) compromisso com o layout, quando os nós iniciais e subsequentes foram mal colocados, promovendo viscosidade para mudança, mesmo que haja maneiras de ajustar o layout do código; (2) compromisso com conexões, quando não há antecipação ou planejamento para a ordem e organização de conexões dos fios, o código facilmente torna-se bagunçado e confuso, devido à alta viscosidade de VPLs, pode-se levar muito tempo para limpar o código; (3) compromisso com a ordem de criação, quando a interpretação de um programa depende da ordem em que as entidades são criadas; e (4) compromisso com a escolha do constructo, quando é necessário escolher uma construção sintática e mais tarde descobrir que é necessário alterá-la para outra que não compartilha dos mesmos atributos, por exemplo, ao criar um plano em malha (mesh) e mais a frente no desenvolvimento do código, necessitar alterar o plano para superfície (surface).

Apontaremos a seguir em quais seções do estudo de aplicação a dimensão Comprometimento Prematuro foi encontrada e analisada:

Na seção 4.4.1 Mudança na topologia do projeto, foi difícil avaliar o Comprometimento Prematuro, uma vez que nos baseamos em um fluxo de trabalho existente. Ou seja, ao implementar o estudo de aplicação em TPL, tínhamos um panorama geral do fluxo de trabalho e comunicação (em teoria). No entanto, verificamos que mesmo o Planejamento Inicial do fluxo de trabalho implementado no estudo de aplicação em TPL sendo alta, a Viscosidade da VPL era tanta que esse não foi um fator de impacto no desenvolvimento (e era pra ser, visto os relatos de Davis (2013)). Ou seja, nem mesmo o Comprometimento Prematuro ajudou a diminuir a Viscosidade do sistema VPL em utilização.

3.2.4 Proximidade de Mapeamento (Closeness of Mapping)

What 'programming games' need to be learned? (GREEN; PETRE, 1996)

A Proximidade de Mapeamento se dá pelo quão estreita é a relação entre a notação e a entidade que ela descreve (BLACKWELL et al. 2001), ou seja, o mapeamento entre o mundo do problema e o mundo do programa: "Quanto mais próximo o mundo da programação estiver do mundo do problema, mais fácil será a resolução de problemas" (GREEN; PETRE, 1996, p. 19,

tradução nossa⁸²).

Os códigos (adaptados de Green e Petre (1996)) e figuras abaixo ilustram como a diferença na sintaxe das linguagens aproxima ou distancia o mundo do problema e o mundo do programa. Todos esses fragmentos realizam a soma de uma lista de valores:

C:

```
int vector[3] = { 4,6,12 };
sum = 0
for (i = 0; i < N; i++)
{
    sum += x[i];
}
```

Python:

```
list = [4,6,12]
sum = 0
for i in list:
    sum = sum + i
```

Planilha:

=SUM(A1..A3)

Figura 30: Soma de uma lista na VPL Dynamo.

Figura 31: Soma de uma lista na VPL Grasshopper

Figura 32: Soma de uma lista usando scripting em Python na VPL Grasshopper.

O código em C ilustra a quantidade de definidores de escopo incomuns da linguagem. O trecho possui três tipos de parênteses ({}, [] e ()) e três tipos de operadores de a (=, += e <) com uma estrutura com regras de sintaxe complicadas, que podem ser difíceis de lembrar (por exemplo, quando usar o ponto-e-vírgula). É necessário informar o tipo de dado, nome, quantidade de elementos e, se houver, inicialização dos dados. Além disso, C introduz primitivas incomuns que não possuem relação com os objetivos da resolução de problema, solicitando “jogos de programação” para atingir o objetivo do programa, somar valores de uma lista (GREEN; PETRE, 1996).

O código em TPL Python exige menos diretivas de escopo como { } e

⁸² Do original em inglês: “The closer the programming world is to the problem world, the easier the problem-solving ought to be.”

emprega primitivas mais simples quando comparado ao código C.

O código para planilha (considerada uma VPL) possui a menor quantidade de componentes incomuns para ser entendido (GREEN; PETRE, 1996).

Os códigos em VPL Dynamo (Figura 31) e Grasshopper (Figura 32 e Figura 33) possuem apenas um componente que executa a função somatória, chamada de primitiva de objetivo único por Green e Petre. Além disso, no caso das VPLs específicas de domínio, uma vez que a metáfora (como *Math.Sum* ou *MassAddition*) se encaixa no domínio do usuário, a linguagem será mais transparente para executar a tarefa, principalmente quando existem bibliotecas de funções específicas para isso (como a função *MassAddition*). No entanto, quando a lógica do programa falha, por exemplo, em casos em que é necessário usar recursividade de um dado em uma VPL de fluxo de dados (como em Grasshopper), o usuário precisa usar bibliotecas e métodos adicionais para atingir seu objetivo: “esta é uma peça extra de aparato conceitual, que é parte do mundo do programa e não parte do mundo do domínio externo” (GREEN; PETRE, 1996).

Estes exemplos simples, colocados lado a lado nos mostram que as VPLs (Planilha, Dynamo e Grasshopper) possuem mais Proximidade de Mapeamento, uma vez que os métodos disponíveis estão muito próximos do mundo do problema. Essa é uma característica clara de que as linguagens específicas de domínio apoiadas por formalismo visual oferecem o melhor dos mundos para o programador (GREEN; PETRE, 1996). Porém, inversamente, quando a estrutura da linguagem não atende mais às necessidades do usuário (por se tratar de um domínio muito específico), a Proximidade de Mapeamento diminui substancialmente, forçando o programador a usar muitos “jogos de programação” para contornar os problemas. Isso geralmente dificulta a exploração efetiva do projeto e pode tornar o desenvolvimento do código contraproducente.

Aish e Hanna (2017) descrevem a característica Interferência Semântica (baseada no padrão “nomes claros”, proposto por Woodbury (2010)) de maneira muito próxima à Proximidade de Mapeamento no contexto específico do domínio do PDS.

Figura 33: A dimensão Proximidade de Mapeamento e suas relações com as demais métricas de avaliação de PLs.

Segundo os autores, Interferência Semântica ocorre quando há uma incompatibilidade entre o termo usado em um método computacional e o que seu significado transmite para o usuário, ou seja, quando a linguagem cria uma metáfora para facilitar seu uso, porém oculta ou distancia a funcionalidade real da operação.

Aish e Hanna (2017), providenciam um exemplo equivalente em Grasshopper:

Computação geral: eu quero transpor a matriz de números [terminologia precisa descrevendo uma operação abstrata aplicada a dados abstratos]

Computação específica de domínio [retenção de abstrações de computação]: eu quero transpor a matriz de feixes [terminologia precisa descrevendo uma operação abstrata aplicada ao termo específico do domínio: feixes]

Um exemplo de computação específica de domínio baseada em metáfora [onde os arrays são implementados como uma estrutura de dados da árvore e a operação em árvores usa metáforas vernáculas, como 'flip']: eu quero virar a árvore de feixes [a metáfora vernácula "flip tree" combinada com o termo específico do domínio "feixes" resulta em um idioma híbrido que não é uma expressão computacional válida nem entendida como válida no domínio do aplicativo.

Não só isso é confuso, mas também pode ocultar a funcionalidade da operação abstrata. (AISH; HANNA, 2017, p. 5, tradução nossa⁸³).

De uma maneira mais simples, o nó Mass.Addition usado em Grasshopper também distancia o usuário do padrão geralmente utilizado em computação para a operação somatória "Sum", encontrado em C, Python e Dynamo.

A dimensão Proximidade de Mapeamento também possui alguns paralelos com o conceito de Ortogonalidade estabelecido por Sebesta (2011). O autor define a Ortogonalidade como: "um conjunto relativamente pequeno de construções primitivas que pode ser combinado a um número relativamente pequeno de formas para construir as estruturas de controle e de dados da linguagem" (SEBESTA, 2011, p. 28).

Embora as características de avaliação descritas por Sebesta se refiram a especificidades diretas de linguagens de programação, relacionamos Ortogonalidade com Proximidade de Mapeamento no sentido de que ambos dividem o mesmo objetivo geral de combinação/mapeamento (map) entre duas instâncias: combinar primitivas e estruturas de controle e dados dentro de uma linguagem, no caso da Ortogonalidade, e combinar a notação com a entidade que ela descreve, no caso da Proximidade de Mapeamento (e por sua vez, Interferência Semântica de Aish e Hanna (2017)).

Quanto mais ortogonal é a linguagem, mais regularidade no projeto, o que a torna mais fácil de aprender, ler e entender. Quanto menos, exceções às regras são necessárias (por exemplo na

⁸³ Do original em inglês: "General computing: I want to transpose the array of numbers [precise terminology describing an abstract operation being applied to abstract data] Domain specific computing [retaining computing abstractions]: I want to transpose the array of beams [precise terminology describing an abstract operation applied to domain specific term: beams] An example of metaphor based domain specific computing [where arrays are implemented as a tree data structure and operation on trees use vernacular metaphors, such as 'flip']: I want to flip the tree of beams [the vernacular metaphor 'flip tree' combined with domain specific term 'beams' results in a hybrid language which is neither a valid computational expression nor understood as a valid in the application domain. Not only is this confusing but it may also mask the functionality of the abstract operation]"

língua inglesa, onde i é usado antes de e, exceto após c), ou seja, as primitivas não podem ser recombinadas de qualquer maneira (SEBESTA, 2011).

No entanto, segundo o autor, assim como a Proximidade de Mapeamento, muita Ortogonalidade também possui problemas. Caso a linguagem possua muitas primitivas, a quantidade de combinações possíveis entre elas leva a uma complexidade desnecessária.

O autor ainda discute quais paradigmas de programação estão mais adequados a esses parâmetros. Essa discussão também é útil para esta pesquisa, uma vez que o estudo de aplicação foi implementado utilizando o paradigma funcional:

Algumas pessoas acreditam que as linguagens funcionais oferecem uma boa combinação de simplicidade e ortogonalidade. Uma linguagem funcional, como LISP, é uma na qual as computações são feitas basicamente pela aplicação de funções para parâmetros informados. Em contraste a isso, em linguagens imperativas como C, C++ e Java, as computações são normalmente especificadas com variáveis e sentenças de atribuição. As linguagens funcionais oferecem a maior simplicidade de um modo geral, porque podem realizar tudo com uma construção, ao chamarmos uma função, a qual pode ser combinada com outras funções de maneiras simples. Essa elegância simples é a razão pela qual alguns pesquisadores de linguagens são atraídos pelas linguagens funcionais como principal alternativa às complexas linguagens não funcionais como C++. Outros fatores, como a eficiência, entretanto, têm evitado que as linguagens funcionais sejam mais utilizadas. (SEBESTA, 2011, p.30)

As características Facilidade de Uso (MEYER, 1997) e Facilidade de Compreensão (KIPER; HOWARD; AMES, 1997) descrevem a facilidade com que usuários podem aprender e utilizar a notação para a resolução de problemas. A Proximidade de Mapeamento, a relação de proximidade entre a notação e o que ela descreve, reflete diretamente na facilidade de compreensão dessas notações.

3.2.5 Dependências Ocultas (Hidden-Dependencies)

Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic? (GREEN; PETRE, 1996)

Dependências Ocultas acontecem quando dois componentes dependem um do outro, mas essa dependência não é totalmente visível (BLACKWELL et al., 2001). Green e Petre (1996) exemplificam esse relacionamento a partir das planilhas:

A fórmula em uma célula informa de quais outras células ela obtém seu valor, mas não informa quais outras células tiram seu valor dela. No pior dos casos, toda a folha deve ser examinada célula por célula, antes de se ter certeza de que é seguro mudar uma célula. As dependências ocultas aparentemente são responsáveis por uma frequência de erro muito alta nas planilhas. (GREEN; PETRE, 1996, p. 27, tradução nossa⁸⁴)

⁸⁴ Do original em inglês: “The formula in a cell tells which other cells it takes its value from, but does not tell which other cells take their value from it. In the worst case the whole sheet must be scrutinised cell by cell before one can be certain that it is safe to change a cell. Hidden dependencies are apparently responsible for a very high error frequency in spreadsheets.”

Figura 34: Exemplo de dependências ocultas entre dados de uma planilha. Fonte: adaptado de Green e Petre (1996).

Planilha:

	A	B	C	D	E	F
1	item	unit price	units	price	tax	total price
2	apples	£ 0.84	2	= B2*C2	0.29	= E2*D2

Grasshopper:

Figura 35: A declaração na VPL Grasshopper das mesmas relações existentes na planilha.

Python:

```
u_price = 0.84
unit = 2
tax = 0.29
price = u_price * unit
total = price + tax * price
```

Os exemplos acima foram adaptados de Green e Petre (1996) para ilustrar a dimensão.

Na planilha (Figura 34), a célula D2 registra a quais células ela se refere (B2 e C2), mas não informa para quais células ela fornece este dado (o resultado de B2*C2).

Na VPL Grasshopper (Figura 35) as dependências são tornadas visíveis pela representação do fluxo de dados (embora os autores apontem aumento na Viscosidade).

Na TPL Python, como as variáveis guardam os resultados simbolicamente em vez de visualmente, os autores os classificam como parcialmente ocultos.

Eles enfatizam que o material necessário para a operação é visível em todos esses casos, no entanto, são as dependências que estão ocultas.

Similaridades entre Dependências Ocultas e os critérios de avaliação dos demais autores

Sebesta (2011)

Simplicidade
Ortogonalidade
Tipos de Dados
Projeto da Sintaxe
Suporte à Abstração
Expressividade
Verificação de Tipos
Tratamento de Exceções
Apelidos

Meyer (1997)

Precisão
Robustez
Extensibilidade
Reutilização
Compatibilidade
Eficiência
Portabilidade
Facilidade de Uso
Funcionalidade
Pontualidade

Green, Petre (1996)
Dimensões Cognitivas

Viscosidade
Abstração
Comprometimento Prematuro
Proximidade de Mapeamento
Dependências Ocultas
Expressividade
Notação Secundária
Visibilidade
Operações Mentais Difíceis
Consistência
Difusão/Concisão
Propensão a Erros
Avaliação Progressiva

Kiper et. al (1997)

Natureza Visual
Funcionalidade
Facilidade de Compreensão
Suporte à Paradigmas
Escalabilidade

Aish, Hanna (2017)

Barreiras de Abstração
Interferência Semântica
Consistência entre Representações
Descoberta
Flexibilidade
Efeitos Colaterais
Retrabalho
Fluxo de Trabalho Complicado
Interatividade

A categoria Efeitos Colaterais de Aish e Hanna (2017, p. 7, tradução nossa⁸⁵), descreve quando “uma mudança do usuário tem efeito abrangente e inesperado no comportamento do modelo ou programa”. Ambos se relacionam pelo fato de que os dados encadeados em um código são inevitavelmente dependentes. No entanto, é justamente o fator oculto da dependência que desencadeia resultados indesejados no código, ou seus efeitos secundários.

O critério Reutilização de Meyer (1997) também possui relação com Dependências Ocultas, uma vez que elas podem inviabilizar que o mesmo código seja reutilizado para resolver problemas similares, caso existam dependências não visíveis que impeçam a abrangência do uso de outras variáveis.

Apontaremos a seguir em quais seções do estudo de aplicação a dimensão Dependências Ocultas foi encontrada e analisada:

Nas seções 4.4.1 Mudança na topologia do projeto e 4.4.2 Relação e ordenação entre geometria e listas, identificamos que mesmo em um modelo bem estruturado, existem muitas aberturas para acontecerem casos de Dependências Ocultas em uma VPL.

3.2.6 Expressividade (Role-Expressiveness)

Can the reader see how each component of a program relates to the whole?
(GREEN; PETRE, 1996)

A dimensão da Expressividade torna mais fácil descobrir por que o programador definiu a estrutura do código de uma maneira particular. Em notações com pouca expressividade, as entidades não possuem características que as diferem e pode ser difícil descobrir seus relacionamentos (BLACKWELL et al., 2001). Essa dimensão pode ser aumentada através do uso de identificadores significativos, por modularidade bem estruturada, por sinalização de agrupamentos funcionalmente relacionados e qualquer nível de descrição explícito do código (GREEN; PETRE, 1996).

Similaridades entre Expressividade e os critérios de avaliação dos demais autores

Figura 37: A dimensão Expressividade e suas relações com as demais métricas de avaliação de PLs.

Sebesta (2011) também estabelece Expressividade como uma característica de avaliação de linguagem. O autor a define como a característica em oferecer operadores mais poderosos que permitem computações em programas menores, declarados de maneira sucinta.

Um exemplo de mecanismos mais expressivos numa PL são, por exemplo, compreensão de listas (*list comprehension*), disponíveis em algumas linguagens, como Python e Haskell. Trata-se de uma construção sintática que segue a forma de uma notação de definição de conjunto matemático e possui uma estrutura capaz de simplificar uma declaração. Por exemplo, no código Python, o código abaixo aplica a potência 2 a todos os itens de uma lista que vai de 0 a 9.

```
for item in range(10):  
    lista.append(x**2)
```

O mesmo Código escrito em compreensão de listas é:

```
lista = [item**2 for item in range(10)]
```

Essas construções aumentam a facilidade de escrita. A expressividade da linguagem oferece uma forma conveniente de escrita, em vez de deselegante (SEBESTA, 2011)

Expressividade se diferencia da dimensão Notação Secundária (seção 3.2.9 Notação Secundária (*Secondary Notation*)) por estar diretamente relacionada à facilidade de codificar instruções proporcionada pela sintaxe da linguagem. A chamada Notação Secundária se relaciona mais com aspectos acessórios que facilitam a legibilidade e documentação dessas instruções

A categoria Facilidade de Compreensão definida por Kiper, Howard e Ames (1997) pode ser relacionada a esta dimensão no âmbito das VPLs, uma vez que trata de maneira geral sobre a compreensão que os usuários têm dos programas.

Na mesma medida, Natureza Visual (KIPER; HOWARD; AMES, 1997) considera características inerentes da própria linguagem visual como uso da cor, arranjo espacial e ícones para aumentar a expressividade da linguagem.

Apontaremos a seguir em quais seções do estudo de aplicação a dimensão Expressividade foi encontrada e analisada.

Nas seções 4.4.7 Centralização e controle de dados e 4.4.8 Organização e clareza do código, no modelo VPL, temos todas as relações entre a ordem de execução do fluxo de trabalho (interação entre as camadas de projeto) apresentada de maneira explícita na tela. O tempo de compreensão para entender a relação entre as partes é menor do que em TPL, onde é necessário desenhar diagramas para entender quais módulos usam o que, e em qual ordem.

3.2.7 Notação Secundária (*Secondary Notation*)

Can programmers use layout, colour, or other cues to convey extra meaning, above and beyond the 'official' semantics of the language? (GREEN; PETRE, 1996)

A Notação Secundária é qualquer mecanismo pelo qual informações extras podem ser indicadas no código, que não seja sua sintaxe formal. Muitas linguagens permitem comentários, recuos, convenção para nomenclaturas, agrupamento de declarações relacionadas ou qualquer outra técnica que transmita significado ao leitor humano (GREEN; PETRE, 1996).

Essa dimensão é importante para a criação de programas complexos que precisam ser entendidos por outras pessoas e pelo próprio programador em uma data posterior. As notações podem servir para qualquer tipo de documentação do código, como informações, dúvidas e intenções.

Os autores exemplificam a diferença do uso das Notações Secundárias em VPLs a partir de um relato de usuário que passa algumas horas apenas reorganizando nós e fios para tornar o código mais compreensível, sem nenhuma alteração em sua funcionalidade. Já em TPLs, geralmente os programadores resolvem espaços em branco e recuos na medida em que avançam na escrita do código, embora também relatem que podem gastar muito tempo apenas analisando um programa e inserindo comentários (GREEN; PETRE, 1996).

Muitas das categorias paralelas possuem relação com Notação Secundária por abrangerem mecanismos que facilitam a compreensão, uso e manutenção dos programas, como é o caso das categorias Facilidade de Uso de Meyer (1997) e Facilidade de Compreensão de Kiper, Howard e Ames (1997).

Figura 38: A dimensão Notação Secundária e suas relações com as demais métricas de avaliação de PLs.

Sebesta (2011) aponta a legibilidade, a facilidade com o qual programas podem ser lidos e escritos, como um dos critérios mais importantes para julgar uma linguagem de programação. As características que impactam na legibilidade de uma PL são Simplicidade, Ortogonalidade, Tipos de Dados e Projeto de Sintaxe. Consideraremos o impacto geral destas categorias no quesito Legibilidade. As Notações Secundárias podem ser relacionadas à legibilidade por se tratarem de uma medida importante para manter a manutenção e ciclo de vida dos programas. Além disso, a categoria de Sebesta também discorre sobre considerar a legibilidade no contexto do domínio do problema: “se um programa que descreve uma computação é escrito em uma linguagem que não foi projetada para tal uso, ele pode não ser natural e desnecessariamente complexo, tornando complicada sua leitura (quando em geral seria algo simples)” (SEBESTA, 2011, p. 27).

Essa análise da Notação Secundária dentro do domínio específico da programação foi bem abordada por Davis, Burry e Burry (2011), que mediu o grau de legibilidade que usuários tiveram ao utilizar códigos na VPL Grasshopper não estruturados e estruturados através de princípios da programação modular (Figura 40). Os autores estabeleceram regras de organização de título, entradas (inputs), saídas (outputs), grupos e textos de descrição, encontradas na programação modular, e atingiram um alto grau de melhora na legibilidade do código pelos participantes da pesquisa. Atingir essa estrutura só foi possível pela Notação Secundária disponível no ambiente utilizado Grasshopper.

Figura 39: Comparação entre a legibilidade de um código estruturado (cima) e não estruturado (baixo) na VPL Grasshopper. Fonte: Davis, Burry e Burry (2011)

Apontaremos a seguir em quais seções do estudo de aplicação a dimensão Notação Secundária foi encontrada e analisada:

Nas seções 4.4.7 Centralização e controle de dados e 4.4.8 Organização e clareza do código, mostram como VPLs possuem grande vantagem em Notação Secundária.

