UNIVERZA V LJUBLJANI | FGG Fakulteta za gradbeništvo in geodezijo

A ASHIQUL MURSALIN CHY

PARAMETRIC GENERATION OF STANDARDIZED SPACES

PARAMETRIČNO GENERIRANJE STANDARDIZIRANIH PROSTOROV

BIM A+ European Master in Building Information Modelling

Master thesis No.:

Supervisor:
Asist. Prof. Tomo Cerovšek, Ph.D.

President of the Committee
Prof. Goran Turk, Ph.D.

Ljubljana, 2025

ERRATA

Page	Line	Error	Correction

BIBLIOGRAFSKO – DOKUMENTACIJSKA STRAN IN IZVLEČEK

UDK: 004.42:69.01/.07(043.2)

Avtor: A Ashiqul Mursalin Chy

Mentor: doc. dr. Tomo Cerovšek, Ph.D.

Naslov: Parametrično generiranje standardiziranih prostorov

Tip dokumenta: Magistrsko delo

Obseg in oprema: 96 str., 60 sl., 2 pregl, 5 pril.

Ključne besede: Parametrično generiranje, avtomatizacija stanovanjskega načrtovanja, skladnost z gradbeno zakonodajo, PyRevit, obdelava naravnega jezika, BIM

Izvleček:

Arhitekturni načrtovanje stanovanjskih objektov je pogosto neučinkovito, ker tradicionalni delotoki zahtevajo tedne za zasnovo in mesece za razvoj, kar ovira manjše investitorje in preobremenjuje arhitekte s tehničnimi nalogami. Magistrska naloga predstavlja sistem za parametrično generiranje standardiziranih prostorov, ki premošča vrzel med uporabniškimi nameni in informacijskim modeliranjem stavb (BIM) z integracijo obdelave naravnega jezika, skladnosti z gradbeno zakonodajo in neposrednega generiranja modelov BIM. Sistemska arhitektura obsega tri komponente: zunanji uporabniški vmesnik z interpretacijo naravnega jezika po meri, vtičnik PyRevit in komunikacijski protokol na osnovi datotek. Implementacija vključuje preverjene zahtev gradbene zakonodaje iz desetih držav, kar omogoča proaktivno zagotavljanje skladnosti med generiranjem rešitev, namesto naknadnega preverjanja. S parametričnim prilagajanjem na osnovi predlog in algoritmi za porazdelitev površin sistem v nekaj sekundah generira podrobne, s predpisi skladne modele BIM za konfiguracije stanovanj, kar nadomešča procese, ki so zahtevali tedne. Testiranje je pokazalo uspešno generiranje raznolikih konfiguracij od garsonjer do štirisobnih stanovanj, ob ohranjanju arhitekturne kakovosti in regulativne skladnosti. Z demokratizacijo dostopa do strokovnih načrtovalskih orodij sistem omogoča investitorjem, strankam in arhitektom, da raziskujejo in generirajo arhitekturne rešitve brez tehničnih ovir. Ta raziskava potrjuje, da lahko vmesniki z naravnim jezikom uspešno prevedejo človeško prostorsko predstavo v kompleksna opravila BIM brez zmanjšanega pomena strokovnih standardov, kar korenito spreminja način zasnove, načrtovanja in izvedbe standardiziranih stanovanjskih prostorov.

BIBLIOGRAPHIC– DOKUMENTALISTIC INFORMATION AND ABSTRACT

UDC: 004.42:69.01/.07(043.2)

Author: A Ashiqul Mursalin Chy

Supervisor: Assist. Prof. Tomo Cerovšek, Ph.D.

Title: Parametric Generation of Standardized Spaces

Document type: Master Thesis

Scope and tools: 96p, 60 fig, 2 tab, 5 ann.

Keywords: Parametric Generation, Residential Design Automation, Building Code Compliance, PyRevit, Natural Language Processing, BIM

Abstract:

The architectural design process for residential developments faces a critical inefficiency where traditional workflows require weeks for preliminary designs and months for detailed development, creating barriers for smaller investors and overwhelming architects with repetitive technical tasks. This thesis presents the Parametric Generation of Standardized Spaces, a comprehensive system that bridges the gap between user intent and Building Information Modelling (BIM) through the integration of natural language processing, building code compliance, and direct BIM generation. The system architecture comprises three interconnected components: an external user interface with custom natural language interpretation, a PyRevit plugin implementing parametric generation within Autodesk Revit, and a robust file-based communication protocol. The implementation successfully incorporates verified building code requirements from ten countries, enabling proactive compliance during generation rather than post-generation verification. Through template-based parametric adjustment and sophisticated area distribution algorithms, the system generates fully detailed, code-compliant apartment BIM models within seconds, replacing processes that traditionally required weeks. Testing demonstrated successful generation of diverse configurations from studio to four-bedroom layouts, while maintaining architectural quality and regulatory compliance. By democratizing access to professional-grade design tools, the system empowers investors, clients, and architects alike to explore and generate architectural solutions without technical barriers. This research validates that natural language interfaces can successfully translate human spatial thinking into complex BIM operations without sacrificing professional standards, fundamentally transforming how standardized residential spaces are conceived, designed, and delivered.

ACKNOWLEDGEMENTS

My deepest gratitude goes first to Allah, whose guidance and blessings enabled me to navigate this challenging academic journey.

This thesis stands as a testament to the unwavering support of my wife, Masuma Chowdhury, whose presence has been both my anchor and my sail throughout this endeavor.

I extend my appreciation to my supervisor, Prof. Tomo CEROVSEK, whose mentorship transcended traditional academic boundaries. His expertise in BIM, coupled with his genuine investment in my professional development, shaped not only this research but also my understanding of what excellence in academia truly means. His efforts in facilitating industry connections with architectural and construction firms proved invaluable in grounding this work in practical reality.

To Masuma, my life partner and intellectual companion, your multifaceted support, spanning financial, emotional, and academic dimensions, transformed an ambitious dream into a tangible achievement. Your patience during countless late nights and your insightful feedback on my work have been instrumental in reaching this milestone.

I honor my parents, whose sacrifices and unconditional love laid the foundation upon which all my achievements stand. Their belief in education as a transformative force continues to inspire my journey.

I owe a debt of gratitude to Engineer Sangeen Khan, whose mentorship and innovative insights regarding MCP became a turnaround of this thesis. His guidance helped shape the conceptual framework that made this parametric generation system possible.

The BIM A+, European Master in Building Information Modelling program deserves special recognition for providing this transformative educational experience. I particularly acknowledge José Granja, whose brilliance in BIM opened new horizons in my understanding; Miguel Azenha and Maria Laura Leonardi for their pedagogical excellence; Bruno Muniz and Bruno Figueiredo for their practical insights; Luka Gradišar, Prof. Žiga Turk, and Prof. Tomo CEROVSEK for broadening my theoretical perspectives.

My journey was enriched by the camaraderie of exceptional friends: Víctor Fernández de Manzanos, Sagar Chandra Singha, and Shaieen Kadir, whose friendship provided both intellectual stimulation and emotional sustenance. I also thank my BIM A+ Colleagues Afonso Ramos Portela, Ahtisham Ali Baig, Bakht Yaseen, Haris Waheed Bhatti, and Mouadh Khammassi for creating a supportive community that made this international academic experience truly memorable.

Each person mentioned here contributed uniquely to this achievement, and while this page cannot fully capture my gratitude, I hope it serves as a lasting acknowledgment of their invaluable roles in my academic journey.

TABLE OF CONTENTS

INDEX OF FIGURES

INDEX OF TABLES

1 INTRODUCTION

1.1 BACKGROUND

The architectural design process stands at a critical juncture where traditional methodologies increasingly struggle to meet the accelerating demands of modern construction and real estate development. In contemporary practice, the journey from initial concept to constructible Building Information Model (BIM) typically spans weeks or months. This process involves multiple iterations between architects, clients, and regulatory authorities.

The extended timeline inflates project costs and creates significant barriers for smaller investors and developers. These stakeholders often cannot afford lengthy design exploration phases. The emergence of Building Information Modelling as the industry standard has paradoxically both enhanced and complicated this process. BIM enables unprecedented coordination and documentation accuracy. However, it has also introduced layers of technical complexity that require specialized expertise to navigate effectively.

The parametric design paradigm offers a solution to these challenges. It encodes design logic into adjustable rule sets that can generate multiple variations from a single framework. Yet, existing parametric tools remain largely inaccessible to non-specialists. Users typically require either programming knowledge or extensive training in visual scripting environments. This technical barrier has confined parametric design to specialized applications despite its potential to democratize architectural exploration.

The integration of building code compliance, a fundamental requirement for any constructible design, remains predominantly manual. This creates a disconnect between generative capabilities and regulatory requirements. The efficiency gains promised by