3.2.8 Visibilidade (*Visibility*)

Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to juxtapose any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it? (GREEN; PETRE, 1996)

A dimensão Visibilidade se interessa pela acessibilidade de todas as informações do código, ou seja, se a parte solicitada do código está prontamente visível ou se pode ser facilmente acessada ou identificada (GREEN; PETRE, 1996):

Visualizar componentes relacionados simultaneamente é essencial, de acordo com o nosso modelo de programação. A justificativa cognitiva é dupla. (i) Os programadores geralmente resolvem um problema refinando uma solução anterior para um problema relacionado, de modo que precisam de acesso simultâneo aos dois trechos de código. (ii) Os programadores entendem um programa não

apenas pela inspeção local, mas olhando para uma varredura de coisas para formar um todo, exigindo, portanto, a capacidade de visualizar simultaneamente um grande número de itens de programa que podem ser amplamente dispersos. (GREEN; PETRE, 1996, p. 37, tradução nossa⁸⁶)

Similaridades entre Visibilidade e os critérios de avaliação dos demais autores

Sebesta (2011)	Meyer (1997)	Green, Petre (1996) Dimensões Cognitivas	Kiper et. al (1997)	Aish, Hanna (2017)
Simplicidade	Precisão	Viscosidade		Barreiras de Abstração
Ortogonalidade	Robustez	Abstração		Interferência Semântica
Tipos de Dados	Extensibilidade	Comprometimento Prematuro		Consistência entre Representações
Projeto da Sintaxe	Reutilização	Proximidade de Mapeamento	Natureza Visual	Descoberta
Suporte à Abstração	Compatibilidade	Dependências Ocultas	Funcionalidade	Flexibilidade
Expressividade	Eficiência	Expressividade	Facilidade de Compreensão	Efeitos Colaterais
Verificação de Tipos	Portabilidade	Notação Secundária	Suporte à Paradigmas	Retrabalho
Tratamento de Exceções	Facilidade de Uso	Visibilidade	Escalabilidade	Fluxo de Trabalho Complicado
Apelidos	Funcionalidade	Operações Mentais Díficeis		Interatividade
	Pontualidade	Consistência		
		Difusão/Concisão		
		Propensão a Erros		
		Avaliação Progressiva		

Num sentido semelhante, porém mais específico, Aish e Hanna (2017) abordam Consistência entre Representações. A visibilidade entre as partes do código é essencial para que o usuário compreenda a correspondência entre as partes, por exemplo: em uma VPL, é necessário que trechos do código VPL, nós descritos em TPL e as geometrias resultantes destes códigos possam ser visualizados simultaneamente.

Figura 40: A dimensão Visibilidade e suas relações com as demais métricas de avaliação de PLs.

Apontaremos a seguir em quais seções do estudo de aplicação a dimensão Visibilidade foi encontrada e analisada:

Na seção 4.4.8 Organização e clareza do código, a VPL indicou ter grande vantagem na dimensão Visibilidade.

3.2.9 Operações Mentais Díficeis (Hard-Mental Operations)

Are there places where the user needs to resort to fingers or pencilled annotation to keep track of what's happening? (GREEN; PETRE, 1996)

Os autores definem duas propriedades para esta dimensão:

Primeiro, as operações mentais problemáticas devem estar no nível notacional, não apenas no nível semântico, porque nosso tópico de interesse

⁸⁶ Do original em inglês: "Viewing related components simultaneously is essential, according to our model of programming. The cognitive justification is twofold. (i) Programmers often solve one problem by refining an earlier solution to a related problem, so they need simultaneous access to both pieces of code. (ii) Programmers understand a program not just by local inspection but by looking at a sweep of things in order to form a whole, hence requiring the ability to view simultaneously quite a number of program items which may be widely dispersed."

é como projetar boas notações, em vez de quais significados são em si difíceis de expressar, qualquer que seja a notação. (GREEN; PETRE, 1996, p. 23, tradução nossa⁸⁷)

A segunda é a combinação de cláusulas que podem aumentar substancialmente a dificuldade de compreensão, como duas ou três condicionais encadeadas, múltiplos negativos ou booleanos complexos.

Os autores definem que duas questões podem auxiliar o teste das Operações Mentais Difíceis, se a resposta for sim para ambas, essa dimensão então pode influenciar negativamente o uso da linguagem: “(i) se eu encaixar duas ou três dessas construções juntas, isso fica incompreensível? (ii) existe uma maneira em alguma outra linguagem de torná-la compreensível? (Ou é apenas uma ideia realmente difícil de entender?)” (GREEN, PETRE, 1996, p. 24, tradução nossa⁸⁸).

Similaridades entre Operações Mentais Difíceis e os critérios de avaliação dos demais autores

Não foram encontradas relações diretas entre esta dimensão e os autores analisados.

No entanto, uma vez que esta dimensão é descrita principalmente através do viés notacional da linguagem, faremos um paralelo entre Operações Mentais Difíceis e a característica geralmente reconhecida de que modelos paramétricos descritos em VPL possuem complexidade visual, resultante da representação complexa do código, com quantidade numerosa de nós e fios de fluxo de dados (LABELLE; NEMBRINI; HUANG, 2009; DINO, 2012; DAVIS, 2013)

Davis (2013) aborda essa característica como Cegueira a Mudança (Change Blindness). Em modelos complexos, o projetista muitas vezes pode não perceber visualmente mudanças nos códigos. Davis argumenta que, mesmo em modelos em que a mudança está desobstruída na tela enquanto se olha em direção a ela e as procura ativamente, é possível não conseguir enxergá-la. Geralmente, as pessoas não são capazes de detectar mudanças em uma cena visual, ao menos que ela ocorra em uma posição específica da retina onde a cena já estava localizada (DAVIS, 2013, p. 44).

O principal impacto deste fato no projeto de arquitetura é citado por Smith (2007), uma vez que alguma parte do código ou parâmetro foi alterado por outro projetista e está em desacordo com o desenvolvimento atual do modelo, a mudança pode vir à tona numa fase mais tardia do projeto, quando o custo de mudança do modelo pode ser ainda mais caro ou acarretar problemas maiores.

Apontaremos a seguir em quais seções do estudo de aplicação a dimensão Operações Mentais Difíceis foi encontrada e analisada:

Na seção 4.4.8 Organização e clareza do código descrevemos como algumas características presentes em Viscosidade são positivas na VPL e afetam diretamente a usabilidade da TPL.

⁸⁷ Do original em inglês: “First, the problematic mental operations must lie at the notational level, not solely at the semantic level, because our topic of interest is how to design good notations, rather than in the question of which meanings are in themselves hard to express, whatever the notation.”

⁸⁸ Do original em inglês: “[...] (i) if I fit two or three of these constructs together, does it get incomprehensible? (ii) is there a way in some other language of making it comprehensible? (Or is it just a really hard idea to grasp?).”

3.2.10 Consistência (Consistency)

When some of the language has been learnt, how much of the rest can be inferred? (GREEN; PETRE, 1996)

A noção de Consistência ou harmonia na linguagem observa se semânticas similares são expressas em formas sintáticas similares (BLACKWELL et al., 2001).

Ao analisar algumas VPLs (como LabView e ProGraph), os autores não perceberam nenhum problema específico de Consistência, em parte, porque VPLs possuem uma sintaxe muito mais simples.

Similaridades entre Consistência e os critérios de avaliação dos demais autores

Sebesta (2011)	Meyer (1997)	Green, Petre (1996) Dimensões Cognitivas	Kiper et. al (1997)	Aish, Hanna (2017)
Simplicidade	Precisão	Viscosidade		Barreiras de Abstração
Ortogonalidade	Robustez	Abstração		Interferência Semântica
Tipos de Dados	Extensibilidade	Comprometimento Prematuro		Consistência entre Representações
Projeto da Sintaxe	Reutilização	Proximidade de Mapeamento		Descoberta
Suporte à Abstração	Compatibilidade	Dependências Ocultas	Natureza Visual	Flexibilidade
Expressividade	Eficiência	Expressividade	Funcionalidade	Efeitos Colaterais
Verificação de Tipos	Portabilidade	Notação Secundária	Facilidade de Compreensão	Retrabalho
Tratamento de Exceções	Facilidade de Uso	Visibilidade	Suporte à Paradigmas	Fluxo de Trabalho Complicado
Apelidos	Funcionalidade	Operações Mentais Díficeis	Escalabilidade	Interatividade
	Pontualidade	Consistência		
		Difusão/Concisão		
		Propensão a Erros		
		Avaliação Progressiva		

Duas características similares à esta dimensão foram encontradas em Aish e Hanna (2017), são elas Descoberta e Consistência entre Representações.

Os autores definem sucintamente a primeira delas, Descoberta:

[...] descreve como a funcionalidade do sistema é apresentada e documentada para que possa ser descoberta pelo usuário sem ajuda. [...] Sem qualquer lógica geral, o usuário novato é forçado a consumir recursos cognitivos adicionais para aprender diretamente as idiossincrasias da estrutura do menu. Isso representa um investimento por parte do usuário não em conhecimento geralmente transferível (da classificação lógica de geometria e outros conceitos paramétricos e computacionais), mas em especificidades de um determinado sistema de design paramétrico. Isso cria dois desincentivos para que o usuário se mova para um sistema de design paramétrico diferente: abandonando o investimento em um sistema e investindo em aprender outro sistema. (AISH; HANNA, 2017, p. 6, tradução nossa⁸⁹).

Figura 41: A dimensão Consistência e suas relações com as demais métricas de avaliação de PLs.

⁸⁹ Do original em inglês: "[...] describes how the functionality of the system is presented and documented so that it can be discovered by the user unaided. [...] without any overall logic, the novice

Embora descreva consistência em uma dimensão diferente da descrita por Green e Petre, Consistência entre Representações diz respeito às múltiplas representações que o usuário das VPLs em arquitetura (ou Sistemas de Design Paramétrico) utilizam. É necessário que as representações como o fluxo de dados, representação geométrica e a do programa baseado em texto estejam consistentes para que o usuário possa construir um modelo mental interno unificado sobre o programa (AISH; HANNA, 2017).

Em VPLs, Dynamo parece ter mais consistência que Grasshopper, porque possui metáforas mais próximas à computação geral, uma vez que os nós são baseados em métodos que são escaláveis para uma TPL. Um exemplo disso foi mencionado na dimensão Proximidade de Mapeamento. Esse fator também influencia na descoberta. Grasshopper possui muitos nós que o usuário não percebe linearmente como podem ser utilizados, sendo que, às vezes, tratam-se de nós com semântica específica.

3.2.11 Difusão/Concisão (Diffuseness/Terseness)

How many symbols or graphic entities are required to express a meaning?
(GREEN; PETRE, 1996)

Esta dimensão se refere à quantidade de espaço e de símbolos na área de exibição que as linguagens utilizam para alcançar resultados que outras notações alcançariam de forma mais compacta (GREEN; PETRE, 1996). A diferença entre Difusão/Concisão e dimensão Proximidade de Mapeamento é que a primeira possui relação com a organização e quantidade de informações e entidades da linguagem:

É difícil dissecar essa diferença de uma maneira totalmente satisfatória, porque as notações que têm um mapeamento muito próximo do domínio do problema exigirão menos lexemas para alcançar seus resultados e, portanto, parecerão concisas; enquanto nosso interesse está na difusão das notações, independentemente da proximidade do mapeamento. (GREEN; PETRE, 1996, p. 22, tradução nossa⁹⁰)

user is forced to consume additional cognitive resources to directly learn the idiosyncrasies of the menu structure. This represents an investment on the part of the user not in generally transferrable knowledge (of the logical classification of geometry and other parametric and computational concepts) but in the specifics of a particular parametric design system. This then creates two disincentives for the user to move to a different parametric design system: abandoning the investment in one system and investing in learning another system."

⁹⁰ Do original em inglês: "It is difficult to dissect this difference in a wholly satisfactory way, because notations which have a very close mapping to the problem domain will require fewer lexemes to achieve their results, and will therefore appear terse; whereas our interest is in the diffuseness of notations independently of closeness-of-mapping."

Similaridades entre Difusão/Concisão e os critérios de avaliação dos demais autores

Figura 42: A dimensão Difusão/Concisão e suas relações com as demais métricas de avaliação de PLs.

Ao estabelecer métricas para classificar VPLs, Kiper apresenta a dimensão “Natureza Visual”: “A linguagem desejada deve ser inerentemente visual em seus principais componentes. O termo visual neste contexto inclui gráficos, diagramas e ícones [...] O uso da cor e do arranjo espacial de formas efetivas fazem parte desse critério.” (KIPER; HOWARD; AMES, 1997, p. 181, tradução nossa⁹¹).

3.2.12 Propensão a Erros (Error-Proneness)

Does the design of the notation induce ‘careless mistakes’? (GREEN; PETRE, 1996)

Os autores se referem a erros causados quando alguma característica sintática da linguagem ajuda o erro a ocorrer ou dificulta que o usuário o encontre depois que ocorreu.

Por exemplo, sistemas de delimitadores⁹²: “parênteses em Lisp, *begin-end* em Pascal e uma grande variedade de símbolos emparelhados em C - em todos os casos, não é incomum que o emparelhamento dê errado” (GREEN; PETRE, 1996, p. 23, tradução nossa⁹³).

Green e Petre (1996, p. 23) indicam que erros causados pela sintaxe são menores em VPLs do que TPLs: “Delimitadores e separadores⁹⁴ não são

⁹¹ Do original em inglês: “*The desired language must be inherently visual in its major components. The term visual in this context includes graphs, diagrams and icons [...] The use of color and spatial arrangement in effective ways are part of this criterion.*”

⁹² Um delimitador é uma sequência de um ou mais caracteres usados para especificar o limite entre regiões separadas e independentes em texto simples ou outros fluxos de dados. Um exemplo de um delimitador é o caractere de vírgula, que atua como um delimitador de campo em uma sequência de valores separados por vírgulas. Delimitadores representam um dos vários meios para especificar limites em um fluxo de dados (fonte: wikipedia Padrão Federal 1037C - Telecomunicações: Glossário de Termos de Telecomunicações)

⁹³ Do original em inglês: “[...] *parentheses in Lisp, begin-end in Pascal, and a wide variety of paired symbols in C – in all cases, it is not uncommon for the pairing to go wrong.*”

⁹⁴ Sistemas de identificadores também podem ser uma fonte de falhas, como linguagens que solicitam que o usuário use identificadores muito longos, ou apenas nomes formados por letras ASCII, ao contrário de linguagens mais modernas que permitem quase todos os caracteres Unicode

necessários porque a estrutura sintática é diferente. As dependências do fluxo de dados são explicitadas pelas linhas. Assim, a frequência de escorregões devido a essas causas deve ser drasticamente reduzida”.

Similaridades entre Propensão a Erros e os critérios de avaliação dos demais autores

Figura 43: A dimensão Propensão a Erros e suas relações com as demais métricas de avaliação de PLs.

Podemos estabelecer um paralelo entre esta dimensão e a característica Projeto de Sintaxe de Sebesta (2011). O autor define três aspectos que afetam a sintaxe: (1) Formato dos identificadores: onde critica a limitação que algumas linguagens impõe a quantidade de caracteres dos identificadores. Muita limitação impede que o usuário use nomes conotativos para variáveis e piora a legibilidade; (2) Palavras Especiais: onde a aparência e legibilidade de um programa é influenciada pelas palavras reservadas pelo sistema (como *while*, *class* e *for*). Por exemplo: “essas linguagens sofrem porque os grupos de sentenças são sempre terminados da mesma forma, o que torna difícil determinar qual grupo está sendo finalizado quando um *end* ou um *}* aparece.” (SEBESTA, 2011, p. 31); (3) Forma e Significado: “Projetar sentenças de maneira que sua aparência ao menos indique parcialmente seu propósito é uma ajuda óbvia à legibilidade” (SEBESTA, 2011, p. 32).

Verificação de Tipos (SEBESTA, 2011) diz respeito aos testes para detectar erros de tipos, se o operando é compatível com o operador, durante a execução do código. A verificação de tipos pode acontecer tanto por parte do compilador (tempo de compilação) quanto durante a execução do programa (tempo de execução).

Tratamento de Exceções (SEBESTA, 2011) é a habilidade que o programa tem em interceptar erros em tempo de execução, tomar medidas corretivas e então continuar.

Além de serem mecanismos importantes para a confiabilidade do programa, eles ajudam a detecção de erros mais cedo. Quanto antes

(GREEN, PETRE, 1996).

erros forem detectados, menos caro é fazer todos os reparos necessários (SEBESTA, 2011).

Grasshopper detecta erros de tipos em tempo de execução (quando o dado é imediatamente conectado aos *inputs* do nó) (Figura 45), no entanto, isso não é suficiente para eliminar erros durante a execução do programa. Além disso, tais problemas são difíceis de serem diagnosticados. Isso acontece porque o problema de erro na VPL está além da verificação de tipo, e passa a estar totalmente relacionado com a dimensão Avaliação Progressiva.

Figura 44: Um exemplo de Verificação de Tipos na VPL Grasshopper.

Em um sentido semelhante, Meyer (1997) também estabelece a característica Robustez. Por se tratar de diretrizes para a Programação Orientada a Objetos, o autor direciona sua preocupação aos bugs e defeitos na programação de softwares. Aspectos como especificações mais formais, verificações internas durante todo o processo de construção do software e testes e depuração podem permitir que as ferramentas detectem inconsistências antes que elas conduzam a defeitos.

Tanto Meyer quanto Sebesta consideram essas características importantes para a confiabilidade da linguagem de programação e do programa.

Embora Green e Petre (1996) indiquem que VPLs possuem, no geral, menos Propensão a Erros devido a sua sintaxe com menos declarações de delimitadores e separados, não consideraremos no quadro abaixo a avaliação da VPL como positiva nesta dimensão. Para que a linguagem possa ser considerada confiável, suas especificações necessitam estar de acordo com todas as condições da dimensão.

Apontaremos a seguir em quais seções do estudo de aplicação a dimensão Propensão a Erros foi encontrada e analisada:

A seção 4.4.6 Confiabilidade de resultados se dedica a discutir como essa dimensão influencia no desenvolvimento do fluxo de trabalho usando VPL e TPL.

3.2.13 Avaliação Progressiva (Progressive Evaluation)

Can a partially-complete program be executed to obtain feedback on “How am I doing”? (GREEN; PETRE, 1996)

A descrição da dimensão Avaliação Progressiva é bastante direta. Trata-se da tarefa de avaliar o progresso do código na solução de problemas em intervalos frequentes. Para isso o ambiente de programação deve apoiar essa dimensão permitindo a avaliação progressiva em programas parcialmente concluídos (GREEN; PETRE, 1996).

Similaridades entre Propensão a Erros e os critérios de avaliação dos demais autores

Figura 45: A dimensão Avaliação Progressiva e suas relações com as demais métricas de avaliação de PLs.

Esta dimensão possui alinhamentos com Verificação de Tipos de Sebesta (2011), que trata de um mecanismo de verificação parcial do funcionamento do código.

Também há uma forte ligação entre esta dimensão e as características Precisão e Robustez de Meyer (1997). O autor considera Precisão a melhor qualidade de um sistema:

Se um sistema não faz o que é suposto fazer, todo o resto sobre ele - seja rápido, tem uma boa interface de usuário - pouco importa. Mas isso é mais fácil dizer do que fazer. Mesmo o primeiro passo para a correção já é difícil: devemos ser capazes de especificar os requisitos do sistema de forma precisa, por si só uma tarefa bastante desafiadora. (MEYER, 1997, p. 4, tradução nossa⁹⁵)

Meyer sugere que uma técnica realista para melhorar a Precisão em um sistema é tratar os problemas (ou programas, no caso de softwares) em camadas, iniciando a confiabilidade pelas camadas iniciais. Ao conseguir garantir que cada camada esteja correta, pode-se supor que os níveis inferiores (ou anteriores) estejam corretos: “esta é a única técnica realista, pois consegue separar as preocupações e nos permite concentrar em cada estágio em um conjunto limitado de problemas” (MEYER, 1997, p. 4, tradução nossa⁹⁶).

No entanto, no caso de VPLs como Grasshopper o problema da avaliação progressiva se torna ainda mais complexo, pois o programa também depende de outras bibliotecas de componentes reutilizáveis. Problemas de Precisão nestes *plugins*/bibliotecas interferem diretamente na

⁹⁵ Do original em inglês: “If a system does not do what it is supposed to do, everything else about it — whether it is fast, has a nice user interface... — matters little. But this is easier said than done. Even the first step to correctness is already difficult: we must be able to specify the system requirements in a precise form, by itself quite a challenging task.”

⁹⁶ Do original em inglês: “This is the only realistic technique, as it achieves separation of concerns and lets us concentrate at each stage on a limited set of problems.”

confiabilidade. Além disso, devido à natureza de fluxo de dados, códigos em VPL não retornam resultados até que cada nó e fio estejam conectados e suas sub-rotinas estejam presentes no código (GREEN; PETRE, 1996). Isso torna um desafio maior realizar testes parciais de funcionamento em códigos VPL, embora não sejam impossíveis.

Depurar e testar permanecem indispensáveis como um meio de checar duplamente o resultado do programa.

O conceito de Robustez (MEYER, 1997) complementa a Precisão, pois diz respeito ao comportamento de um sistema nos casos cobertos pela sua especificação. O papel desta característica é garantir que se um caso anormal surgir durante a execução, o sistema não cause eventos catastróficos, mas sim produza mensagens de erro apropriadas e encerre sua execução de forma limpa.

Apontaremos a seguir em quais seções do estudo de aplicação a dimensão Avaliação Progressiva foi encontrada e analisada:

A seção 4.4.6 Confiabilidade de resultados se dedica quase inteiramente e discutir como essa dimensão influencia no desenvolvimento do fluxo de trabalho orientado ao desempenho.

3.2.14 Outros paralelos

Encontramos paralelos relevantes entre características dos autores que não são abordadas pelas CDs.

Funcionalidade:

Meyer (1997) e Kiper, Howard e Ames (1996) discutem definições similares em Funcionalidade e Suporte à Paradigmas. Embora olhem para produtos finais diferentes - sistemas de softwares e sistemas de programação visual – as características dividem as mesmas preocupações enquanto diretrizes de uma PL.

Sebesta (2011)	Meyer (1997)	Green, Petre (1996)	Kiper et. al (1997)	Aish, Hanna (2017)
		Dimensões Cognitivas		
		Viscosidade		
		Abstração		
		Comprometimento Prematuro		
		Proximidade de Mapeamento		
		Dependências Ocultas		
		Expressividade	Natureza Visual	Barreiras de Abstração
Simplicidade	Precisão		Funcionalidade	Interferência Semântica
Ortogonalidade	Robustez			Consistência entre Representações
Tipos de Dados	Extensibilidade			Descoberta
Projeto da Sintaxe	Reutilização			Flexibilidade
Suporte à Abstração	Compatibilidade	Notação Secundária	Facilidade de Compreensão	Efeitos Colaterais
Expressividade	Eficiência	Visibilidade	Suporte à Paradigmas	Retrabalho
Verificação de Tipos	Portabilidade	Operações Mentais Díficeis	Escalabilidade	Fluxo de Trabalho Complicado
Tratamento de Exceções	Facilidade de Uso	Consistência		Interatividade
Apelidos	Funcionalidade	Difusão/Concisão		
	Pontualidade	Propensão a Erros		
		Avaliação Progressiva		

Meyer (1997) define Funcionalidade como a extensão das possibilidades oferecidas por um sistema. No entanto, o desafio enfrentado por pelo projetista do sistema consiste em saber quanta funcionalidade é suficiente para este sistema:

Figura 46: A relação entre Funcionalidade de Meyer (1997) e Kiper, Howard e Ames (1997).

Um bom produto de software é baseado em um pequeno número de ideias poderosas; mesmo que tenha muitos recursos especializados, todos eles devem ser explicados pelas consequências desses conceitos básicos. (MEYER, 1997, p. 12, tradução nossa⁹⁷)

Kiper, Howard e Ames (1996) estabelece que a Funcionalidade ajuda a avaliar se a VPL é equivalente em energia a outras PLs de propósito geral. Os autores consideram que uma VPL completa (uma linguagem visual de propósito geral) deveria “ser tão equivalente em poder a Máquina de Turing quanto as linguagens de programação C, Ada, Prolog e Lisp” (KIPER; HOWARD; AMES, 1996, p. 182, tradução nossa⁹⁸).

Embora Kiper, Howard e Ames (1996) estabeleçam Suporte à Paradigmas como um critério paralelo, sua relação com a Funcionalidade é direta, uma vez que um paradigma de programação é um meio de se classificar PLs baseado em suas funcionalidades (um conjunto coerente de princípios voltados para o sistema em questão) (VAN ROY, 2009, p.10): “Cada paradigma suporta um conjunto de conceitos que torna o melhor para um determinado tipo de problema”.

Os autores apontam que a quantidade de paradigmas suportados aumenta a funcionalidade da VPL, que por sua vez deve ser idealmente personalizável para uma variedade de paradigmas:

[...] nosso critério aqui não pretende ser prescritivo ao julgar a validade ou importância de vários paradigmas. Fazer isso arriscaria discussões com proponentes de paradigmas que sustentam suas opiniões com um fervor quase religioso. Nosso ponto é que um VPL completo deve suportar seu paradigma subjacente, ou deveria ser personalizável para um conjunto de paradigmas (por exemplo, a linguagem visual Vista fornece suporte multiparadigmático). (KIPER; HOWARD; AMES, 1996, p. 184, tradução nossa⁹⁹)

Eficiência

Outra característica de fator relevante para a avaliação de uma PL são os critérios Eficiência (MEYER, 1997) e Interatividade (AISH; HANNA, 2017). Ambos se referem a capacidade de resposta em um sistema interativo:

Eficiência é a capacidade de um sistema de software colocar algumas demandas em recursos de hardware, como tempo do processador, espaço ocupado em memórias internas e externas, largura de banda usada em dispositivos de comunicação (MEYER, 1997, p. 9, tradução nossa¹⁰⁰).

⁹⁷ Do original em inglês: “A good software product is based on a small number of powerful ideas; even if it has many specialized features, they should all be explainable as consequences of these basic concepts.”

⁹⁸ Do original em inglês: “[...] should be as equivalent in power to Turing Machines as are the programming languages C, Ada, Prolog and Lisp.”

⁹⁹ Do original em inglês: “[...] our criterion here is not intended to be prescriptive in judging the validity or importance of various paradigms. To do that would risk arguments with proponents of paradigms who hold their views with an almost religious fervor. Our point is that a complete VPL should support its underlying paradigm, or should be customizable to a set of paradigms (for example, the visual language Vista provides multi-paradigm support).”

¹⁰⁰ Do original em inglês: “Efficiency is the ability of a software system to place as few demands as possible on hardware resources, such as processor time, space occupied in internal and external memories, bandwidth used in communication devices.”

Sebesta (2011)	Meyer (1997)	Green, Petre (1996) Dimensões Cognitivas	Kiper et. al (1997)	Aish, Hanna (2017)
Simplicidade	Precisão	Viscosidade		Barreiras de Abstração
Ortogonalidade	Robustez	Abstração		Interferência Semântica
Tipos de Dados	Extensibilidade	Comprometimento Prematuro		Consistência entre Representações
Projeto da Sintaxe	Reutilização	Proximidade de Mapeamento	Natureza Visual	Descoberta
Suporte à Abstração	Compatibilidade	Dependências Ocultas	Funcionalidade	Flexibilidade
Expressividade	Eficiência	Expressividade	Facilidade de Compreensão	Efeitos Colaterais
Verificação de Tipos	Portabilidade	Notação Secundária	Suporte à Paradigmas	Retrabalho
Tratamento de Exceções	Facilidade de Uso	Visibilidade	Escalabilidade	Fluxo de Trabalho Complicado
Apelidos	Funcionalidade	Operações Mentais Difíceis		Interatividade
	Pontualidade	Consistência		
		Difusão/Concisão		
		Propensão a Erros		
		Avaliação Progressiva		

Figura 47: A relação entre Eficiência de Meyer (1997) e Interatividade de Aish e Hanna (2017).

Para Meyer (1997, p. 9, tradução nossa¹⁰¹) eficiência é sinônimo de performance: “Em termos mais gerais, a preocupação com a eficiência deve ser equilibrada com outros objetivos, como a extensibilidade e a reusabilidade”.

Aish e Hanna (2017) consideram Interatividade (*Liveness*) como o desempenho e suporte do sistema para que o usuário obtenha uma impressão do produto final durante os estágios intermediários do projeto. Um exemplo desta interatividade é a latência do sistema, ou o período de tempo que o usuário aguarda para ver o resultado depois de uma alteração no código.

A Eficiência/Interatividade das linguagens são características relevantes no uso dos sistemas, ao mesmo tempo que são um desafio. Os modelos paramétricos usados na arquitetura, declarados através de VPLs são por sua própria natureza grandes conjuntos de dados atrelados à visualização de geometrias. Explorar esses modelos parametricamente pode ser inviável devido a latência em um sistema de fluxo de dados. Isso já é evidente durante a manipulação de um código de complexidade intermediária. Davis (2013, p. 76, tradução nossa¹⁰²) afirma que a latência baixa em um modelo paramétrico pode ser um desafio pelas “demandas computacionais de vários cálculos geométricos, as limitações do hardware do computador e os gargalos nos algoritmos subjacentes dos ambientes de modelagem paramétrica”.

¹⁰¹ Do original em inglês: “More generally, the concern for efficiency must be balanced with other goals such as extendibility and reusability.”

¹⁰² Do original em inglês: “[...] computational demands of various geometric calculations, the limitations of computer hardware, and the bottlenecks in the underlying algorithms of parametric modelling environments.”

Sebesta (2011)

Meyer (1997)

Green, Petre (1996)

Kiper, Howard, Ames (1997)

Aish, Hanna (2017)

Dimensões Cognitivas

- Desempenho positivo em TPLs
- Desempenho positivo em VPLs
- Desempenho neutro em VPLs e TPLs
- Desempenho negativo em VPLs e TPLs

3.3 Considerações Parciais

A Figura 48 apresenta as treze Dimensões Cognitivas com as relações encontradas nos critérios de avaliação dos autores.

É interessante notar que os aspectos que giram em torno das dimensões Viscosidade, Abstração, Proximidade de Mapeamento e Notação Secundária, são os que possuem relatos mais frequentes nas avaliações de linguagem dos outros autores (Figura 49).

Figura 48: Cruzamento e semelhanças encontradas entre CDs e as métricas de avaliação de linguagens dos autores citados.

Também chamamos a atenção para as dimensões Propensão a Erros e Avaliação Progressiva. Ambas estão mais atreladas às qualidades internas da programação, por se tratarem de características específicas, como Projeto de Sintaxe, Verificação de Tipos, Tratamento de Exceções, Precisão e Robustez, como descritas por Sebesta (2011) e Meyer (1997). No entanto, tratam-se de dimensões chaves para definir a confiabilidade do processo de projeto orientado a desempenho. Isso evidencia a importância da aproximação do arquiteto programador com questões mais próximas do desenvolvimento TPL.

Além disso, Meyer (1997, p. 15, tradução nossa¹⁰³) afirma que o critério Precisão (*Correctness*) se destaca do restante e o considera o mais importante: “se o software não executa sua função, o resto é inútil”. Este é o único critério que está acima das leis de compensação (*trade-offs*) entre as dimensões.

As relações de compensação entre as dimensões cognitivas é um consenso entre os autores (MEYER, 1997; GREEN; PETRE, 1996; BLACKWELL et al. 2001). Isso acontece porque muitos destes requisitos podem entrar em conflito: “embora em muitos casos seja possível encontrar uma solução que reconcilie fatores aparentemente conflitantes, às vezes você precisará

Figura 49: Dimensões Cognitivas e métricas de avaliação que possuem similaridades mais relacionadas entre autores.

¹⁰³ Do original em inglês: “If the software does not perform its function, the rest is useless.”

fazer concessões” (MEYER, 1997, p. 15, tradução nossa¹⁰⁴).

Meyer (1997) aponta que muitas vezes, desenvolvedores fazem trocas entre as características de um sistema de maneira implícita, ou seja, sem examinar as questões envolvidas e as várias opções disponíveis. No entanto, o autor atenta para a importância de uma análise que se esforça em indicar claramente os critérios e fazer as escolhas conscientemente. Green e Petre (1996) apontam que um vocabulário no nível das dimensões cognitivas oferece uma abordagem para encontrar a melhor compensação e realizar estas escolhas de maneira mais consciente.

É a partir da melhor compreensão entre as relações dos critérios de avaliação de PLs e do vocabulário das Dimensões Cognitivas feitas neste capítulo, que analisaremos, no capítulo seguinte, a implementação do Estudo de Aplicação. Avaliamos comparativamente o mesmo fluxo de trabalho implementado em duas linguagens específicas, a VPL Grasshopper e a TPL Racket/Rosetta. O objetivo é compor o quadro geral da comparação entre VPLs e TPLs com evidências empíricas mais específicas, encontradas no funcionamento interno destas linguagens.

¹⁰⁴ Do original em inglês: “Although it is in many cases possible to find a solution that reconciles apparently conflicting factors, you will sometimes need to make tradeoffs.”

104

Estudo de Aplicação

github.com/landimg/designspaceconstruction_TPL

Neste capítulo apresentaremos como a análise de evidências empíricas pode compor o quadro geral de comparação entre VPLs e TPLs.

Exploramos um estudo de aplicação real usando VPL e TPL. O algoritmo foi desenvolvido inicialmente pelo escritório Perkins+Will usando Grasshopper. O objetivo foi otimizar o projeto da fachada explorando o equilíbrio entre aspectos de eficiência energética e iluminância. Em seguida, o projeto foi reimplementado usando a TPL Racket e o ambiente de programação Rosetta (LOPES; LEITÃO, 2011).

À medida que procuramos solucionar a lacuna existente na comparação dos fluxos de trabalho de VPLs e TPLs, recorrendo a um projeto real, selecionamos este estudo de aplicação porque ele abrange todos os estágios do Projeto Orientado ao Desempenho - concepção, simulação e otimização - em vez de um projeto simplesmente paramétrico. Existem diferenças fundamentais encontradas entre os dois fluxos de trabalho em estratégias de escalabilidade e fluxo de dados.

Consequentemente, os resultados deste estudo visam orientar os futuros aprimoramentos nas linguagens e interfaces de programação para arquitetura, ao mesmo tempo em que tenta fortalecer a confiabilidade dos fluxos de trabalho em Design Computacional.

Necessidade do trabalho empírico na comparação de linguagens

É comum que projetistas de sistemas VPLs, em sua maioria engenheiros de software, sejam forçados a fazer escolhas de projeto sobre as linguagens, sem terem dados suficientes para direcionar suas decisões (WHITLEY, 1997). Isso acontece pela escassez de estudos empíricos no projeto das linguagens e interfaces VPLs. Pelo menos até o final da década de 1990, a maior parte da comunidade acadêmica da ciências da computação estava mais inclinada a descartar as VPLs como uma moda de pesquisa pouco promissora: “embora algumas alegações sejam hipóteses iniciais razoáveis, a falta de evidências de apoio deixa a porta aberta a ataques justos dos críticos” (WHITLEY, 1997, p. 110, tradução nossa¹⁰⁵).

Em sua pesquisa sobre o uso de interfaces de programação para usuários finais (*end-user programming*), Blackwell (2006) indica que as reflexões sobre as diretrizes que engenheiros de software usam para projetar essas interfaces devem ser guiadas por: (1) estudos empíricos; (2) pela participação dos próprios usuários do sistema e (3) pela psicologia da programação.

Olhando especificamente para o uso de VPLs no processo de projeto de arquitetura, ainda hoje Aish e Hanna (2017) apontam para a falta de estudos empíricos que investigam a forma como design thinking é suportado pelas ferramentas de Design Computacional disponíveis.

Desta maneira, essa pesquisa busca através da: (1) revisão de literatura sobre o uso de VPLs e TPLs em arquitetura; (2) embasamento teórico sobre os métodos de cognição para programação; e (3) a experiência dos arquitetos e programadores envolvidos na equipe e a implementação do estudo de aplicação em TPL, a levantar reflexões sobre as estratégias de fluxo de trabalho computacional em arquitetura e diretrizes para as interfaces de programação para arquitetos.

¹⁰⁵ Do original em inglês: “Even though some claims have reasonable initial hypotheses, the shortage of supporting evidence leaves the door wide open to justifiable attacks from critics.”

4.2 Estudo de Aplicação

Durante o desenvolvimento do projeto de arquitetura que usa programação, é útil estabelecer separações entre fases no fluxo de trabalho. Isso pode aumentar a confiabilidade do processo, uma vez que se tem mais controle de como o projeto foi executado ao observá-lo progressivamente ao final de cada fase.

Em sua tese, Davis (2013) aponta para paralelos entre as áreas de engenharia de software e arquitetura. O autor discute o conjunto de conhecimentos que ajudaram os programadores a superar a crise do software e observa quais aspectos desses conhecimentos podem ser aplicáveis à prática da modelagem paramétrica na arquitetura. Por exemplo: em engenharia de software é comum a utilização de fases na produção do ciclo de vida do software, como por exemplo: projeto, codificação, depuração e tarefas de modificação (WHITLEY, 1997). Esse particionamento das tarefas segue uma metodologia que facilita o controle e entendimento em programação.

A separação de fases no processo de projeto é largamente utilizada na arquitetura, no entanto, ainda existem lacunas quando se utiliza programação:

O *Design Thinking* é uma estrutura de conceitos que ajuda as equipes por meio de fases iterativas de inspiração, ideação e implementação, com diversos subprocessos para guiar as equipes por essas fases (Rowe, 1987; Cross et al., 1992; Brown, 2008). No entanto, esses métodos são focados na formulação e solução criativa de problemas, e carecem de um foco computacional para guiar equipes na implementação de processos computacionais para apoiá-los nessas fases. (HAYMAKER et al., 2018, p. 158, tradução nossa¹⁰⁶)

O estudo de aplicação utilizado nesta pesquisa parece separar essas fases de maneira interessante. Trata-se do projeto *Vancouver Academic Building*, desenvolvido segundo a metodologia *Design Space Construction* (DSC) (HAYMAKER et al., 2018), ambos propostos pelo escritório Perkins+Will da cidade de Atlanta, Geórgia, nos Estados Unidos.

Analisaremos a comparação entre VPLs e TPLs no contexto do Projeto Orientado ao Desempenho, que possui as fases de concepção, simulação e otimização.

Por requerer muitos requisitos de desempenho quantificáveis para guiar o projeto, este fluxo de trabalho necessita um controle maior do código durante o desenvolvimento dos algoritmos: “o pensamento mais racional e a análise científica são trazidos para o campo do projeto arquitetônico” (SHI; YANG, 2010, p. 125, tradução nossa¹⁰⁷). Desta maneira, as análises realizadas durante e após a implementação do Estudo de Aplicação, nos ajudam a ir em direção a um processo de projeto mais transparente, aberto e confiável, em resposta às lacunas e problemas apontados por Terzidis (2006), conforme apresentado na seção de introdução desta dissertação.

¹⁰⁶ Do original em inglês: “*Design Thinking is a framework of concepts that helps teams through iterative phases of inspiration, ideation, and implementation, with several sub processes to guide teams through these phases (Rowe, 1987; Cross et al, 1992; Brown, 2008). However, these methods are focused on creative problem formulation and solving, and lack a computational focus to guide teams in the implementation of computational processes to support them in these phases.*”

¹⁰⁷ Do original em inglês: “*performance-driven design is encouraged and more rational thinking and scientific analysis are brought into the field of architectural design.*”

4.2.1 Vancouver Academic Building

O projeto Vancouver Academic Building e o fluxo de trabalho Design Space Construction (HAYMAKER et. al, 2018) foram utilizados pelo escritório Perkins+Will durante um Workshop oferecido como parte das atividades do congresso internacional ACADIA (Association for Computer Aided Design in Architecture). A pesquisadora participou da atividade e obteve treinamento e autorização para utilizar o projeto como Estudo de Aplicação. O escritório disponibilizou a pasta de arquivos contendo o código de programação visual em Grasshopper, planilhas e documentos com informações necessárias para a execução do código. O código disponibilizado apenas considerava a parametrização de um dos pavimentos do edifício. A pesquisadora realizou a adaptação de todo o código em Grasshopper para iniciar a investigação do projeto.

O edifício utilizado prevê o projeto para a fachada do *Vancouver Academic Building*, do escritório Perkins+Will Atlanta. Trata-se de um edifício educacional composto por duas alas de salas de aula, separadas por um átrio central que as conecta (Figura 50). Os novos painéis que deverão compor a fachada possuem diferentes configurações de envidraçamento (Figura 51).

Figura 50: A fachada do edifício existente. Fonte: Perkins+Will

Figura 51: Os oito tipos de painéis de irão compor a nova fachada do edifício. Fonte: Haymaker et al. (2018).

O projeto se enquadra como um bom estudo de aplicação para esta pesquisa por incorporar um fluxo de trabalho computacional que prevê, entre seus objetivos de projeto, atender as certificações da *International Passive House Association* (IPHA) e LEED, que constituem objetivos concorrentes e que dificilmente podem ser otimizados de maneira eficaz sem o uso do Design Computacional.

Os painéis (Figura 51) posicionados na fachada devem estar adaptados ao programa arquitetônico dentro do edifício. Os ambientes internos foram agrupados por zonas térmicas (Figura 52), as regras para esse agrupamento foram: (1) os espaços têm programas de ambiente e carga energética (como densidade de iluminação, carga de equipamento por área, ventilação por pessoa, etc) semelhantes; (2) as zonas exteriores com vidro têm a mesma orientação; e (3) as zonas podem ser servidas pelo mesmo tipo de sistema AVAC (aquecimento, ventilação e ar condicionado).

Figura 52: Planta do 3º pavimento. Os ambientes foram separados por zonas térmicas (em vermelho) que serão usadas nas análises de conforto ambiental. Fonte: Perkins+Will

São essas zonas térmicas que definem os grupos de fachada onde os painéis poderão ser posicionados (Figura 53).

Figura 53: Fachadas do edifício e seus respectivos Grupos de Painéis. Fonte: Perkins+Will.

O projeto utiliza um fluxo de trabalho orientado ao desempenho uma vez que testa a melhor combinação de fachada para equilibrar alguns aspectos de conforto térmico e eficiência energética, como a incidência de luz do dia dentro das salas e o ganho e perda de calor no desempenho energético. Devido a grande quantidade de possibilidades de posicionamento dos painéis nas fachadas (Figura 54), as alternativas podem ultrapassar milhares de opções possíveis, além de cada combinação ter um desempenho

diferente no balanceamento das métricas de performance.

Diferentes alternativas para o mesmo problema de projeto podem apresentar melhor ou pior desempenho em relação a cada objetivo, e diferentes partes interessadas podem ter diferentes prioridades em relação aos seus objetivos nesses fatores. As equipes de projeto devem, de alguma forma, envolver as partes interessadas, entender metas e preferências, gerar e analisar alternativas, fazer e comunicar decisões que maximizem o valor [...] Hoje, as equipes de projeto lutam para reunir e comunicar toda essa informação, causando uma exploração de projeto ineficiente e ineficaz e uma perda de valor potencial. (PERKINS+WILL, 2017, p. 4, tradução nossa¹⁰⁸)

4.2.2 Design Space Construction

O *Design Space Construction* é uma metodologia, também desenvolvida pelos pesquisadores da Perkins+Will Atlanta, que estrutura conceitos e processos que apoiam equipes de projeto na construção e exploração de espaços de design. Essa estrutura guia as equipes por meio de processos de formulação de problemas, geração de alternativa de projetos (Figura 55), análise de impacto e avaliação de valor dessas alternativas geradas.

Em uma visão aplicada, o DSC oferece um modelo computacional que dá suporte para definição de objetivos de projeto, geração de modelos paramétricos e simulações de alternativas de projeto. Para isso, foram estabelecidas camadas de interação entre as fases do fluxo de trabalho. O DSC pode ser aplicado no projeto de edifícios/construções diferentes: “Essa infraestrutura ajuda as equipes a organizar e definir os objetivos, análises, intervalos para cada variável de projeto e automatiza a geração,

Figura 54: Gráfico de Coordenadas Paralelas referente aos inputs e resultados de todas as alternativas de projeto. Fonte: Perkins+Will

¹⁰⁸ Do original em inglês: “Different alternatives for the same design problem may perform better or worse on each objective, and different stakeholders may have different priorities over their objectives on these factors. Design teams must somehow engage project stakeholders, understand goals and preferences, generate and analyze alternatives, and make and communicate decisions that maximize value. [...] Today, design teams struggle to gather and communicate all of this information, causing inefficient and ineffective design exploration, and a loss of potential value.”

a análise e a visualização de dados do espaço de design” (HAYMAKER et al., 2018, p. 161, tradução nossa¹⁰⁹).

Figura 55: Fluxo de trabalho de formulação de decisão de projeto. Fonte: Adaptado de PERKINS+WILL (2017).

O estudo de aplicação *Design Space Construction* (DSC) foi escolhido por apresentar um modelo paramétrico bem estruturado e de complexidade particularmente relevante para a área do Design Computacional.

A metodologia separa o fluxo de trabalho em quatro camadas (Figura 56). A maior parte, camadas Parametric Modeling, Plugins e Analysis Engine, foi implementada usando a VPL Grasshopper. Os detalhes do funcionamento de cada uma delas compreendem:

Figura 56: Camadas da infraestrutura computacional. Fonte: Haymaker et al., 2017.

(1) Visualização de dados:

Numa ponta (*data visualization*), essa camada reúne alguns dados sobre as preferências de projeto estabelecidas pelas partes interessadas (como arquitetos, consultores, clientes, etc) em uma planilha. A equipe envolvida pode gravar esses dados na planilha e acessar todos os parâmetros que quiser sobre o projeto, por exemplo, as dimensões estabelecidas para cada painel que irá compor a fachada do edifício (Figura 58), os tipos de painéis (Figura 51) que podem ser alocados em cada zona de fachada (Figura 57), entre outros.

¹⁰⁹ Do original em inglês: “This infrastructure helps teams organize and define the objectives, analyses, ranges for every design variable, and automates the generation, analysis and data visualization of the design space.”

Panel Group Definitions	Allowed Panels	Panel Types	Panel Name	Bottom Offset	Right Offset	Top Offset	Left Offset
1	1 2 4 6	0	Atrium Fully Glazed	0,1	0,1	0,1	0,1
2	0 1 2 3 6	1	Glazed	0,4	0,4	0,4	0,4
3	0 1	2	Bench	1,17	0,4	0,4	0,4
4	none	3	Dropped Ceiling	0,4	0,4	1,4	0,4
5	none	4	Bench + Dropped Ceiling	1,7	0,4	1,4	0,4
6	0 1 2 3	5	Clerestorey	2,8	0,4	0,4	0,4
7	4 5 7	6	Half Vertical	0,4	0,4	0,4	1,35
8	2 3 4 5 6	7	Fully Solid	2	1,25	2	1,25
9	2 3 4 5 6						
10	2 3 4 5 6						
11	5 6 7						
12	none						
13	none						
14	5 7						
15	none						
16	0 1						

Figura 57 (à esquerda): Dados dos painéis permitidos serem posicionados em cada zona de fachada.

Figura 58: Dimensão dos painéis da fachada definidos pela equipe de projeto na planilha.

Na outra ponta (*value assessment*) o código no Grasshopper grava na mesma planilha todos os parâmetros atribuídos às alternativas de projeto, assim como as imagens do modelo e resultados das simulações de conforto ambiental (ou outra simulação que estiver associada ao projeto). Esse arquivo pode ser carregado e visualizado na interface *Design Explorer*.

(2) Modelagem paramétrica:

Todos os modelos geométricos do edifício (alternativas de projeto), assim como todo o fluxo de trabalho é programado usando a VPL Grasshopper (Figura 60) (o escritório também implementou em Dynamo). A modelagem paramétrica gera o modelo analítico com os atributos necessários para cada análise de simulação.

Figura 59: Interface Design Explorer. O gráfico de coordenadas paralelas (em cima) mostra todos os parâmetros de projeto e resultados de desempenho. É possível visualizar todas as alternativas de projeto (embaixo).

Figura 60: Código do DSC na VPL Grasshopper.

O objetivo do método é explorar a combinação de parâmetros que melhor atende o balanceamento dos seguintes itens:

Window Wall Ratio (WWR, razão entre a área de janela e parede) das fachadas, o resultado é em áreas percentual; Iluminação dentro da edificação, o resultado equivale a porcentagem da área dentro da faixa aceitável; Refrigeração (*cooling*) em kWh / m2; e Aquecimento (*heating*) em kWh / m2.

Por se tratarem de valores de simulação com unidades de medidas diferentes, os resultados são normalizados (todos os valores ficam na mesma escala, de 0 a 1), através da função:

$$\text{fator-min(fator)} / \text{max(fator)-min(fator)}$$

Desta maneira, a função objetivo da otimização considera:

$$(\text{wwr} * \text{wwr_peso}) + (\text{cooling} * \text{cooling_peso}) + (\text{heating} * \text{heating_peso}) + (\text{lighting} * \text{lighting_peso})$$

Os pesos dos fatores são definidos em um acordo entre a equipe de projeto e clientes. Por exemplo: 0.2 para WWR, 0.3 para cooling, 0.2 para heating, 0.3 para lighting. Os pesos somam cem por cento.

Por nos concentrarmos na comparação dos fluxos de trabalho usando linguagens diferentes, não faz parte desta pesquisa discutir o método de otimização utilizado pelo escritório. Essas informações são úteis para melhor entender os objetivos do projeto por ser orientado ao desempenho.

As variáveis do projeto são:

Área de fachada (Panel Group) 1

Painéis: 1, 2, 4, 6

Área de fachada (Panel Group) 2

Painéis: 0, 1, 2, 3, 6

Área de fachada (Panel Group) 3

Painéis: 0, 1

Área de fachada (Panel Group) 4

Painéis: 7

Área de fachada (Panel Group) 5

Painéis: 7

Área de fachada (Panel Group) 6

Painéis: 0, 1, 2, 3

Área de fachada (Panel Group) 7

Painéis: 4, 5, 6

Área de fachada (Panel Group) 8

Painéis: 2, 3, 4, 5, 6

Área de fachada (Panel Group) 9

Painéis: 2, 3, 4, 5, 6

Área de fachada (Panel Group) 10

Painéis: 2, 3, 4, 5, 6

Área de fachada (Panel Group) 11

Painéis: 5, 6, 7

Área de fachada (Panel Group) 12

Painéis: 7

Área de fachada (Panel Group) 13

Painéis: 7

Área de fachada (Panel Group) 14

Painéis: 5, 7

Área de fachada (Panel Group) 15

Painéis: 7

Área de fachada (Panel Group) 16

Painéis: 0, 1

Atributos permitidos:

R-Value: 5.0, 6.0, 7.0

U-Value: 0.65, 0.7, 0.8

SHGC: 0.3, 0.4, 0.5

Dimensões dos brises:

Brises Fachada Sul: 0, 0.5, 1

Brises Fachada Leste-Oeste: 0, 0.5, 1

Brises Fachada Norte: 0.05, 0.5, 1

(3) Interface (*plugins/wrappers*):

As interfaces são os plugins que permitem a conexão entre Grasshopper e os programas de análise e simulação. É nesta camada que estão os plugins que coletam os dados da planilha (Excel Read, da TT Toolbox para Grasshopper, CORE Studio), gravam os resultados das análises na planilha e salvam as imagens do projeto (Colibri, da TT Toolbox for Grasshopper, CORE Studio) para pós-processamento e visualização posterior. Os plugins Honeybee & Ladybug

conectam Grasshopper aos programas de análise EnergyPlus, Radiance, Daysim e OpenStudio (SADEGHPOUR ROUDSAR; PARK; SMITH, 2013)

(4) Mecanismos de análise:

São considerados mecanismos de análise todos os softwares necessários para a simulação de desempenho. O escritório sintetiza o objetivo de cada um deles:

As simulações de energia são baseadas no motor EnergyPlus (EnergyPlus, 2016) que requer modelos de entrada analítica representando zonas térmicas, atributos relacionados e dados meteorológicos para calcular a temperatura dentro de cada sala para cada hora do ano, para estimar o uso total de energia em kWh. O software adicional OpenStudio (OpenStudio, 2016) facilita o processo exportando as zonas e seus atributos para o EnergyPlus, produzindo relatórios de entradas e saídas e mantendo o controle dos arquivos IDF do EnergyPlus. Simulações de luz do dia dependem do mecanismo DAYSIM (DAYSIM, 2016) baseado em Radiance (Radiance, 2016) que realiza os cálculos usando a geometria do envelope e as propriedades do material. O processo simula os raios que representam a luz refletida no espaço e cria uma grade de pontos de teste a certa altura acima do solo para registrar e traçar o nível da luz do dia em cada ponto” (HAYMAKER et al., 2018, p. 162, tradução nossa¹¹⁰)

A implementação do estudo de aplicação para observar as diferenças e implicações entre o fluxo de trabalho que usa VPL e TPL também possui potencial de refletir sobre a metodologia DSC enquanto uma estrutura de um processo de projeto orientado a desempenho:

A implementação do fluxo de trabalho depende da integração de modelagem paramétrica e ferramentas de análise em estágios preliminares de projeto conceitual fornecem uma representação geométrica consistente e indicadores de desempenho precoces para uma grande população de alternativas. No entanto, até mesmo alguns dos processos descritos [nessa pesquisa] são ad-hoc e ainda existem várias oportunidades relacionadas à interatividade, automação, escalabilidade e extensibilidade dessa estrutura modular para abordar análises além do escopo do caso de teste. (HAYMAKER et al., 2018, p. 177, tradução nossa¹¹¹).

4.2.3 Implementação do código TPL

Para comparar o fluxo de trabalho entre VPL e TPL, reimplementamos o estudo de aplicação utilizando a linguagem de programação textual Racket no ambiente de programação Rosetta

¹¹⁰ Do original em inglês: “Energy simulations are based on the EnergyPlus engine (EnergyPlus, 2016) that requires analytical input models representing thermal zones, related attributes, and weather data to calculate the temperature inside each room for every hour of the year, in order to estimate the total energy use, in kWh. The additional OpenStudio software (OpenStudio, 2016) facilitates the process by exporting the zones and their attributes to EnergyPlus, producing reports of inputs and outputs, and keeping track of the EnergyPlus IDF files. Daylight simulations rely on the DAYSIM engine (DAYSIM, 2016) based on Radiance (Radiance, 2016) which undertakes the calculations using the envelope geometry and the material properties. The process simulates rays representing light bouncing within the space and creates a grid of test points a certain height above the floor to register and plot the daylight level at each point.”

¹¹¹ Do original em inglês: “The workflow implementation that relies on the integration of parametric modeling and analysis tools in preliminary conceptual design stages provides consistent geometric representation and performance indicators early on for a large population of alternatives. However, even some of the processes described above are ad-hoc, and there remain several opportunities regarding interactivity, automation, scalability and extensibility of this modular structure to address analyses beyond the scope of the test case.”

(LOPES; LEITÃO, 2011). Os resultados deste estudo visam orientar as melhorias futuras dos Sistemas de Design Paramétrico, ao observar maneiras de aumentar a confiabilidade dos fluxos de trabalho em projeto algorítmico.

A implementação da versão do projeto em TPL foi realizada junto ao grupo de pesquisa ADA (Algorithmic Design for Architecture), do Instituto Superior Técnico da Universidade de Lisboa. A equipe envolvida diretamente no desenvolvimento da pesquisa foi:

Inês Caetano, doutoranda da Arquitetura (INESC-ID/IST, Universidade de Lisboa). A pesquisadora focou na implementação do código em Racket e no ambiente de programação Rosetta;

Catarina Belém, mestranda da Engenharia de Informática (INESC-ID/IST, Universidade de Lisboa). A pesquisadora focou no desenvolvimento em Racket e no desenvolvimento do código necessário para viabilizar a comunicação para realizar a fase de otimização do projeto;

Dyego Digiandomenico, mestrando da Arquitetura (IAU-USP) e pesquisador com estágio no mesmo grupo (ADA). O pesquisador desenvolve a dissertação sobre métodos de otimização em arquitetura e realiza análise crítica dos fluxos do trabalho deste estudo de caso;

Prof. Dr. António Leitão (INESC-ID/IST, Universidade de Lisboa), coordenador do grupo ADA, orientador de todas as pesquisas desenvolvidas pelo grupo e supervisor desta implementação.

4.2.4 Rosetta

Para a implementação TPL, utilizamos Rosetta, um ambiente de programação que permite o uso de diferentes linguagens de programação (*front-ends*) e gera o modelo tridimensional para diferentes programas CAD e BIM (*back-ends*) (LEITÃO; SANTOS; LOPES, 2012) (Figura 61). Desta maneira, um único programa escrito em Rosetta pode criar modelos idênticos em programas diferentes:

O Rosetta permite que os usuários explorem diferentes front-ends e back-ends, a fim de encontrar uma combinação mais adequada para o problema em questão. Como resultado, os usuários têm acesso a diferentes linguagens de programação, a saber, visual e textual, que podem ser usadas de forma intercambiável para escrever programas de design generativo, sem quebrar a portabilidade (LOPES; LEITÃO, 2011, p. 196, tradução nossa¹¹²).

Rosetta é composto pelas camadas:

- (1) *front-ends*: pelo qual escolhe-se uma das linguagens de programação disponíveis, como Racket, JavaScript, Python e Processing;
- (2) uma biblioteca para a construção de sólidos, geometrias e operações necessárias para

¹¹² Do original em inglês: “Rosetta allows users to explore different front-ends and back-ends, in order to find a combination that is most suitable for the problem at hand. As a result, users have access to different programming languages, namely, visual and textual, which can be used interchangeably to write generative design programs, without breaking portability.”

modelagem tridimensional, com uma camada de abstração que permite gerar modelos analíticos, CAD e famílias BIM; e

(3) *back-ends* que traduzem a camada de abstração para os softwares disponíveis, como AutoCad, Rhinoceros, OpenGL, Revit, ArchiCAD, Radiance e Robot.

Figura 61: Esquema de linguagens de programação textuais (à esquerda) suportadas pelo ambiente de programação Rosetta (centro) e programas conectados ao ambiente para receber a saída do modelo tridimensional (à direita).
Fonte: Castelo Branco e Leitão (2018)

Optamos por utilizar o Rosetta por ser simultaneamente: (1) um ambiente de programação para arquitetura que permite a escalabilidade tanto do código quanto do modelo tridimensional, ao permitir interoperabilidade entre modelos analíticos, CAD e BIM; e (2) ser um ambiente implementado e em desenvolvimento no grupo de pesquisa português *Algorithmic Design for Architecture* (ADA), onde a pesquisadora desenvolveu o estágio de pesquisa para implementação do estudo de aplicação.

4.3 Implementação

As informações presentes nesta dissertação, sobretudo no Capítulo 2, foram baseadas em discussões e aspectos geralmente relatados em artigos recentes de pesquisadores envolvidos com a prática da programação em arquitetura.

A discussão, representa de maneira geral, uma tendência entre reconhecer que as VPLs tornaram a programação mais amigável a um público mais vasto de não programadores (AISH, 2013), enquanto as TPLs são linguagens mais completas e permitem escalar o código que se torna complexo de maneira eficaz.

Neste capítulo, a comparação foi realizada entre a VPL Grasshopper, que suporta modelagem de fluxo de dados e a TPL Racket/Rosetta, através do paradigma de programação funcional¹¹³.

Implementamos o *Vancouver Academic Building* usando o mesmo fluxo de trabalho DSC com a TPL Racket/Rosetta, portanto as camadas escolhidas foram:

(1) *Front-end*: Racket. Utilizamos a TPL Racket por ser a linguagem em

¹¹³ A programação funcional enfatiza o uso de funções, em contraste com a programação imperativa, que enfatiza mudanças no estado do programa.

que Rosetta foi escrito (conforme descrevemos na seção 2.6 Interfaces de programação TPL para arquitetura), desta maneira, é a linguagem que os pesquisadores do grupo ADA estão mais familiarizados e na qual a implementação do estudo de aplicação pôde ser realizada de forma mais rápida. Adicionalmente, a linguagem aceita o paradigma de programação funcional adotado.

(2) *Back-end*: Rhinoceros. Além de gerar o modelo tridimensional e analítico no Rhinoceros, o código também cria as partes do edifício separado em famílias BIM.

4.3.1 Substituição de Grasshopper por Rosetta

Utilizaremos diagramas para explicar os três fluxos de trabalho encontrados durante a pesquisa com o Estudo de Aplicação, são eles: (1) original, no qual explicaremos o que foi implementado e disponibilizado inicialmente pelo escritório; (2) ideal, onde consta a substituição pretendida da VPL Grasshopper pela TPL Racket/ Rosetta e; (3) o fluxo de trabalho atual, onde apresentamos o diagrama do que foi implementado nesta pesquisa e porquê.

(1) Original

A Figura 62 mostra um diagrama das etapas dos fluxos de trabalho original, implementado no DSC pela Perkins+Will. O fluxo horizontal do diagrama indica que as fases do Projeto Orientado ao Desempenho estão separadas e possuem três etapas: modelo paramétrico, simulação e otimização. Na vertical, podemos identificar quais interfaces de programação ou programas foram utilizados em cada uma das etapas.

As bases do edifício existente foi modelada no Rhinoceros e fornecido como input inicial de geometria no Grasshopper. O código no Grasshopper gera uma série de alternativas randômicas dos painéis de fachada do edifício e exporta e grava esses dados em uma planilha no Excel. O Grasshopper acessa na planilha a primeira coleção de parâmetros que definem o posicionamento dos painéis na fachada, o modelo desta opção é gerado na

Figura 62: Fluxo de trabalho computacional implementado originalmente pelo escritório Perkins+Will.

visualização do Rhinoceros (3d model). A partir desse momento, inicia-se o processo de iteração entre as fases do Projeto Orientado ao Desempenho – no caso do DSC, as fases que iteram são design e simulação.

Os plugins de análise executam as simulações de iluminância e eficiência energética a cada alternativa de projeto (com Honeybee e Ladybug), os resultados dessas simulações são gravados iterativamente na planilha Excel. Após o término das iterações, os parâmetros de cada alternativa de projeto e seus respectivos resultados nas simulações recebem a função de avaliação na planilha.

O resultado da função de avaliação atribui um valor final de desempenho a cada uma das alternativas de projeto. Elas são ranqueadas a fim de verificar qual possui melhor performance para os critérios estabelecidos pela equipe de projeto.

Qualquer uma das opções de modelo pode ser resgatada e visualizada pelo Grasshopper.

(2) Ideal

A implementação ideal contava com a substituição completa de todo o código do Grasshopper pela TPL Racket/Rosetta (Figura 63). Neste caso, as simulações de iluminância e eficiência energética aconteceriam diretamente no programa Energy Plus. No entanto, a comunicação entre Rosetta e EnergyPlus está em processo de desenvolvimento e ainda não foi finalizada pelo grupo de pesquisa ADA.

Figura 63: Fluxo de trabalho computacional previsto inicialmente, considerando a substituição ideal da VPL pela TPL.

(3) Atual

Desta maneira, optamos por manter os simuladores do DSC original Honeybee e Ladybug, *plugins* que permitem a conexão e transferência de dados entre Grasshopper (VPL para CAD), Radiance, Daysim, EnergyPlus e OpenStudio.

Sendo assim, todo o fluxo de trabalho foi reimplementado usando a TPL Racket/Rosetta, exceto na fase de simulação, onde o modelo paramétrico é enviado para Grasshopper, para que as alternativas de projeto sejam analisadas pelos simuladores Honeybee e Ladybug (Figura 64).

Este desvio no fluxo de trabalho não comprometeu a substituição da VPL, uma vez que todo o trabalho ainda é feito pela TPL. Adicionalmente, isso gerou uma camada extra de comunicação entre todas as interfaces conectadas. Com isto, surgiu uma série de exemplos práticos sobre algumas das limitações da VPL Grasshopper, que será relatada com mais detalhes na seção 4.4.3 Comunicação com simuladores.

4.4 Análise da implementação

Após a implementação do estudo de aplicação, as observações realizadas foram agrupadas em tópicos e serão analisadas na próxima seção. A metodologia utilizada para realizar essa análise foi identificar os pontos críticos que emergiram durante o trabalho de implementação do código. Os problemas e aspectos gerais presentes dentro destes tópicos foram analisados a partir do escopo das Dimensões Cognitivas e as métricas de avaliação de PLs discutidas no Capítulo 3.

4.4.1 Mudança na topologia do projeto

Para que pudéssemos iniciar a implementação e análise do estudo e caso em TPL, foi necessário realizar uma adaptação no código Grasshopper disponibilizada pelo escritório. A equipe possuía a versão do código correspondente a apenas um dos pavimentos do projeto (Figura 65) e foi necessário alterá-lo para que o modelo paramétrico correspondesse ao edifício todo (Figura 66).

Desta maneira, teríamos o modelo completo para considerá-lo como a opção original, base da comparação.

Figura 64: Fluxo de trabalho computacional implementado na pesquisa, devido às restrições de conexão entre Rosetta e simuladores.

Figura 65: Código disponibilizado pelo escritório P+W referente a apenas um dos pavimentos do edifício.

Figura 66: Código e modelo adaptados para servir de base na comparação da implementação.

Dois principais problemas foram identificados já nesta etapa e ilustram a dimensão cognitiva **Viscosidade**:

(1) mudança na topologia geral do projeto em função da mudança de número de pavimentos:

A geometria base que constrói o edifício é composta por linhas CAD. Assim, para cada novo pavimento necessário no edifício, é preciso adicionar o pavimento manualmente à uma lista. Isso pode ser resolvido ao alterar a lógica de adição de pavimentos. Ao invés de adicionarmos as linhas base CAD manualmente, é possível recriá-las parametricamente (recriar uma laje e repeti-la quantas vezes for necessário através da mudança de um *input* numérico inicial que indica o número total de pavimentos, por exemplo).

(2) aumento do número de painéis em função do aumento de número de pavimentos:

(a) os novos painéis precisam ser adicionados manualmente como *input* nos nós que manipulam os dados (nó *Entwine* do Grasshopper). Isso pode ser resolvido substituindo o nó que requer *inputs* adicionados manualmente por outro nó que considere uma lista paramétrica e automatize a tarefa, independentemente do tamanho da lista de entrada.

Especificamente, o Grasshopper não possui um nó equivalente para isso. Portanto, neste ponto, seria necessário: (i) escrever um componente especializado para substituí-lo, ou seja, recorrer ao *scripting* em TPL ou (ii) repensar outra construção lógica para este trecho de código (um plano B).

(b) a quantidade de novos painéis também devia ser declarada em outros três locais do código que dependiam destes dados (Figura 68). Esta lógica não estava expressa parametricamente no código original. Trata-se de um problema de **Dependências Ocultas**, quando componentes dependem um do outro, mas essa dependência não é totalmente visível.

Figura 67: Canvas do código DSC em Grasshopper. As setas vermelhas indicam os locais que possuem Dependências Ocultas.

Para resolvermos tanto (a-ii) quanto (b), seria necessário traçar outra estratégia para recombinação da relação fachada vs. painéis permitida, atualmente declarada manualmente através da combinação de *inputs* em uma ordem específica (Figura 68). No entanto, a lógica desta combinação não é trivial, pois três instâncias de dados estão sendo combinadas: restrições de projeto definidas pela equipe de arquitetura (lista de números inteiros com o número de cada painel), trecho de fachada em que os painéis permitidos podem ser alocados (referenciados por números inteiros) e posição no plano cartesiano de alocação desses painéis no modelo (lista de linhas no Rhino/Grasshopper). Isso significa empregar muito tempo e esforço na edição do modelo, ou até mesmo reconstruir grande parte do código.

Figura 68: O nó Entwine recebe as entradas em ordem cruzada, de acordo com os painéis permitidos por área de fachada. Os inputs precisam ser adicionados manualmente e para parametrizar esta lógica, é necessário reconstruir grande parte do código.

Temos aqui um encadeamento de problemas nomeados pelas Dimensões Cognitivas apresentadas.

Este problema ilustra que há um alto grau de **Viscosidade** deste código em Grasshopper, mesmo em um caso em que há um planejamento inicial desta estrutura, uma vez que o modelo paramétrico originalmente desenvolvido pelo escritório foi planejado e programado considerando todo o edifício (e só posteriormente ganhou uma versão reduzida de um pavimento, para testar as análises de simulação mais rapidamente). Ou seja, nem mesmo o baixo grau de **Comprometimento Prematuro**, que deveria facilitar a resolução deste problema, ajudou a diminuir a **Viscosidade**. Além disso, a alta Viscosidade também dificultou a resolução das **Dependências Ocultas**

existentes. Por estes motivos, avaliamos um desempenho negativo da VPL em ambas as dimensões cognitivas (Figura 70).

Pelo tempo disponível, optamos por adaptar o edifício para três pavimentos adicionando os inputs aos nós manualmente, mantendo a estrutura do código original. No entanto, esta resolução não resolve a flexibilidade e escalabilidade do modelo, caso os pavimentos voltem a aumentar ou outras modificações topológicas no projeto sejam realizadas.

(TPL)

Já na implementação em TPL utilizando Rosetta, para resolver o problema da mudança na topologia do projeto, as estruturas da linguagem Racket ofereceram mais flexibilidade ao aumentar ou diminuir o número de pavimentos do edifício. Por exemplo:

Caso a relação de alguma restrição de fachada vs. painel mude ou seja adicionada, é mais fácil alterar apenas os parâmetros específicos escrito no código. Um exemplo é a função que define os painéis permitidos na área de fachada 1.

A linha de código abaixo indica que a área de fachada 1 (*g1*) só pode receber os painéis do tipo 1, 2, 4 e 6 (*windowX*). O objeto “*window-X-pt*” representa o ponto no plano cartesiano onde o painel será posicionado.

```
(define g1 (list (list window1 window1-pt) (list window2 window2-pt) (list window4 window4-pt) (list window6 window6-pt)))
```

Caso uma nova restrição de painéis nesta área de fachada seja adicionada às intenções de projeto, basta adicionar o painel desejado (em negrito), ou então, excluir o respectivo painel desejado:

```
(define g1 (list (list window1 window1-pt) (list window2 window2-pt) (list window4 window4-pt) (list window6 window6-pt) (list window7 window7-pt)))
```

Caso o número de pavimentos do edifício aumente e novas áreas de fachada sejam adicionadas, uma nova linha “(*define g_numeroNovaAreaFachada*)” deve ser adicionado no código.

Caso isso aconteça, será possível identificar em quais outros locais do código a nova função *define* deverá ser adicionada, pois a IDE Dr. Racket da linguagem utilizada indica onde estão as outras “ocorrências vinculadas” (Figura 69). Estas características refletem no aspecto positivo da TPL para diminuir ou resolver problemas de **Dependências Ocultas** (Figura 70).

```

#lang racket
(define window1-pt (xy 0.4 0.4))
;janela tipo2
(define window2 (window-family-element (default-window-family) #:width 1.7 #:height 2.43))
(define window2-pt (xy 0.4 1.17))
;janela tipo3
(define window3 (window-family-element (default-window-family) #:width 1.7 #:height 2.2))
(define window3-pt (xy 0.4 0.4))
;janela tipo4
(define window4 (window-family-element (default-window-family) #:width 1.7 #:height 1.43))
(define window4-pt (xy 0.4 1.17))
;janela tipo5
(define window5 (window-family-element (default-window-family) #:width 1.7 #:height 0.8))
(define window5-pt (xy 0.4 2.8))
;janela tipo6
(define window6 (window-family-element (default-window-family) #:width 0.75 #:height 3.2))
(define window6-pt (xy 0.1 0.4))
;janela tipo7
(define window7 (window-family-element (default-window-family) #:width 0 #:height 0))
(define window7-pt (x 0))
;janela do skylight
(define glazing-s (window-family-element (default-window-family) #:width 2.40 #:height 1.85))
(define glazing-l (window-family-element (default-window-family) #:width 2.44 #:height 1.85))
(define glazing-pt (xy 0.05 0.05))

;-----
; Agrupamento das janelas por grupos
;-----
; Cada grupo é representado por uma lista de painéis
; Cada painel é representado por uma janela (windowX) e por um ponto onde vamos posicionar (window-X-pt)
(define g1 (list (list window1 window1-pt) (list window2 window2-pt) (list window4 window4-pt) (list window6 window6-pt)))
(define g2 (list (list window0 window0-pt) (list window1 window1-pt) (list window2 window2-pt) (list window6 window6-pt)))
(define g3 (list (list window0 window0-pt) (list window1 window1-pt)))
(define g4 (list (list window7 window7-pt)))
(define g5 (list (list window7 window7-pt)))
(define g6 (list (list window0 window0-pt) (list window1 window1-pt) (list window2 window2-pt) (list window3 window3-pt)))
(define g7 (list (list window4 window4-pt) (list window5 window5-pt) (list window6 window6-pt)))
(define g8 (list (list window2 window2-pt) (list window3 window3-pt) (list window4 window4-pt) (list window5 window5-pt) (list window6 window6-pt)))
(define g9 (list (list window2 window2-pt) (list window3 window3-pt) (list window4 window4-pt) (list window5 window5-pt) (list window6 window6-pt)))
(define g10 (list (list window2 window2-pt) (list window3 window3-pt) (list window4 window4-pt) (list window5 window5-pt) (list window6 window6-pt)))
(define g11 (list (list window5 window5-pt) (list window6 window6-pt) (list window7 window7-pt)))

```

A implementação TPL seguiu a mesma estrutura de declaração dos painéis que a versão VPL. Ambas poderiam ter sido implementadas completamente parametrizadas com qualquer quantidade de áreas de fachadas. A diferença entre elas ainda permanece no fato de que a TPL possui mais estruturas de controle para facilitar esta declaração, uma vez que dada a complexidade das relações de painéis na VPL, em algum momento seria necessário recorrer a um nó especializado (TPL).

Para que essas listas fossem completamente parametrizadas, poderíamos recorrer a mecanismos de **Abstração**, conhecidos por melhorar a **Viscosidade**.

O uso de abstrações em programação possui um retorno e gratificação menos imediato, porém são essenciais para resolver os problemas de flexibilidade geralmente relatados na programação com VPLs em arquitetura (como em Davis (2013) e Chaszar e Joyce (2016)), uma vez que ajudam a definir aspectos gerais de um contexto qualquer e ignorar características menos importante para a generalidade deste contexto (GREEN; PETRE, 1996). As abstrações são consideradas um dos pilares da Programação Orientada a Objetos (POO) (GREEN; PETRE, 1996), como já mencionado na seção 3.2 Dimensões Cognitivas.

As resoluções para o problema de mudança na topologia do projeto exemplificadas pelas implementações VPL e TPL acima nos mostram que a TPL possui **Viscosidade** significativamente menor, a ponto de viabilizar a adição de novos Grupos de Painéis com muito menos esforço. Isto resulta na avaliação positiva da TPL em Viscosidade (Figura 70). Além disso, devido à sua baixa Viscosidade em relação a VPL, a TPL permite manter a relação entre as funções definidas de maneira mais simples, possui mais mecanismos de abstração ao suportar paradigmas de programação diferentes, facilitando a implementação de métodos de Programação Orientada a

Figura 69: Funcionalidade da IDE Dr. Racket que indica onde estão localizados os objetos vinculados ao objeto selecionado.

Objetos e/ou Metaprogramação (no caso do uso do paradigma funcional).

Figura 70: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico "Mudança na topologia do projeto".

4.4.2 Relação e ordenação entre e listas

(VPL)

Ainda durante a fase de adaptação para que o código VPL passasse a corresponder ao edifício todo, identificamos a falta de consistência em como Grasshopper considera a ordem de listas quando importa as linhas CAD criadas no programa Rhinoceros.

A lista de linhas CAD que são importadas para dentro do Grasshopper, darão origem aos limites das áreas de fachada do edifício (Figura 71). Quando informadas como *inputs*, precisam ter a mesma sequência de entrada em nós diferentes, para que a consistência entre a ordem dos dados seja mantida (Figura 72). Esse tipo repetição de *inputs* (os mesmos dados sendo referenciados em partes diferentes e desconectadas do código) pode passar despercebido para o programador da VPL. Neste caso, Grasshopper não levanta exceções (acusa erros), pois o tipo de dado está correto, embora a ordem entre geometria base e listas não.

Figura 71: Exemplo de sequência da lista de linhas da fachada no código original. A linha 0 está posicionada na extremidade esquerda da fachada.

Isso acontece porque ao fornecer as linhas CAD como *input* no Grasshopper, a maneira como a ordem da lista de linhas é considerada pelo programa não é explícita. Além disso, outras características que não são explícitas ao usuário são consideradas pelo Grasshopper, como as direções que as linhas foram desenhadas no plano cartesiano (sentido positivo ou negativo dos eixos x, y e z). Foi necessária uma investigação nó a nó de todo o código e um tempo considerável para identificar que a ordem da lista é dada pela ordem de criação das linhas CAD.

Figura 72: Exemplo de sequência de lista de linhas da fachada deslocada (shift). A linha 0 não está posicionada na extremidade da fachada, conforme esperado pelo código VPL.

Esse é claramente um exemplo em que a dimensão **Dependências Ocultas** da VPL interfere no desenvolvimento do projeto, uma característica negativa da linguagem (Figura 73).

Um dos aspectos que ajudaram a solucionar o problema foi o bom desempenho que o Grasshopper possui na dimensão **Visibilidade** (Figura 73). É possível visualizar os trechos de código simultaneamente com sua representação no modelo tridimensional (Figura 72), aumentando a compreensão que o usuário possui do problema ao tornar mais visual a relação entre dados presentes nos *data trees* e seu posicionamento no modelo.

No entanto, identificamos um alto grau de **Operações Mentais Difíceis**. Foi necessário recorrer a muitos meios de notação secundária (anotações paralelas) para identificar e resolver o problema de inconsistências e relações ocultas encontradas na VPL.

Mesmo no caso de códigos Grasshopper considerados bem estruturados e desenvolvidos por arquitetos programadores experientes, como é o caso do código DSC do escritório Perkins+Will, esse tipo de inconsistência é comumente encontrado quando há tentativa de mudança topológica do modelo tridimensional.

(TPL)

Ao substituir a implementação VPL em TPL o problema foi resolvido, pois a relação entre a linha CAD (geometria) e a respectiva área de fachada relacionada é imediatamente declarada:

```
(define (groups)
(list (list "pg1 lines" (g1-type))
(list "pg2 lines" (g2-type))
(list "pg3 lines" (g3-type)))
```

Onde “*pg1 lines*” é o nome do layer onde estão as linhas CAD da fachada e (*g1-type*) é o número da área de fachada associada a esta linha.

A TPL mostrou bom desempenho na dimensão **Dependências Ocultas** (Figura 73) ao tornar explícita a relação entre linha CAD e as áreas da fachada (como visto no código acima). Isso acontece porque Rosetta apenas considera os vértices iniciais das linhas CAD dadas como base. A geração das linhas e sua ordenação foram declaradas pelo programador do código TPL e não estão a mercê de reordenações vindas da ação do usuário no programa Rhinoceros. Ou seja, eliminamos uma camada de interferência de outros softwares.

Figura 73: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico “Relação e ordenação entre geometria e listas”.

4.4.3 Comunicação com simuladores

O objetivo inicial da implementação era substituir todo o código escrito na VPL Grasshopper inteiramente na TPL Rosetta. No entanto, para que o uso do Grasshopper pudesse ser completamente substituído, era necessário estabelecer uma comunicação direta entre Rosetta e os simuladores de análise Radiance e EnergyPlus. Porém, a comunicação entre Rosetta e EnergyPlus está em processo de desenvolvimento pelo grupo de pesquisa ADA e ainda não foi finalizada.

Portanto, é necessária uma ferramenta que conecte o projeto aos programas de simulação ambiental. Optamos por manter os simuladores de conforto ambiental do DSC original Honeybee e Ladybug, *plugins* que permitem a conexão e transferência de dados entre Grasshopper e

Radiance, Daysim, EnergyPlus e OpenStudio.

A necessidade de comunicar o modelo paramétrico gerado por Rosetta com os simuladores dentro do Grasshopper foi positiva para testarmos aspectos de Extensibilidade (MEYER, 1997) do programa. Foi necessário adaptar os códigos para atender mudanças de especificação no fluxo de trabalho. Esta adaptação nos trouxe alguns desafios.

Rosetta cria o modelo tridimensional no Rhinoceros, em seguida o modelo é vinculado ao Grasshopper para que o *plugin* Honeybee e Ladybug efetuem as análises energéticas e de iluminação (conforme visto na Figura 63).

A geração de geometrias na TPL acontece de maneira iterativa e incremental, as funções necessárias para criação dos painéis os criam individualmente, um após o outro.

A transferência da geometria gerada pelo Rosetta se deu através do seguinte fluxo: Rosetta gera as geometrias em *layers* previamente especificadas no Rhinoceros, o código VPL importa para dentro do Grasshopper qualquer geometria que estiver vinculada a essas *layers* através do nó “*Geometry Pipeline*”.

O Grasshopper está classificado como uma VPL de modelagem de fluxo de dados (JANSSEN; STOUFFS, 2015), isso significa que quando um nó detecta uma mudança de dado no *input*, o fluxo de dados é recalculado a partir daquele nó. Isso gerou dois problemas:

(1) Sincronização entre a TPL e VPL:

Embora Rosetta gerasse toda a geometria do edifício automaticamente no Rhinoceros, os nós de importação no Grasshopper somente identificam novas geometrias geradas caso o usuário faça uma mudança manual no programa, ou seja, o Grasshopper somente atualiza seu fluxo de dados se houver um “clique” com o *mouse* no Rhinoceros.

Para resolver este problema, implementamos uma função para sincronizar o Grasshopper apenas no momento desejado. Adicionamos um nó na VPL que lê constantemente um arquivo “*xpto*” referenciado no computador. Na TPL, incluímos uma função que gera e atualiza esse arquivo sempre que o Rosetta termina de criar todo o edifício. Dessa maneira, criamos uma comunicação para que a TPL sinalize quando terminou seu trabalho, o Grasshopper lê através da modificação do arquivo que pode disparar novamente seu fluxo de dados e realiza a leitura da nova alternativa de projeto. Somente então as geometrias são analisadas pelos simuladores Honeybee e Ladybug.

Da mesma maneira, no sentido inverso, precisávamos sinalizar para a TPL o momento em que deveria reexecutar o código e gerar uma nova variação do edifício. Implementamos um novo trecho de código na VPL para detectar quando os *plugins* terminam a avaliação em Grasshopper. Assim que um novo resultado sai do *output* dos *plugins* de simulação, um valor numérico é fornecido como *input* no *plugin* Colibri, que exporta para uma planilha os parâmetros do modelo, resultados das simulações e imagens do projeto. A TPL fica na espera de uma atualização no arquivo da planilha. Quando a mesma recebe os dados e é atualizada, Rosetta limpa os *layers* do Rhinoceros e reinicia o processo. Este processo é reexecutado até atingir o número de iterações especificadas pelo usuário na TPL.

No trecho de código abaixo é possível visualizar esta ordem de execução:

```

(define (create-all-panels)
  (begin
    (display "Cleaning up previous shapes...")
    (clean-up)
    (display "Creating allowed attributes...")
    (create-allowed-attributes)
    (display "Creating facade panels...")
    (create-facade-panels)
    (display "Creating glazing panels...")
    (create-skylight-panels)
    (synchronize-grasshopper)
    (display "Finishing..."))
  (run-n-analysis 64 create-all-panels 101 "" ("out:ILL Within"
                                              "out:ILL Above"
                                              "out:ILL Below"
                                              "out:Cooling (kWh/m2)"
                                              "out:Heating (kWh/m2)"
                                              "results/data.csv"))

```

(2) Sincronia dentro da VPL entre os nós de importação e os simuladores HoneyBee e Ladybug.

Adicionamos um nó de importação (*Geometry Pipeline*) para cada *layer* das linhas bases do edifício. No entanto, verificamos que desta maneira, as simulações nos *plugins* HoneyBee eram reiniciadas concomitantemente, e não aguardavam a iteração do novo edifício.

A avaliação deste problema não foi trivial, uma vez que logicamente, o Grasshopper captava a mudança em todos os nós de importação simultaneamente. No entanto, o fluxo dos novos dados não parecia chegar ao nó HoneyBee simultaneamente, o que disparava o *plugin* inúmeras vezes.

Devido à natureza caixa-preta do funcionamento do Grasshopper, não pudemos investigar exatamente o que causava esta falta de sincronia. Aqui, a falta de mecanismos para avaliar o funcionamento e teste do código no Grasshopper possui relação com a dimensão **Avaliação Progressiva**, uma característica negativa da VPL (Figura 76).

O problema foi resolvido ao testarmos importar todos os *layers* de uma só vez em apenas um nó de importação, ao invés de vários. Um nó especializado em Python separou os *layers* específicos (*walls*, **panels**, *windows*, etc), uma vez que não há outros nós com a mesma funcionalidade pré disponíveis em Grasshopper. Ou seja, foi necessário recorrer a uma solução via *scripting* (TPL) (Figura 75).

Esses pontos nos ajudam a avaliar a **Viscosidade** do sistema em relação a sua Extensibilidade (MEYER, 1997).

Meyer (1997) aponta que a Simplicidade do Projeto e a Descentralização (autonomia dos módulos do código) são princípios importantes para melhorar a Extensibilidade, a adaptação do programa a novas mudanças de especificação.

Figura 75: Trecho de código adicionado ao Grasshopper para importar a geometria criada por Rosetta.

Identificamos que, de maneira geral, a metodologia DSC oferece uma estrutura positivamente modular. A separação do fluxo de trabalho em fases possibilitou que algumas das etapas pudessem ser substituídas. Na impossibilidade de conectar o Rosetta diretamente com os simuladores, pudemos reaproveitar parte do código para realizar as simulações. No entanto, esse aspecto positivo está na maneira que a metodologia foi estruturada. Os maiores desafios para realizar esta conexão corretamente recaíram na alta **Viscosidade** da VPL Grasshopper (Figura 76) e a grande necessidade de Retrabalho (AISH; HANNA, 2017) para adaptar as novas conexões do fluxo de trabalho às especificidades da VPL de fluxo de dados.

Além disso, para realizar qualquer tipo de conexão entre Grasshopper e outros programas, é necessário recorrer a *plugins* de terceiros (como HoneyBee, LadyBug e Colibri). Este fato cria uma camada adicional de especificidades nas estruturas de dados do código.

Figura 76: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico "Comunicação com simuladores".

4.4.4 Inputs numéricos na VPL

Alguns dos parâmetros variáveis pelo processo de otimização do projeto são os valores *R-Value*¹¹⁴, *U-Value*¹¹⁵ e *SHGC*¹¹⁶ (*Solar Heat Gain Coefficient*) utilizados na simulação do desempenho energético da edificação.

A equipe de projeto do escritório especificou que cada um desses parâmetros pode variar entre três valores, sendo eles.

R-Value: 5.0, 6.0, 7.0
U-Value: 0.65, 0.7, 0.8
SHGC: 0.3, 0.4, 0.5

No fluxo de trabalho VPL, os valores de todos os parâmetros do edifício são escritos dentro do próprio Grasshopper (Figura 77). O valor é atribuído como *input* nos nós que calculam a iluminância e eficiência energética.

Figura 77: Valores numéricos dos parâmetros no Grasshopper

Quando o mesmo fluxo de trabalho foi implementado utilizando a TPL, o Rosetta passou a gerar os valores dos parâmetros. No entanto, como foi necessário passar o projeto para dentro dos simuladores na VPL, verificamos que não há mecanismos disponíveis em Grasshopper para leitura de valores numéricos criados fora de sua interface. Para contornar essa limitação, foi necessário criar segmentos de retas em que as dimensões correspondiam aos valores permitidos em cada um dos atributos.

Valores possíveis: *R-Value* | *U-Value* | *SHGC*

```
(define r-values (list 5 6 7))  
(define u-values (list 0.65 0.7 0.8))  
(define shgc-values (list 0.3 0.4 0.5))
```

¹¹⁴ *R-Value* (ou valor R) é uma medida de resistência térmica usada para análises térmicas em edificações. O valor é a razão da diferença de temperaturas entre um isolante térmico e o fluxo de calor. Quanto maior o número, melhor a eficiência do isolamento da edificação. Valor R é o inverso do Valor U.

¹¹⁵ *U-Value* (valor U ou transmitância térmica) é a taxa de transferência de calor (em watts/ m²) dividida pela diferença de temperatura na estrutura.

¹¹⁶ O ganho solar é o aumento da energia térmica de um espaço que absorve a radiação solar incidente. A quantidade de ganho solar que um espaço experimenta é uma função da irradiância solar incidente total e da capacidade de qualquer material interveniente para transmitir ou resistir à radiação.

Criação de linhas para que a VPL possa utilizar os valores *R-Value* *U-Value* e SHGC

```
(define (create-allowed-attributes)
  (with-current-layer (create-layer "R-Value")
    (begin
      (line (u0) (x (r-value)))
      (text (format "r-value:~a" (r-value)) (x (r-value)) 1)))
  (with-current-layer (create-layer "U-Value")
    (line (y 1) (xy (u-value) 1))
    (text (format "u-value:~a" (u-value)) (xy (u-value) 1) 1))
  (with-current-layer (create-layer "SHGC")
    (line (y 2) (xy (shgc-value) 2))
    (text (format "SHGC:~a" (shgc-value)) (xy (shgc-value) 2) 1)))
```

Desta maneira, as linhas foram criadas por Rosetta e importadas como geometria para dentro do Grasshopper. Adicionamos um trecho de código em Grasshopper que lê a dimensão das linhas e as converte em valores numéricos para que as simulações possam as considerar como inputs válidos (Figura 78).

Este é outro exemplo onde a limitação da VPL cria problemas de **Viscosidade**, sua resistência à mudança é uma característica negativa (Figura 79). Contornar o problema exigiu a criação do trecho de código TPL (visto acima), uma vez que a flexibilidade da TPL permite criar linhas e associar esses valores numéricos a ela, característica da baixa **Viscosidade** da linguagem (Figura 79). Isso também gerou a adição de trechos e edições na VPL, ou seja, foi necessário trabalho adicional em ambas as linguagens para contornar a limitação do sistema VPL.

Figura 78: À esquerda, linhas criadas pelo Rosetta para os valores SHGC, U-Value e R-Value. À direita, trecho de código em Grasshopper que transforma a dimensão da linha em valor numérico.

Figura 79: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico "Inputs numéricos na VPL".

4.4.5 Escalabilidade do código e modelo

A implementação em TPL mostrou avanços em relação à **Viscosidade**, Escalabilidade (KIPER; HOWARD; AMES, 1997) e Extensibilidade (MEYER, 1997) em relação à VPL (Figura 82).

O Rosetta é altamente adaptado à escalabilidade do projeto em dois sentidos:

(1) propriedades escaláveis inerentes às TPLs, uma vez que é possível atingir níveis mais sofisticados de iteração, recursividade, estrutura de dados e menos limitação de softwares específicos (conforme discutido na seção 2.4 Linguagens de Programação Textual (TPLs))

(2) propriedades escaláveis inerentes a portabilidade do modelo:

Uma das principais características do Rosetta é a portabilidade (LOPES; LEITÃO, 2011). Essa característica permite explorar facilmente o modelo desenvolvido no *front-end* (linguagem de programação escolhida para desenvolvimento) e *back-ends* diferentes (programas de modelagem generativa CAD, BIM e de análises de desempenho). Isso significa que a partir do mesmo código é possível extrair modelos tridimensionais adaptados à cada fase de projeto.

Dessa maneira foi possível, no mesmo código TPL, criar os elementos do edifício com propriedades BIM adaptadas às fases de um modelo conceitual de exploração geométrica, ao mesmo tempo que as superfícies necessárias para análises poderiam ser extraídas (Figura 80). Os painéis foram modelados, atrelados a famílias BIM, e também possuem atributos específicos para a análise ambiental. Fora deste ambiente, seria necessário adaptar o modelo para cada saída de software específico. Desta maneira, é possível escalar do modelo conceitual e analítico ao modelo com construtibilidade e detalhes necessários às fases mais avançadas do projeto arquitetônico.

Figura 80: Modelos analíticos adaptados a demandas específicas de simulação.

A portabilidade e capacidade de reutilização do código é um fator-chave para um modelo paramétrico em arquitetura, uma vez que permite a grande variabilidade da concepção e adaptações no projeto sem necessidade de quebrar a interoperabilidade do modelo gerado algoritmicamente (Figura 81). Estes fatores influenciam fortemente a produtividade num fluxo de trabalho algorítmico.

Figura 81: Exemplo de necessidade de quebra de parametria de um modelo para exportá-lo de CAD para BIM. Fonte: adaptado de Janssen, Stouffs (2015)

Janssen (2015) abordam o problema de portabilidade ao discutir o acoplamento entre sistemas paramétricos e BIM. Por exemplo, o Dynamo (modelagem processual) está fortemente acoplado ao BIM por se tratar de uma linguagem VPL utilizada especificamente em Revit. Os programas escritos em Dynamo não são portáveis para outros softwares BIM ou CAD e o trabalho dedicado pelo usuário na criação de um programa apenas funciona nesta VPL específica. Este é um caso de uma abordagem fortemente acoplada (JANSSEN, 2015).

Caso o usuário queira exportar um modelo construído em outros tipos de sistemas paramétricos, como em Grasshopper (modelagem de fluxo de dados), para BIM (modelagem associativa), pode utilizar abordagens livremente acopladas (JANSSEN, 2015) que possibilitam a exportação do modelo. Por exemplo, o plugin para Grasshopper GeometryGym produz um arquivo IFC do modelo paramétrico. O IFC é uma extensão de arquivo que permite que o modelo contenha metadados atrelados às geometrias e possibilita a troca de informações entre programas BIM diferentes.

No entanto, mesmo na abordagem livremente acoplada, toda a inteligência paramétrica do modelo é perdida, pois no processo de exportação da geometria, os tipos de modelagem paramétricas mais avançadas (processual e fluxo de dados) são convertidos em modelagem explícita (Figura 81).

Conforme descrevemos acima, Rosetta resolve este problema, pois possui abordagem livremente acoplada, ou seja, permite que o mesmo código TPL possa criar modelos CAD, BIM, analíticos, ou para qualquer outro formato solicitado pelo software conectado no *back-end*.

Figura 82: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico “Escalabilidade do código e modelo”.

4.4.6 Confiabilidade de resultados

A confiabilidade do código, ou seja, a verificação se trechos ou funções do algoritmo que funcionam de acordo com o planejado é parte importante de um projeto, principalmente quando o código tende a se tornar complexo ou com muitas dependências.

Alguns problemas que interferem na confiabilidade durante o desenvolvimento em VPL Grasshopper foram encontrados. Nos casos a seguir, nenhum deles acusou uma exceção (erro), mas apresentaram mau funcionamento:

(1) Zonas térmicas:

As zonas térmicas do projeto são áreas (Figura 83) que possuem atributos específicos para o cálculo energético. Para que elas possam ser fornecidas como input para o cálculo de zonas adiabáticas (uma zona que está isolada de quaisquer trocas de calor), é necessário informar a posição (índice) de cada uma dessas zonas em uma lista. No entanto, mesmo tratando de um *input* estático (pois as zonas são fixas), o Grasshopper reorganiza a ordem da lista das geometrias das zonas conforme o arquivo é reaberto, intermitentemente, hora a lista é reorganizada, hora não. Desta maneira, os índices atribuídos às zonas de adjacências sofrem mudanças de posição sob uma lógica caixa-preta, que não é clara ao usuário da VPL, ou seja, não é possível identificar o motivo sem consultar o desenvolvedor do programa.

Figura 83: Exemplo de uma zona térmica do edifício.

O problema interfere drasticamente nos resultados das análises energéticas, uma vez que as zonas de adjacência consideradas nas análises podem estar erradas. Esse problema pode facilmente passar despercebido pelo usuário, pois é comum manter a pré-visualização desligada de geometrias sem relevância primária para o projeto.

Para resolver o problema, criamos um novo trecho de código que avalia quais as zonas eram adiabáticas, independente da ordem em que aparecem na lista de geometrias. O problema estava em estabelecer uma regra apropriada para essa identificação automática, uma vez que existe uma grande uniformidade entre as geometrias das zonas, ou seja, pouca variação entre elas por se tratar de uma planta simétrica. Por se tratar de uma resolução pontual, existem muitas chances que este trecho de código seja perdido caso o projeto mude de forma (mudança de topologia), uma vez que se trata de uma resolução um domínio de problema específico.

(2) Versão de software:

Outro problema encontrado foi um bug em um componente específico do *plugin* Honeybee quando o mesmo código é aberto nas versões 5 e 6 do Rhinoceros.

O componente que converte uma superfície a um objeto pronto para ser lido como vidro pelo simulador energético apenas funciona na versão 5 do programa. Além disso, existem quatro componentes iguais no mesmo trecho do código, apenas um deles não funciona apropriadamente. O problema foi identificado ao realizarmos uma verificação manual do funcionamento da simulação. Nenhuma exceção é levantada pelo Grasshopper para acusar este problema, pois a lista possui os tipos de dados esperados pelos nós seguintes, independente se estão corretos ou não.

A resolução para esse problema foi manter o desenvolvimento na versão 5 do programa.

(3) Erro no separador de decimal dos valores da simulação energética:

Durante a implementação do estudo de aplicação identificamos um erro no separador de decimais dos resultados das análises energéticas. Os erros aconteciam durante a exportação desses valores entre a VPL e a planilha. Por exemplo, o valor 17.3333 calculado pela análise energética era coerente dentro do Grasshopper, porém, logo após a gravação deste dado em uma planilha (realizado pelo *plugin* Colibri), o número aparecia com a separação da casa decimal deslocado à esquerda 1.73333 (Figura 84). O problema é intermitente, só acontece algumas vezes e não possui padrão aparente que ajude na resolução do problema, hora aparecia nos resultados das simulações *heating*, hora no *cooling*.

Figura 84: Valor com diferença no separador decimal entre a planilha (esquerda) e o resultado da simulação em Grasshopper (direita).

Não pudemos identificar a exata causa do problema. Embora a redução do espaço de possibilidades do causador do erro indicasse-nos o *plugin* Colibri, o problema não existia nas versões anteriores do projeto. Além disso, trata-se de um *plugin closed-source*, impossibilitando a investigação do problema.

Isso afeta diretamente a confiabilidade dos resultados. Caso qualquer um destes erros passe despercebido, o projeto pode não estar verdadeiramente otimizado/eficiente. Verificamos aqui que a VPL Grasshopper, mesmo com um código em funcionamento, sem nenhum problema explícito na declaração do algoritmo, possui alta **Propensão a Erros**. A maioria deles de fontes obscuras e de repetição intermitente. A falta de mecanismos para checar o funcionamento do código progressivamente é um ponto negativo da linguagem (Figura 85).

Esse encadeamento do uso de *plugins* e interfaces de programação de fontes diferentes também pode aumentar as chances de ocorrerem *bugs* ou mau funcionamento em alguma etapa do fluxo de trabalho. Por se tratarem de programas caixa-preta, o usuário não possui meios para resolver os problemas diretamente, tendendo a adicionar mais complexidade no código e Retrabalho (AISH; HANNA, 2017) para contornar os problemas.

A utilização de uma TPL proporcionou abertura suficiente para que o usuário tenha mais controle do código, além de possuir mais mecanismos para investigar onde o erro ocorre (*unit tests*, por exemplo), melhorando significativamente a **Avaliação Progressiva** (Figura 85). Além disso, ao trabalhar em Racket, a mesma linguagem utilizada na criação das bibliotecas do Rosetta, o usuário possui conhecimento da linguagem e autonomia para investigar e propor resoluções diretamente nos códigos da biblioteca quando detecta um *bug* ou mau funcionamento.

Dessa maneira, ao abrir as caixas-pretas, o uso de TPLs melhora a interação do usuário com o código, dando maior controle e aumentando a confiabilidade geral do projeto.

Porém, é interessante observar que Green e Petre (1996) afirmam que VPLs têm menos **Propensão a Erros** do que TPLs, uma vez que é mais frequente que o programador cometa erros no uso de delimitadores e separadores durante a escrita de uma TPL (como parênteses, chaves e *ends*) do que durante a explicitação das dependências do fluxo de dados conectando nós e fios em uma VPL.

Durante o desenvolvimento de algoritmos em VPL é possível perceber que esta afirmação está correta, uma vez que ao realizar a conexão entre os nós, caso algum dado não mantenha a especificação desejada com o nós subsequentes, uma exceção é levantada imediatamente no tempo de execução. Em contraposição, algumas linguagens TPL retornarão o erro em tempo de compilação.

No entanto, conforme o código em VPL se torna mais complexo, embora seja possível visualizar qual dos nós levantou exceção, torna-se mais difícil e/ou mais demorado mapear a causa do erro, como no caso do relato desta implementação.

Figura 84: Valor com diferença no separador decimal entre a planilha (esquerda) e o resultado da simulação em Grasshopper (direita).

4.4.7 Centralização e controle de dados

O estudo de aplicação também exemplifica como a grande **Expressividade** encontrada no uso das VPLs Grasshopper e Planilhas ajudou no fluxo de trabalho do estudo de aplicação, um ponto positivo da linguagem (Figura 87).

No estudo de aplicação, uma planilha centraliza informações importantes para a equipe de projeto. Por exemplo, a planilha “*ZoneInput*” contém os dados que serão considerados como inputs no cálculo das zonas térmicas (Figura 85).

Zone #	Program #	number of People	LightingDensity (W/m²)	equipmentLoadPerArea (W/m²)	CoolingSetPoint (°C)	HeatingSetPoint (°C)	CoolingSetBack (°C)	HeatingSetBack (°C)	Heat Recovery Efficiency (%)	Coefficient of Heating Performance (COP)	Coefficient of Cooling Performance (COP)	ventilationPerArea (m³/s/m²)	ventilationPerPerson (m³/s per person)
4	9	0	5,381595	3,982	24	20	27	16	70	1	1	0	0
7	10	0	5,381595	3,982	24	20	27	16	70	1	1	0,0000235	0
13	9	0	5,381595	3,982	24	20	27	16	70	1	1	0	0
14	7	50	5,381595	14,961	24	20	27	16	70	1	1	0	0,00705
15	9	0	5,381595	3,982	24	20	27	16	70	1	1	0,0000235	0
16	7	50	5,381595	14,998	24	20	27	16	70	1	1	0	0,00705
17	9	0	5,381595	3,982	24	20	27	16	70	1	1	0,0000235	0
18	0	0	5,381595	10,010	24	20	27	16	70	1	1	0	0
19	10	0	5,381595	3,982	24	20	27	16	70	1	1	0,0000235	0
20	0	0	5,381595	10,010	24	20	27	16	70	1	1	0	0
21	9	0	5,381595	3,982	24	20	27	16	70	1	1	0,0000235	0
22	7	50	5,381595	14,998	24	20	27	16	70	1	1	0	0,00705
23	9	0	5,381595	3,982	24	20	27	16	70	1	1	0,0000235	0
24	7	50	5,381595	14,961	24	20	27	16	70	1	1	0	0,00705

Figura 85: Dados na planilha *ZoneInput*, necessários para realizar a simulação de eficiência energética do edifício.

A planilha “*PannelInput*” centraliza todos as variáveis paramétricas do projeto que foram atribuídas na geração dos indivíduos pela VPL (Figura 86). Desta maneira, a equipe de projeto consegue visualizar de maneira simples quais parâmetros foram atribuídos à cada alternativa do edifício gerada randomicamente pelo código VPL.

RUN #	PG 1	PG 2	PG 3	PG 6	PG 7	PG 8	PG 9	PG 10	PG 11	PG 14	PG 16	R-Value	U-Value	SHGC	Shading S	Shading E-W	Shading N
1	2	2	1	3,000	5	3	4	6	5	5	1	7	0,8	0,3	0,5	0	0,5
2	2	2	1	3,000	6	5	2	3	6	7	1	6	0,8	0,4	0,5	0,5	0,05
3	1	1	1	3,000	6	2	3	3	6	7	1	5	0,7	0,4	1	0,5	1
4	1	0	0	1,000	6	6	2	3	5	5	1	6	0,65	0,4	0,5	1	0,5
5	4	3	1	0,000	6	3	2	3	5	5	1	6	0,8	0,5	0,5	0,5	0,5
6	6	6	1	3,000	4	6	3	2	7	5	1	5	0,7	0,5	1	0,5	0,05
7	4	3	1	1,000	5	6	2	3	6	5	1	5	0,7	0,3	0,5	1	0,5
8	2	6	1	0,000	5	3	3	5	6	5	1	6	0,8	0,4	1	0	1
9	1	1	0	1,000	6	4	5	5	6	7	1	5	0,8	0,5	0	0,5	0,5
10	2	2	0	1,000	5	2	2	2	6	5	0	6	0,8	0,4	1	1	0,05
11	1	0	0	1,000	6	4	2	3	6	5	1	6	0,8	0,3	1	0,5	1
12	4	2	0	0,000	4	3	5	5	5	5	0	5	0,65	0,3	0	1	1
13	1	2	0	0,000	5	3	6	3	6	5	0	6	0,8	0,3	0	1	0,5
14	1	1	0	2,000	4	4	3	4	7	5	1	5	0,8	0,3	0,5	0	0,5
15	2	2	0	1,000	5	2	2	2	6	5	0	6	0,8	0,4	1	1	0,05

Figura 86: Dados na planilha PanellInput, com a escolha randômica dos parâmetros do edifício em cada alternativa de projeto.

Os principais benefícios deste fluxo de trabalho são: (1) a equipe de projetistas que não está familiarizada com a programação em VPL ou a manipulação do código possui uma linguagem familiar para gravar, modificar e controlar os *inputs* das análises de conforto térmico e dos parâmetros que são permitidos variar no projeto de arquitetura; (2) É possível rodar as simulações e análises de conforto térmico em máquinas paralelas, obter os dados das análises separadamente e depois mesclá-los todos novamente na mesma planilha (por exemplo, uma máquina pode rodar as análises 1 a 20, outra de 21 a 40, e assim sucessivamente).

Esse é um benefício que está mais atrelado à maneira pela qual os projetistas do código DSC estruturaram a interação entre as etapas do fluxo de trabalho. Tanto Grasshopper (através do *plugin* Colibri) como Rosetta desempenham bem a conexão com a planilha.

Já no caso da TPL, é necessário possuir muito mais controle dos parâmetros e combinações de projetos que estão sendo declarados no código. A falta de **Expressividade** da linguagem os deixa sem sinalização, um ponto negativo da TPL (Figura 87). Essas informações podem não ser totalmente vistas em uma linguagem unidimensional em texto.

Figura 87: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico "Centralização e controle de dados".

4.4.8 Organização e clareza do código

A Notação Secundária reúne muitos aspectos importantes sobre a usabilidade de sistemas paramétricos. É uma dimensão impactante para que usuários sem educação formal em computação possam compreender programação. As características de **Notação Secundária** presentes em ambientes como Grasshopper ajudaram a compor a popularidade do uso de VPLs em arquitetura e são um ponto positivo das linguagens VPL (Figura 89).

Na VPL, a relação geral entre os módulos é diretamente visível dada a representação bidimensional da programação por fluxos de dados. Os desenvolvedores do código DSC providenciaram instruções de operação do código através de ferramentas de textos e sequências de sinalizações; organização e indicação de módulos, no qual os trechos de códigos foram separados em grupos e sinalizados com cores relativas às suas funções (azul para o código referente a construção da modelagem paramétrica, laranja para análise de iluminação e verde para análise energética); e os resultados das simulações puderam ser agrupados e visualizados de maneira clara, devido à liberdade na organização do *layout* da VPL (Figura 88).

Figura 88: Organização e informação do código VPL criada através dos mecanismos de Notação Secundária disponíveis em Grasshopper.

Já na implementação da TPL, devido à sua representação unidimensional¹¹⁷, a leitura e compreensão do código são menos explícitas e dependem mais do nível de experiência do usuário com a sintaxe e semântica da linguagem, uma vez que a disponibilidade de mecanismos de **Notação Secundária** é drasticamente reduzida, um aspecto negativo da linguagem (Figura 89).

¹¹⁷ TPLs são consideradas unidimensionais porque o compilador ou interpretador as processa como um fluxo longo e unidimensional (Kiper, 1997)

A IDE Dr.Racket oferece um mecanismo de **Notação Secundária** que sinaliza e ajuda que o usuário encontre no texto onde estão as dependências entre funções já declaradas. No entanto, isto compreende um espectro muito limitado no entendimento que o usuário precisa construir entre o código e os resultados do programa (o modelo tridimensional).

Ao compararmos o mesmo estudo de aplicação implementado em ambas linguagens VPL e TPL, foi possível visualizar que muitos dos aspectos de legibilidade dos códigos não estavam somente atrelados à maneira como os programadores originais estabeleceram sua estrutura conceitual, mas à maior expressividade das VPLs e dos mecanismos de texto, agrupamentos, cores, setorizações, clusters e manipulação dinâmica do layout disponibilizados pelo Grasshopper. Ou seja, a Expressividade na VPL Grasshopper possibilitou descobrir com mais facilidade porque o programador definiu a estrutura do código de tal maneira.

Em TPL, os mecanismos de organização e documentação disponíveis também impactaram negativamente no nível de **Operações Mentais Difíceis, Visibilidade e Expressividade** geral da linguagem (Figura 89). Para que o usuário entenda claramente o funcionamento de todos os módulos e arquivos implementados na TPL, caso não tenha sido o principal desenvolvedor do código, é provável que seja necessário desenhar as relações entre as partes para ajudar a entender o fluxo de trabalho.

Figura 89: Avaliação dos desempenhos positivos ou negativos das linguagens nas CDs de acordo com os achados no tópico “Organização e clareza do código”.

4.4.9 Manipulação de interface

Identificamos que no processo de iteração do código Grasshopper, durante a geração das variações do edifício, a interface do Grasshopper e Rhinoceros permanecem travadas, impossibilitando qualquer interação do usuário enquanto o programa está em execução.

Além disso, após o início da iteração do código (usando o *plugin* Colibri, uma vez que o Grasshopper não suporta recursão sem o uso de *plugins* de

terceiros), não existem maneiras de interromper o processo de iteração e simulação iniciados. O usuário precisa encerrar manualmente as janelas de *prompt* do simulador que aparecem na tela. Esta é uma tarefa inviável quando o número de iterações é grande (64 iterações, neste estudo de aplicação). Portanto, é necessário finalizar a execução do Rhinoceros forçadamente através do Gerenciador de Tarefas do sistema operacional, ocasionando perda do estado atual do código, caso o mesmo não tenha sido salvo.

Já a TPL Racket fornece a possibilidade de interromper a compilação do código imediatamente quando o usuário deseja encerrar ou pausar o processo. Além disso, a utilização da IDE Dr. Racket não trava a interface do programa Rhinoceros durante a execução do código.

A Interatividade dos sistemas é abordado por Aish e Hanna (2017) e possui relação com o desempenho de visualização do programa como um todo e de suas características de latência. Meyer (1997) também aborda características de Eficiência dos programas, conforme discutido na seção 3.2.16 Outros paralelos.

Essas características não possuem paralelos com as Dimensões Cognitivas de Green e Petre (1996). No entanto, oferecem vantagens que afetam drasticamente o uso destes sistemas. Neste estudo de aplicação, o desempenho da TPL nos aspectos de interatividade foi positivo, pois permitiu que o usuário manipule o programa e a visualização do modelo geométrico ao mesmo tempo que a geometria é gerada na tela (*zoom*, *pan*, *orbit*, etc), além da oferecer a capacidade de interromper o processo sem maiores impedimentos.

4.4.10 Modularização e versionamento

Outras duas características se destacaram durante a implementação do estudo de aplicação, a capacidade de modularizar o código e o versionamento dos arquivos, são aspectos que afetam fortemente o trabalho colaborativo.

Devido as características da programação VPL, o código escrito em Grasshopper, por exemplo, possui um arquivo único e de extensão própria (.gh). Isso significa que apenas um usuário pode editar o arquivo por vez. Além disso, ainda não existem mecanismos que mesclam novas alterações em códigos VPLs diferentes, como acontece nos Sistemas de Controle de Versão (*Version Control System* ou VCS) distribuídos utilizados nas TPLs. O versionamento VCS documenta as alterações produzidas nos arquivos de código e permite que o usuário recupere versões específicas. Além disso, o sistema (como *GitHub*, por exemplo) identifica e compara modificações realizadas, rastreando alterações feitas no arquivo.

Sistemas de Controle de Versão estão divididos em centralizados e distribuídos. Nos sistemas centralizados, os usuários trabalham em cópias de arquivos que estão hospedados em um servidor central. Na arquitetura, esse tipo de versionamento é comum em programas BIM, que permite o trabalho colaborativo em um mesmo arquivo, porém, centraliza as permissões de alterações no arquivo. Nos sistemas distribuídos, os repositórios são autônomos e independentes, um para cada colaborador. Trabalhar em sistemas distribuídos permite a colaboração entre diferentes usuários simultaneamente, pois as alterações nos arquivos são analisadas e mescladas pelo sistema.

Durante a implementação TPL, foi possível utilizar técnicas da programação modular, ao dividir e organizar o código com base na tarefa que é executada. Por exemplo, existe um código que

cria o edifício base, com todas as partes do projeto que não se alteram, um código que cria todas as partes variáveis do edifício, outro que executa a rotina das análises de simulação e controla as iterações do edifício, entre outros.

Figura 90: Diagrama de conexão entre os módulos e arquivos necessários para executar o programa em TPL.

Alguns dos benefícios das técnicas de programação modular facilitam, por exemplo, a depuração do código, ou seja, é mais fácil verificar se há algum erro e encontrá-lo em uma função específica. Também auxilia na legibilidade do fluxo de trabalho como um todo, pois é possível organizar e separar as funcionalidades, em vez de tentar ler um código que executa diversos tipos de tarefas em um único arquivo. Além disso, nos casos que os códigos foram escritos com níveis de abstração suficiente, as soluções dos módulos podem ser reutilizadas sempre que o usuário precisar executar a tarefa novamente em outro projeto.

Técnicas de modularização também foram aplicadas pelo escritório Perkins+Will na criação do DSC em Grasshopper, como a separação das funções do código por grupos de cores (Figura 91). No entanto trata-se mais de um meio de utilizar os mecanismos de Notação Secundária disponível em Grasshopper a favor da organização e legibilidade do código. O usuário ainda não pode se beneficiar das características de modularização oferecidas pelas TPL, como a separação em arquivos diferentes (uma vez que não há maneiras totalmente efetivas de comunicar arquivos diferentes em Grasshopper) e nem utilizar o sistema de versionamento distribuído.

Figura 91: Código DSC em Grasshopper separado por módulos.

Com exceção da dimensão Abstração, que interfere na generalização como o código pode ser escrito para então ser modularizado, nenhuma das Dimensões Cognitivas parecem estar diretamente atreladas às características discutidas nesta seção.

4.5 Comparação entre VPL x TPL

Os aspectos discutidos ao longo desta dissertação, apontam para as limitações do uso das VPLs em arquitetura. De maneira geral, as discussões indicam que o uso de TPL possui algumas vantagens de utilização, principalmente em Projetos Orientado ao Desempenho. Isso acontece principalmente porque as TPLs facilitam sua aplicação a problemas maiores ou mais complexos (NOONE; MOONEY, 2018).

As análises da implementação do estudo de aplicação neste capítulo apontam para o mesmo comportamento. Reunimos na Figura 92 a visão geral do desempenho positivo e negativo da VPL Grasshopper e da TPL Racket/Rosetta conforme comentamos nas seções anteriores.

Figura 92: Panorama geral do desempenho da VPL Grasshopper e TPL Racket/Rosetta nos tópicos abordados.

Embora a compensação entre os aspectos positivos e negativos das linguagens pareçam balanceados quando olhamos a Figura 92, a investigação empírica nos mostrou que as dimensões em que a TPL se desempenhou melhor, influenciam muito mais na usabilidade e confiabilidade geral do fluxo de trabalho.

A **VPL Grasshopper** obteve melhor desempenho nas Dimensões Cognitivas **Expressividade**, **Notação Secundária** e **Visibilidade**.

A **TPL** se desempenhou melhor nas Dimensões Cognitivas **Viscosidade**, **Abstração**, **Dependências Ocultas**, **Propensão a Erros** e **Avaliação Progressiva**.

Ambas as linguagens mostraram evidências de mau desempenho em **Operações Mentais Díficeis**. Além disso, não concluímos sobre o desempenho da VPL Grasshopper na dimensão **Abstração**. Comentaremos

estes aspectos com mais detalhes na próxima seção.

Adicionalmente, traduzimos o desempenho das linguagens em outro formato de tabela a fim de compreender os mesmos dados sob uma perspectiva paralela (Figura 93). A intenção aqui é observar o desempenho geral nos estudos de caso considerando também os aspectos que classificamos como neutros. Ressaltamos que por se tratar de uma investigação empírica, a contagem dos pontos positivos, negativos e neutros não tem a intenção de gerar dados quantitativos de conclusão. Para isso, estudos específicos orientados a quantificar estatisticamente o desempenho VPL vs. TPL são necessários. Nosso objetivo é compreender qualitativamente o conjunto de resultados. Além disso, a maioria das dimensões relatadas na análise apareceram como ponto negativo em uma linguagem e respectivamente positiva na outra. Isso reforça que um dos tipos de PL ajudou a melhorar seu desempenho na Dimensão Cognitiva analisada.

	VPL			TPL		
	positivo	negativo	neutro	positivo	negativo	neutro
1 Viscosidade		X		X		
2 Abstração			X	X		
3 Comprometimento Prematuro			X			X
4 Proximidade de Mapeamento			X			X
5 Dependências Ocultas		X		X		
6 Expressividade	X				X	
7 Notação Secundária	X				X	
8 Visibilidade	X				X	
9 Operações Mentais Díficeis		X			X	
10 Consistência			X			X
11 Difusão/ Concisão			X			X
12 Propensão a Erros		X		X		
13 Avaliação Progressiva		X		X		
Total	3	5	5	5	4	4

Figura 93: Desempenho da VPL Grasshopper e TPL Racket/Rosetta nas Dimensões Cognitivas.

As evidências indicam que é mais interessante neste fluxo de trabalho utilizar TPLs e melhorar sua usabilidade nas dimensões que possuem interferência visual, como Notação Secundária, Expressividade e Visibilidade.

4.5.2 Desempenho positivo da TPL Racket/Rosetta

No geral, a TPL Racket e Rosetta obtiveram melhor desempenho nas dimensões Viscosidade, Abstração, Dependências Ocultas, Propensão a Erros e Avaliação Progressiva.

Viscosidade:

Esta dimensão se configurou como uma das características mais relevantes da implementação TPL. A dimensão foi identificada como ponto positivo nas tarefas 4.4.1 Mudança na topologia do projeto, 4.4.3 Comunicação com simuladores, 4.4.4 Inputs numéricos na VPL e 4.4.5 Escalabilidade do código e modelo, ou seja, a TPL analisada oferece baixa Viscosidade.

Conforme vimos, esta é uma característica de sistemas que contém muitas dependências entre suas partes, ocasionando que uma pequena alteração

necessite de muitos ajustes subsequentes (GREEN, 1989).

Os aspectos presentes nesta dimensão são protagonistas nas discussões em Design Computacional. Este é basicamente o problema de pesquisa discutido na tese de doutoramento de Daniel Davis (2013) (Capítulos 1.1 Problems with Flexibility e 1.2 Flexibility of Code) e largamente citado por pesquisadores ao discutirem a prática da modelagem paramétrica em VPLs (GERBER, 2007, p. 205; SMITH 2007, p. 2; HOLZER; HOUGH; BURRY, 2007, p. 639; AISH; WOODBURY, 2005, p. 151; BURRY, 1996, p. 78, apud DAVIS, 2013).

Embora as análises do estudo de aplicação tenham mostrado evidências empíricas de que o uso da TPL ajudou a resolver os problemas de flexibilidade e escalabilidade encontrados na VPL, ainda existem outros apontamentos que podem avançar ainda mais esta discussão:

TPLs podem ter menos resistência à mudança se disponibilizarem mais mecanismos de abstrações adequadas ao domínio das resoluções de problema (GREEN; PETRE, 1996, BLACKWELL et al., 2001), uma vez que abstrações bem escolhidas aumentam a compreensão da linguagem (GREEN; PETRE, 1996, SEBESTA, 2011);

Tanto Meyer (1997) quanto Davis (2013) apontam que as técnicas utilizadas na Engenharia de Software, como Programação Estruturada e Programação Orientada a Objetos, podem ajudar a resolver os problemas de Viscosidade. TPLs possuem flexibilidade suficiente para serem utilizadas para resolver estes problemas. O método de Programação Orientada a Objetos melhora muitos problemas de extensibilidade ao ajudar projetistas a produzir sistemas cuja estrutura permanece descentralizada e simples, mesmo para sistemas grandes (MEYER, 1997).

Abstração:

A dimensão Abstração, a habilidade em definir aspectos gerais de um contexto qualquer e ignorar características menos importante para a generalidade deste contexto, foi discutida na tarefa 4.4.1 Mudança na topologia do projeto. Ambos, Grasshopper e Racket/Rosetta, possuem mecanismos de abstração.

Nossas investigações nos estudos de caso não foram suficientes para aprofundar a análise desta dimensão, uma vez que não implementamos nenhuma ação direta para resolver os problemas através de mecanismos de abstração disponíveis nas linguagens.

No entanto, baseado nas discussões das CDs, as TPLs Racket e Rosetta pareceram mais apropriadas em Abstrações por permitirem o uso de paradigmas de Programação Funcional (que incluem técnicas de metaprogramação) e paradigma de Programação Orientada a Objetos (que incluem técnicas de abstração mais avançadas).

Através da discussão realizada na revisão de literatura (seção 2.5 Cognição em programação), podemos concluir que evitar que abstrações se tornem uma barreira (AISH; HANNA, 2017) e balancear os mecanismos de abstração (GREEN; PETRE, 1996) é necessário, pois:

(1) a quantidade de abstrações disponíveis na PL interfere na simplicidade geral da linguagem e sua legibilidade (SEBESTA, 2011);

(2) utilizar abstrações bem escolhidas pode aumentar a compreensão de uma linguagem (GREEN; PETRE, 1996; KIPER; HOWARD; AMES, 1996), sua facilidade de uso (MEYER, 1997) e sua facilidade de escrita (SEBESTA, 2011).

Dependências Ocultas:

A dimensão Dependências Ocultas de Racket e Rosetta, discutida nas tarefas 4.4.1 Mudança na topologia do projeto e 4.4.2 Relação e ordenação entre geometria e listas, mostrou que o uso de TPL diminui ou resolveu os problemas de dependências ocultas encontradas na VPL Grasshopper.

A TPL teve melhor desempenho nesta dimensão ao tornar explícitas as relações entre entidades ou trechos de código.

Propensão a Erros:

A dimensão Propensão a Erros de Racket e Rosetta, discutida na tarefa 4.4.6 Confiabilidade de resultados, fala sobre os erros de sintaxe da linguagem.

Foi possível verificar que VPLs têm menos Propensão a Erros do que TPLs (GREEN; PETRE, 1996), que evitar erros de sintaxe é quase impossível em linguagens baseadas em texto (KAUREL, 2016) e que Racket possui de fato um número alto de delimitadores, como a quantidade de parênteses necessários serem declarados. No entanto, classificamos o desempenho como positivo nesta dimensão, porque a TPL proporciona identificar e corrigir erros mais facilmente do que a VPL. Adicionalmente, muitas IDEs possuem um corretor automático para erros de digitação do usuário.

Avaliação Progressiva:

A dimensão Avaliação Progressiva de Racket e Rosetta, discutida na tarefa 4.4.6 Confiabilidade de resultados, observa se a linguagem oferece mecanismos de avaliação gradual no desenvolvimento do código.

Esta foi uma das características mais positivas do desenvolvimento TPL para o fluxo de trabalho analisado. O código textual possui mais mecanismos de controle e verificação. Por exemplo, o uso da TPL aumenta a confiabilidade da etapa de otimização por permitir verificar se os resultados das simulações são válidos e se os domínios de resultados estão dentro do esperado.

Ao trabalhar em Racket também é possível investigar e propor resoluções diretamente nos códigos da biblioteca Rosetta quando detecta-se um bug ou mal funcionamento.

Quanto à dimensão Comprometimento Prematuro, embora possua características muito relacionadas na literatura dos sistemas de design paramétrico, não encontramos aspectos relevantes durante a implementação do estudo de aplicação suficientes para aprofundar a discussão e comparação desta dimensão entre as interfaces analisadas.

Da mesma maneira, o estudo de aplicação implementado não foi suficiente para investigar a dimensão Proximidade de Mapeamento e Difusão/Concisão mais a fundo.

4.5.1 Desempenho positivo da VPL Grasshopper (aspectos a melhorar na TPL)

A VPL Grasshopper obteve melhor desempenho nas Dimensões Cognitivas Expressividade, Notação Secundária e Visibilidade. Não à toa, estas são dimensões que possuem grande peso no aspecto visual da linguagem.

De maneira inversa, as TPLs mostraram mal desempenho nestas dimensões que possuem ênfase nas funcionalidades visuais. A seguir, comentaremos sobre o desempenho positivo da VPL e como as TPLs podem melhorar nestes aspectos.

Expressividade:

Esta dimensão em Grasshopper, identificada nas tarefas 4.4.7 Centralização e controle de dados, e 4.4.8 Organização e clareza do código, ajuda a identificar como as partes do programa se relacionam com o todo e estão diretamente ligadas à facilidade de compreensão do código.

As TPLs utilizadas na arquitetura podem se desempenhar melhor nesta dimensão através do uso de sinalização de agrupamentos funcionalmente relacionados, identificadores significativos e qualquer nível de descrição explícito do código (GREEN; PETRE, 1996). Modularidade bem estruturada também é uma das características descritas por Green e Petre (1996) para melhorar o desempenho da linguagem nesta dimensão. No entanto, identificamos que, de maneira geral, TPLs já possuem capacidade de modularização suficientemente mais avançadas em relação às VPLs.

Notação Secundária:

Esta dimensão em Grasshopper, identificada na tarefa 4.4.8 Organização e clareza do código, também possui grande protagonismo na popularidade da linguagem. O Grasshopper possui uma série de mecanismos que permitem que informações extras possam ser indicadas no código.

As TPLs podem estar mais apropriadas para o uso no fluxo de trabalho do arquiteto caso sejam implementados mecanismos de notação que vão além dos disponíveis na semântica da linguagem utilizada, como uso do layout, cores, marcadores e comentários que transmitam significados auxiliares ao desenvolvimento do projeto.

Um exemplo de pesquisa que visa avançar neste aspecto é o conceito de Programação Ilustrada (*Illustrated Programming*) (LEITÃO; LOPES; SANTOS, 2014). Segundo os autores, devido ao formato e quantidade de detalhes apresentado pelas linguagens textuais, os usuários do sistema podem gastar um esforço significativo para extrair ideias arquitetônicas relevantes em meio aos detalhes de sintaxe do texto. Para ajudar na resolução desse problema, implementaram duas ferramentas.

A primeira auxilia a correlação entre os esboços geralmente produzidos por arquitetos durante a concepção do projeto e o código. Por exemplo, na Figura 94, é possível integrar anotações e esboços, como os que identificam os parâmetros do programa. Isso ajuda a fornecer explicações visuais sobre os objetivos do código.

Figura 94: Trecho de código que contém os esboços e anotações dos projetistas na documentação do programa (à esquerda), o código (à direita em cima) e o modelo tridimensional resultante (à direita embaixo). Fonte: Leitão, Lopes, Santos (2014), p.294.

A segunda realiza a correlação entre o modelo geométrico e o código através do conceito de Rastreabilidade. Esta funcionalidade permite que: “o usuário identifique quais partes do programa são responsáveis por quais partes do modelo gerado e vice-versa” (LEITÃO; LOPES; SANTOS, 2014, p. 294, tradução nossa¹¹⁸). As correlações são bidirecionais, ao selecionar uma expressão no código, a interface mostra os elementos geométricos que resultam dessa expressão (Figura 95). Ao selecionar uma parte do modelo tridimensional, os elementos do código que criam essas geometrias são destacados (Figura 96).

Figura 95: Correlação entre a expressão no código e a geometria correspondente no modelo tridimensional. Fonte: Leitão, Lopes, Santos (2014), p.295.

Figura 96: Correlação entre o modelo tridimensional e as expressões correspondentes no código. Fonte: Leitão, Lopes, Santos (2014), p.295.

¹¹⁸ Do original em inglês: “[...] allows the user to identify which parts of the program are responsible for which parts of the generated model and vice-versa.”

Visibilidade:

Esta dimensão em Grasshopper, identificada nas tarefas 4.4.2 Relação e ordenação entre geometria e listas, e 4.4.8 Organização e clareza do código, permitiu que o usuário possa visualizar os trechos de código simultaneamente com sua representação no modelo tridimensional, aumentando a compreensão do programa. Green e Petre (1996) apontam que uma notação possui um bom grau desta dimensão quando permite que a parte solicitada do código seja prontamente visível ou facilmente acessada. Embora a visibilidade simultânea entre código e modelo tridimensional de Grasshopper tenha possibilitado um bom desempenho na manipulação do código do estudo de aplicação, alguns mecanismos disponíveis nas TPLs ainda mostram melhor desempenho. Por exemplo, ao acessar um cluster no Grasshopper, o usuário não pode alternar a visualização entre o código geral e o código dentro do Cluster, ou colocá-los lado a lado, devendo optar por apenas uma tela de escrita de código. Da mesma maneira, não é possível modularizar um código em Grasshopper em arquivos diferentes, isso significa que o usuário deve utilizar a manipulação de layout disponível para reorganizar os nós, caso necessite visualizá-los lado a lado.

Já em TPLs, devido à possibilidade de trabalhar em módulos, os arquivos de outros códigos, como bibliotecas (equivalentes aos *clusters* do Grasshopper) podem ser abertos paralelamente e visualizados em justaposição.

No entanto, um dos aspectos mais importantes de Visibilidade encontrados nas VPLs em arquitetura, a relação entre código e modelo tridimensional (ou Consistência entre Representações de Aish e Hanna (2016)), é um dos mais sensíveis na falta de legibilidade e aumento de Operações Mentais Difíceis das TPLs.

Melhorar o uso das TPLs utilizadas na arquitetura nas dimensões Expressividade, Notação Secundária e Visibilidade pode trazer melhoras significativas na legibilidade e compreensão geral dos relacionamentos entre código e modelo tridimensional.

Um exemplo de mecanismo que contribui nestas diretrizes é a Rastreabilidade, conforme visto anteriormente em Leitão, Santos e Lopes (2014). Uma interface mais atual que a implementa é Luna Moth (ALFAIATE, 2017). Nela, é possível destacar expressões e seus respectivos resultados, um mecanismo interessante para aumento da Visibilidade. Quando o usuário aponta para uma expressão ou função no código, o ambiente de programação rastreia a relação e sinaliza sua parte correspondente na visualização tridimensional. O mesmo acontece de maneira inversa, quando o usuário seleciona um objeto ou parte do modelo tridimensional, a interface identifica a função correspondente no código TPL.

Figura 97: Rastreabilidade de Luna Moth. Fonte: Alfaiate (2017, p. 29)

A rastreabilidade é muito importante em códigos que se tornaram complexos, uma vez que os aspectos de legibilidade das TPLs é um dos fatores que dificultam seu uso em Design Computacional.

Considerações Finais

A contribuição desta pesquisa está em mostrar evidências de que, independente da linguagem (Grasshopper, Racket, Python, etc) escolhida pelos usuários para programar, existem diretrizes que podem melhorar a usabilidade da computação pela arquitetura.

Indicamos que os sistemas baseados em TPL podem se concentrar em implementar melhoras nas dimensões Notação Secundária, Visibilidade e Expressividade, uma vez que são aspectos importantes para arquitetos. Programar a concepção de um projeto a fim de explorar suas variações formais atreladas a desempenho em uma TPL sem esses recursos é atualmente um desafio. Isso ocorre pela lacuna existente na educação formal de arquitetos em programação e pela falta de exemplos sólidos de aplicação de TPLs ao projeto arquitetônico. Foi reconhecido que as abordagens de visualização instantânea entre o código e o resultado e as manipulações mais interativas promovem a criatividade de uma forma que pode não ser possível com linguagens baseadas em texto (NOONE; MOONEY, 2018).

Esses fatos incitam a utilização de linguagens híbridas, com as quais a mesma semântica pode ser declarada simultaneamente por VPL e TPL, unindo o melhor dos dois mundos.

Uso da Linguagem Híbrida vs. Linguagem Textual

Conforme discutido nas taxonomias das VPLs e nos aspectos encontrados no Estudo de Aplicação, Grasshopper sofre de muitas limitações e restrições no fluxo de dados para lidar com tarefas complexas. É importante lembrar que apesar da indicação de que as VPLs são reconhecidas por sofrerem problemas de escalabilidade (BURNETT et al., 1996), essas características não são necessariamente generalizáveis a todos os sistemas VPL usados na arquitetura.

Por exemplo, como vimos anteriormente, a interface de programação Möbius (JANSSEN; LI; MOHANTY, 2016) oferece o tipo de modelagem processual, que permite a construção de modelos que incorporam *loops*, aninhamentos complexos e funções de ordem superior. Möbius também possui a capacidade de gerar um programa textual, permitindo que os mecanismos mais avançados da TPL possam ser utilizados pelo usuário, o que o caracteriza como uma linguagem híbrida e com o tipo de modelagem VPL mais avançado entre os atualmente disponíveis em arquitetura (a modelagem processual). Embora não tenha sido testado por esta pesquisa, essas características são bons indicadores que a interface tenta resolver a escalabilidade do modelo, a melhor adaptação do sistema a Projetos Orientados ao Desempenho e a transição do conhecimento do usuário, da VPL para a TPL (do nó para o *script*). Muitas outras características podem ser testadas neste sistema, como a interatividade e latência quando utilizado em projetos grandes ou complexos, ou a capacidade de testes e avaliação progressiva do programa.

No entanto, levantaremos alguns pontos que nos ajudam a questionar se pode ser mais interessante melhorar a usabilidade de TPLs para o domínio da arquitetura do que migrar para uma interface híbrida específica. Essas observações foram levantadas devido às características promissoras encontradas no uso de Rosetta e no desenvolvimento em TPL durante a investigação presente nesta pesquisa.

Portabilidade, flexibilidade e obsolescência:

Problemas com a portabilidade dos modelos paramétricos usados na arquitetura são atualmente um objeto de investigação, como visto em Janssen (2015). Até onde pudemos investigar, Rosetta é o único ambiente de programação disponível na área que oferece uma estrutura que resolve o problema de portabilidade. A característica mais promissora do ponto de vista do arquiteto programador é a eficiência em se dedicar a escrita de apenas um programa (código). Desta maneira é possível, por exemplo extrair modelos tridimensionais adaptados à cada fase de projeto usando o mesmo código

Além de resolver o problema de portabilidade, Rosetta oferece um modelo de sistema que avança em relação a obsolescência das linguagens de programação e softwares. Por permitir um acoplamento flexível é possível estender a sobrevivência do código para além da obsolescência destes sistemas. A sobrevivência da grande quantidade de códigos produzidos através de uma linguagem específica, como Grasshopper, depende exclusivamente do tempo em que este sistema permanecerá funcionando (ou do ponto em que seja possível realizar a portabilidade deste código para outra linguagem mais atual).

Além disso, conforme apontado por Lopes e Leitão (2011, p. 197, tradução nossa¹¹⁹) Rosetta é extensível “no sentido de permitir que novos front-ends e back-ends sejam conectados ao ambiente de programação, sem quebrar a portabilidade com os programas existentes”.

Todas essas interfaces de programação orientadas ao usuário final, como Rosetta, Möbius e Grasshopper, dependem do desenvolvimento, documentação e manutenção de seus desenvolvedores. No entanto, é interessante notar que Rosetta possui a vantagem de ter a biblioteca de geometrias e operações em código fonte aberto. Um programador com conhecimento na linguagem em que foi desenvolvida (Racket), pode realizar modificações na própria ferramenta. Programas e bibliotecas de código fonte aberto possuem mais probabilidade em serem melhorados, mantidos ou transformados por usuários da comunidade.

Avaliação progressiva e Confiabilidade:

Em processo de Projeto Orientado ao Desempenho, a confiabilidade de que todo o código execute suas funções corretamente é imprescindível para assegurar que o resultado do projeto atenda de fato suas métricas de performance.

A utilização da TPL proporcionou avanços significativos na confiabilidade geral do fluxo de trabalho por oferecer mecanismos de avaliação progressiva, onde pode-se testar por exemplo se os resultados das simulações foram executados sem erros e possuem o formato correto. A não ser que o usuário investigue manualmente todas as saídas de resultados das simulações no Grasshopper (o que pode ser contraproducente na maioria dos casos, dada a quantidade de dados), a otimização pode considerar um valor de simulação errado. Desta maneira, a confiabilidade de todo o fluxo de trabalho se perde.

Esta clareza e controle sobre o funcionamento do código solicitada pelo Projeto Orientado ao Desempenho está de acordo com o apontamento de Terzidis (2006, p. 39-40), apresentado na introdução desta pesquisa. A crítica do autor está atrelada a esta falta de clareza e controle, uma vez que o uso obscuro destas ferramentas pode levar a “formas estranhas cuja origem,

¹¹⁹ Do original em inglês: “[...] in the sense that it allows new front-ends and back-ends to be plugged-in to the programming environment, without breaking the portability with existing programs.”

processo ou racionalidade de geração são inteiramente desconhecidos e, portanto, são julgadas com base em sua aparência, muitas vezes utilizando textos místicos, enigmáticos ou ofuscantes para explicação”.

Compartilhamento de soluções e versionamento:

Apesar do aprendizado de uma linguagem específica ser cada vez mais acessível a um público sem educação formal em programação, dado a quantidade de material disponível na rede e a ajuda das comunidades de programadores reunidos em fóruns (como Stack Overflow e Quora), é importante criar e fornecer exemplos de como os arquitetos podem modelar e declarar fluxos de trabalho computacionalmente em TPL. Esta necessidade agrava-se principalmente por se tratar de programação específica de um domínio.

A comunidade de usuários de Grasshopper, por exemplo, discute e compartilha códigos VPLs em um fórum específico. Desde sua criação em 2007, até 2012, o fórum já possuía mais de dois mil códigos compartilhados (DAVIS, 2013, p. 81). Isso vai ao encontro com uma das diretrizes apontadas por pesquisadores da educação em programação, a necessidade de “construir bibliotecas de soluções que representam modelos abstratos para problemas, bem como estratégias para coordená-las e compô-las” (SOLOWAY, 1986, p. 850, tradução nossa¹²⁰).

Uma das contribuições da metodologia DSC (HAYMAKER et al., 2018) vai justamente nesta direção. A única diferença desta abordagem é a de que a modularidade e o reaproveitamento do DSC estão limitados a modelagem de fluxo de dados de Grasshopper. Já a reimplementação do DSC em TPL possui muito mais mecanismos para que suas etapas e camadas estejam moduladas e possam ser aplicadas a outros projetos. Por estar escrita em uma TPL, o reaproveitamento ou tradução deste código para outras linguagens também é muito mais acessível. Além disso, conforme vimos, TPLs possuem capacidade de versionamento dos códigos, característica imprescindível para melhorar o trabalho colaborativo entre equipes. Ao utilizar o DSC em Grasshopper, apenas um usuário pode editar o código por vez.

Por fim, além destes apontamentos que indicam a melhor adequação das TPLs em Projetos Orientados ao Desempenho, observamos que os critérios de avaliação para sistemas de softwares de Meyer (1997) possuem muitas das características relevantes na necessidade do desenvolvimento de projetos de arquitetura que utilizam programação, por serem critérios que incentivam a sobrevivência, funcionalidade e portabilidade dos sistemas de modelagem paramétrica. Ressaltamos que Davis (2013) oferece um aprofundamento sobre as relações entre os métodos da engenharia de software e arquitetura.

O Projeto Orientado ao Desempenho requer mais controle e confiabilidade no desenvolvimento do fluxo de trabalho. As limitações das linguagens visuais utilizadas atualmente influenciam no desenvolvimento do projeto computacional e algorítmico. Esclarecimentos sobre o uso das linguagens visuais e textuais são relevantes para que arquitetos possam desenvolver códigos mais adaptados ao tipo de resolução de problema que eles desejam. Concentrar esforços em melhorar e adaptar interfaces de programação textual para que arquitetos possam utilizá-las para desenvolver projetos algorítmicos tem o potencial de melhorar a escalabilidade, confiabilidade e reaproveitamento de soluções para projetos. Esses aspectos são essenciais para fornecer exemplos claros em como arquitetos podem usar todo o potencial da programação como ferramenta de projetos.

¹²⁰ Do original em inglês: “*They have built up large libraries of stereotypical solutions to problems as well as strategies for coordinating and composing them.*”

Referências Bibliográficas

AISH, R. DesignScript: origins, explanation, illustration. Computational Design Modelling, p. 1–8, 2012. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-642-23435-4_1>.

_____.a. DesignScript: Scalable Tools for Design Computation. Computation and Performance - Proceedings of the 31st eCAADe Conference, v. 2, p. 87–95, 2013.

_____.b. First Build Your Tools. Inside Smartgeometry: Expanding the Architectural Possibilities of Computational Design, v. 9781118522, p. 36–49, 2013.

AISH, R.; HANNA, S. Comparative evaluation of parametric design systems for teaching design computation. Design Studies, v. 52, p. 144–172, 2017. Disponível em: <<http://dx.doi.org/10.1016/j.destud.2017.05.002>>.

AISH, R.; WOODBURY, R. Multi-Level Interaction in Parametric Design. International symposium on smart graphics, p. 151–162, 2005. Disponível em: <http://link.springer.com/chapter/10.1007/11536482_13>.

ALFAIATE, A. A Browser-based Programming Environment for Generative Design. 2017. Universidade de Lisboa, 2017.

ALFAIATE, P.; CAETANO, I.; LEITÃO, A. Luna Moth Supporting Creativity in the Cloud. ACADIA 2017 Disciplines & Disruption, p. 72–81, 2017.

BAROTH, E. HARTSOUGH, C. Visual programming in the real world. In: M. Burnett, A. Goldberg and T. Lewis (eds.), Visual Object-Oriented Programming Concepts and Environments. Greenwich, CT: Manning, pp. 21–42. 1995

BENJAMIN, D. Beyond efficiency. In: SCOTT MARBLE (Ed.). Digital Workflows in Architecture : Design – Assembly – Industry. [s.l.] Birkhäuser, 2012. p. 14–25.

BLACKWELL, A. F.; BRITTON, C.; COX, A.; GREEN, T. R. G.; GURR, C.; KADODA, G.; KUTAR, M. S.; LOOMES, M.; NEHANIV, C. L.; PETRE, M.; ROAST, C.; ROE, C.; WONG, A.; YOUNG, R. M. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In: BEYNON M., NEHANIV C.L., D. K. (Ed.). Cognitive Technology: Instruments of Mind. Berlin: Springer,

2001. p. 325–341.

BLACKWELL, A. Psychological Issues in End-User Programming. *End User Development*, v. 9, p. 9–30, 2006. Disponível em: <<http://www.springerlink.com/index/g51r756lm1775281.pdf>>.

BOEYKENS, S.; NEUCKERMANS, H. Improving Design Workflow in Architectural Design Applications. *International Journal of Architectural Computing*, v. 4, n. 4, p. 1–19, 2006. Disponível em: <http://dx.doi.org/10.1260/147807706779399033%5Cnhttp://multi-science.metapress.com/content/u35256687tv00k07/?genre=article&id=doi:10.1260/147807706779399033%5Cnhttp://www.metapress.com/content/u35256687tv00k07/fulltext.pdf>

BOSHERNITSAN, M.; DOWNES, M. S. Visual programming languages: A survey. [s.l.] Computer Science Division, University of California, 2004.

BRELL-COKCAN, S.; BRAUMANN, J.; COKCAN, B.; KLEINDIENST, M. Just in Time Design: Developing parametric design tools for architectural design. *Beyond Codes and Pixels: Proceedings of the 17th International Conference on Computer-Aided Architectural Design Research in Asia*, p. 455–464, 2012.

BROOKS, R. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9(6), pp.737-751. 1977

BURNETT, M. M.; AMBLER, A. L. Interactive visual data abstraction in a declarative visual programming language. *J. Visual Languages and Computing* 5(1) 29-60. 1994

BURNETT, M.; BAKER, M.; BOHUS, C.; CARLSON, P.; YANG, S.; VAN ZEE, P. Scaling up visual programming languages. *Computer*, v. 28, n. 3, p. 45–54, 1995. Disponível em: <<http://ieeexplore.ieee.org/document/366157/>>.

BURRY, M. Narrowing the Gap between CAAD and Computer Programming *Proceedings of the CAADRIA Conference*, 1997.

_____. Parametric Design and the Sagrada Família. *Architectural Research Quarterly*, no. 1 : 70-80. 1996

BURRY, M.; DATTA, S.; ANSON, S. Introductory Computer Programming as a Means for Extending Spatial and Temporal Understanding. *Eternity, Infinity and Virtuality in Architecture Proceedings of the 22nd Annual Conference of the Association for Computer-Aided Design in Architecture (ACADIA)*, p. 129–135, 2000.

CELANI, M. G. C.; VAZ, C. E. V. CAD Scripting And Visual Programming Languages For Implementing Computational Design Concepts: A Comparison From A Pedagogical Point Of View. *International Journal of Architectural Computing*, v. 10, n. 01, p. 121–138, 2012.

CHASZAR, A.; JOYCE, S. C. Generating freedom: Questions of flexibility in digital design and architectural computation. *International Journal of Architectural Computing*, 2016. Disponível em: <<http://jac.sagepub.com/content/early/2016/04/27/1478077116638945.abstract>>.

CLARISSE, O. CHANG, S. K. VICON: A Visual Icon Manager. In: *Visual Languages Plenum Press*, New York, pp. 151-190. 1986

COATES, P. *Programming Architecture*. Routledge, 2010.

CORREIA, H.; LEITÃO, A. Extending Processing to CAD Applications. *eCAADe* 33, v. 1, p. 159–167, 2015.

DAVIS, D. Modelled on Software Engineering : Flexible Parametric Models in the Practice of Architecture. Ph.D Thesis, n. February, p. 243, 2013. Disponível em: <http://www.danieldavis.com/papers/danieldavis_thesis.pdf>.

DAVIS, D.; BURRY, J.; BURRY, M. Understanding Visual Scripts: Improving Collaboration through Modular Programming. *International Journal of Architectural Computing*, v. 9, n. 4, p. 361–375, 2011. Disponível em: <<http://journals.sagepub.com/doi/10.1260/1478-0771.9.4.361>>.

DINO, I. G. Creative design exploration by parametric generative systems in architecture. *Metu Journal of the Faculty of Architecture*, v. 29, n. 1, p. 207–224, 2012.

DRITSAS, S. An advanced parametric modelling library for architectural and engineering design. CAADRIA 2016, 21st International Conference on Computer-Aided Architectural Design Research in Asia - Living Systems and Micro-Utopias: Towards Continuous Designing, n. 2014, p. 611–620, 2016.

EASTMAN, Charles M. et al. BIM handbook: A guide to building information modeling for owners, managers, designers, engineers and contractors. John Wiley & Sons, 2011.

ERHAN, H. I.; FLEMMING, U. User-system interaction design for requirements modelling. CAADRIA 2005 - The Association for Computer-Aided Architectural Design Research in Asia: Digital Opportunities, n. 1, p. 160–170, 2005. Disponível em: <<http://www.scopus.com/inward/record.url?eid=2-s2.0-84875263802&partnerID=40&md5=9edab0089260aa0f25051cc093f1a6c6>>.

FERREIRA, B.; LEITÃO, A. Generative Design for Building Information Modeling. *eCAADe* 33, v. 1, p. 635–644, 2015.

FISCHER, T.; HERR, C.M. Teaching generative design. In *Proceedings of the 4th Conference on Generative Art*. 2001.

GERBER, D.J., 2007. Parametric practices: models for design exploration in architecture. Harvard University.

GREEN, T. Cognitive dimensions of notations. *Proceedings of the fifth conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and computers V*, p. 443–460, 1989. Disponível em: <<http://www.cl.cam.ac.uk/users/afb21/CognitiveDimensions/papers/Green1989.pdf>>.

GREEN, T. R. G.; PETRE, M. Usability Analysis of Visual Programming Environments: A {“Cognitive Dimensions”} Framework. *Jvlc*, v. 7, n. 2, p. 131–174, 1996. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S1045926X96900099>>.

GROSS, M. D. FormWriter: A Little Programming Language for Generating Three-Dimensional Form Algorithmically. *Computer Aided Architectural Design Futures* 2001, p. 577–588, 2001.

HARDING, J. Meta-Parametric Design: Developing a Computational Approach for Early Stage Collaborative Practice. [s.l.] University of Bath, 2014.

HAYMAKER, J.; BERNAL, M.; RESEARCHER, S.; DESIGNER, C.; MARSHALL, M. T.; RESEARCHER, S.; OKHOYA, V.; RESEARCHER, S.; MANAGER, D. A.; SZILASI, A.; REZAEI, R.;

CHEN, C.; SALVESON, A.; ENGINEER, S.; BRECHTEL, J.; DESIGNER, S. C.; DECKINGA, L.; MANAGER, D. P.; HASAN, H.; EWING, P.; WELLE, B. Design Space Construction: A framework to support collaborative, parametric decision making. v. 23, n. June, p. 157–178, 2018.

HERNANDEZ, C.R.B. Thinking parametric design: introducing parametric Gaudi, Design Studies, R. Oxman (ed.) special issue on Digital Design, 27, pp. 309- 324. 2006

HILL, M. D. What is scalability? ACM SIGARCH Computer Architecture News 18, no. 4. 1990. 18-21.

HOLZER, D.; HOUGH, R.; BURRY, M. Parametric Design and Structural Optimisation for Early Design Exploration. International Journal of Architectural Computing, v. 5, n. 4, p. 625–643, 2007. Disponível em: <<http://jac.sagepub.com/content/5/4/625.short>>.

HUDSON, R. Strategies for parametric design in architecture: An application of practice led research. [s.l.] University of Bath, 2010.65–72, 2010.

JANSSEN, P. Dexen: A scalable and extensible platform for experimenting with population-based design exploration algorithms. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, v. 29, n. 04, p. 443–455, 2015. Disponível em: <http://www.journals.cambridge.org/abstract_S0890060415000438>.

_____. b. Parametric BIM workflows. CAADRIA 2015 - 20th International Conference on Computer-Aided Architectural Design Research in Asia: Emerging Experiences in the Past, Present and Future of Digital Architecture, p. 437–446, 2015. Disponível em: <<http://www.scopus.com/inward/record.url?eid=2-s2.0-84936085002&partnerID=tZotx3y1>>.

JANSSEN, P.; CHEN, K. W.; MOHANTY, A. Automated Generation of BIM Models. eCAADe 34, v. 2, p. 583–590, 2016. Disponível em: <http://papers.cumincad.org/cgi-bin/works/Show?ecaade2016_239>.

JANSSEN, P.; LI, R.; MOHANTY, A. MÖBIUS: A parametric modeller for the web. Living Systems and Micro-Utopias: Towards Continuous Designing, Proceedings of the 21st International Conference on Computer-Aided Architectural Design Research in Asia (CAADRIA 2016), n.

JANSSEN, P.; STOUFFS, R. Types of parametric modelling. CAADRIA 2015 - 20th International Conference on Computer-Aided Architectural Design Research in Asia: Emerging Experiences in the Past, Present and Future of Digital Architecture, n. May, p. 157–166, 2015.

JANSSEN, P.; STOUFFS, R.; MOHANTY, A.; TAN, E. Parametric Modelling with GIS. eCAADe 2016, v. 2, p. 59–68, 2016. Disponível em: <https://www.researchgate.net/profile/Rudi_Stouffs/publication/307413179_Parametric_Modelling_with_GIS/links/57c54d0408ae496e4210e78b.pdf>.

JANSSEN, P.; WEE, C. K. Visual Dataflow Modelling: A Comparison of Three Systems. Proceedings of the 14th International Conference on Computer Aided Architectural Design Futures, p. 801–816, 2011. Disponível em: <http://cumincad.scix.net/cgi-bin/works/Show?_id=cf2011_p075>.

KAUREL, H. G. Easing the Transition from Visual to Textual Programming. n. December, 2016.

KILLIAN, A., Design innovation through constraint modeling, International Journal of Architectural Computing, 2006, 4(1), 87-105.

KIPER, J. D.; HOWARD, E.; AMES, C. Criteria for Evaluation of Visual Programming Languages. *Journal of Visual Languages & Computing*, v. 8, n. 2, p. 175–192, 1996. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S1045926X96900348>>.

KNUTH, D. E. *The Art of Computer Programming Vol. I.* [s.l.] Addison-Wesley, 1969.

KOLAREVIC, B. *Architecture in the Digital Age: Design and Manufacturing.* New York and London: Spon Pres Taylor & Francis Group, 2003.

KOLAREVIC, B.; MALKAWI, A. *Performative Architecture: Beyond Instrumentality.* New York and London, Spon Press, 2005.

KOLARIC, S. ERHAN, H. WOODBURY, R. RIECKE, B. Comprehending Parametric CAD Models: An Evaluation of Two Graphical User Interfaces. In *Extending Boundaries: Proceedings of the 6th Nordic Conference on Human-Computer Interaction*, edited by Ann Blandford, Jan Gulliksen, Ebba Hvannberg, Marta Larusdottir, Effie Law, and Hannes Vilhjalmsson, 707–710. New York: Association for Computing Machinery. 2010

KOTNIK, T. Digital architectural design as exploration of computable functions. *International journal of architectural computing* 8.1 p. 1-16. 2010

LABELLE, G.; NEMBRINI, J.; HUANG, J. Programming framework for architectural design ANAR+: Object oriented geometry. *Languages and Cultures*, 2009. Disponível em: <http://gsct3237.kaist.ac.kr/e-lib/Design_Knowledge/cf2009_771.content.pdf>.

LEE, E. A. MESSERSCHMITT, D. G. Synchronous data flow, *Proceedings of the IEEE*, 75(9), 1235–1245. 1987

LEITÃO, A. Teaching Computer Science for Architecture: a proposal. *Future Traditions: 1ST eCAADe Regional International Workshop*, p. 95–104, 2013.

LEITÃO, A.; CABECINHAS, F.; MARTINS, S. Revisiting the Architecture Curriculum: The programming perspective. *28th eCAADe Conference Proceedings*, p. 81–88, 2010.

LEITÃO, A.; FERNANDES, R.; SANTOS, L. Pushing the Envelope: Stretching the Limits of Generative Design. *Knowledge-based Design - Proceedings of the 17th Conference of the Iberoamerican Society of Digital Graphics SIGRADI*. Valparaiso, Chile, p. 235–238, 2013. Disponível em: <www.proceedings.blucher.com.br/article-details/pushing-the-envelope-stretching-the-limits-of-generative-design-14129>.

LEITÃO, A.; LOPES, J.; SANTOS, L. Illustrated Programming. *Acadia 2014: Design Agency*, p. 291–300, 2014.

LEITÃO, A.; SANTOS, L. Programming Languages For Generative Design: Visual or Textual? *eCAADe* 29, n. 01, p. 549–557, 2011.

LEITÃO, A.; SANTOS, L.; LOPES, J. Programming Languages For Generative Design: A Comparative Study. *International Journal of Architectural Computing*, v. 10, n. 1, p. 139–162, 2012. Disponível em: <<http://jac.sagepub.com/lookup/doi/10.1260/1478-0771.10.1.139>>.

LOPES, J.; LEITÃO, A. Portable generative design for CAD applications. *Integration Through Computation - Proceedings of the 31st Annual Conference of the Association for Computer Aided Design in Architecture, ACADIA 2011*, p. 196–203, 2011.

MANDELBROT, Benoit B.; PIGNONI, Roberto. The fractal geometry of nature. New York: WH freeman, 1983.

MENGES, Achim; AHLQUIST, Sean. Computational Design Thinking: Computation Design Thinking. John Wiley & Sons, 2011.

MENZIES, T. Evaluation issues for visual programming languages. In Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies. 2002

MEYER, B. Object-Oriented Software Construction. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, v. 352, n. 1–2, p. 379–382, 1997.

MILLER, N. Positioning Computational Designers in your Business – 4 Things to Consider. 2016. Disponível em: <<https://provingground.io/2017/02/15/positioning-computationaldesigners-%0Ain-your-business-4-things-to-consider/%0A>>.

MITCHELL, W. J., LIGGET, R. S., KVAN, T., The Art of Computer Graphics Programming, Van Nostrand Reinhold, New York, 1987

MITCHELL, W. J., The theoretical foundation of computer-aided architectural design, Environment and Planning B, 2(2), 127 - 150. 1978

_____. The logic of architecture: Design, computation, and cognition. Cambridge, MA: MIT press, 1990.

MYERS, B. Taxonomies of Visual Programming and Program Visualization. Journal of Visual Languages & Computing, v. 1, n. 1, p. 97–123, 1990.

NATIVIDADE, V. G. Fraturas metodológicas nas arquiteturas digitais. p. 302, 2011. Disponível em: <<http://www.teses.usp.br/teses/disponiveis/16/16138/tde-16062010-153027/pt-br.php>>.

NOONE, M.; MOONEY, A. Visual and textual programming languages : a systematic review of the literature. Journal of Computers in Education, 2018. Disponível em: <<https://doi.org/10.1007/s40692-018-0101-5>>.

OXMAN, N. Get Real Towards Performance-Driven Computational Geometry. International Journal of Architectural Computing (IJAC). 4(5): 663 -684., v. 05, n. 04, p. 663–684, 2007.

OXMAN, R. Theory and design in the first digital age. Design Studies 27, p. 1–32, 2006.

PAYNE, A.; ISSA, R. The Grasshopper Primer: for version 0.6.0007. 2009. Disponível em:<<http://www.liftarchitects.com/journal/2009/3/25/the-grasshop- per-primer-second-edition.html>>. Acesso em: 20 abr. 2011.

PETERS, B.; PETERS, T. Inside Smartgeometry. Expanding the Architectural Possibilities of Computational Design. AD Smart 01, 2013.

PRUSINKIEWICZ, Przemyslaw; LINDENMAYER, Aristid. The algorithmic beauty of plants. Springer Science & Business Media, 2012.

SADEGHPOUR ROUDSAR, M., PARK, M., SMITH, A. Ladybug: a parametric environmental plugin for grasshopper to help designers create an environmentally-conscious design. In Proceedings of the 13th international IBPSA conference held in Lyon, France Aug. 2013.

- SAITO, D.; WASHIZAKI, H.; FUKAZAWA, Y. Comparison of text-based and visual-based programming input methods for first-time learners. v. 16, p. 209–226, 2017.
- SANTOS, L.; LOPES, J.; LEITÃO, A. Collaborative Digital Design: when architects meets the software engineer. eCAADe 30, v. 2, p. 87–96, 2012.
- SCHMITT, G. Information Architecture. Basi e futuro del CAAD. Torino: Testo & Immagine, 1998.
- SCHÖN, D. A. The reflective practitioner: How professionals think in action. [s.l.] Basic Books, 1984.
- SEBESTA, R. W. Conceitos de linguagens de programação. Tradução técnica: Eduardo Kessler Piveta. 9. ed. Porto Alegre: Bookman, 2011.
- SHACKELFORD, R. L.; BADRE, A. N. Why can't smart students solve simple programming problems? International Journal of Man-Machine Studies, 1993, 38.6: 985-997.
- SHEA, K.; AISH, R.; GOURTOVAIA, M. Towards integrated performance-driven generative design tools. Automation in Construction, v. 14, n. 2 SPEC. ISS., p. 253–264, 2005.
- SHI, X. Performance-based and performance-driven architectural design and optimization. Frontiers of Architecture and Civil Engineering in China, v. 4, n. 4, p. 512–518, 2010.
- SHI, X.; YANG, W. Performance-driven architectural design and optimization technique from a perspective of architects. Automation in Construction, v. 32, p. 125–135, 2013. Disponível em: <<http://dx.doi.org/10.1016/j.autcon.2013.01.015>>.
- SMITH, R. Technical Notes From Experiences and Studies in Using Parametric and BIM Architectural Software. 2007. Published 4 March. <http://www.vbtlc.com/images/VBTTechnicalNotes.pdf>.
- SOLOWAY, E. Learning to program= learning to construct mechanisms and explanations. Communications of the ACM, 1986, 29.9: 850-858.
- STEINFELD, K.; KO, J. Decodes. A platform-independent computational geometry environment Kyle. Open Systems: Proceedings of the 18th International Conference on Computer-Aided Architectural Design Research in Asia, p. 499–508, 2013.
- TERZIDIS, K. Algorithmic Architecture. First Edit ed. Oxford: Elsevier, 2006.
- VAN ROY, P. Programming paradigms for dummies: What every programmer should know. New computational paradigms for computer music. 2009 Jun;104:616-21.
- VILLARES, A. Resources for Teaching Programming for Artists, Designers and Architects: Host Platforms & Languages. Disponível em: <<https://villares.github.io/csv-to-html-table/host-platforms-and-languages>>.
- WHITLEY, K. N. Visual Programming Languages and the Empirical Evidence For and Against. Journal of Visual Languages and Computing, v. 8, n. October, p. 109–142, 1997.
- WINSLOW, L. E. Programming Pedagogy -- A Psychological Overview, SIGCSE Bulletin 28(3): 17-25. 1996
- WOLFRAM, Stephen. A new kind of science. Champaign: Wolfram media, 2002.
- WOODBURY, R. Elements Of Parametric Design. New York: Routledge, 2010.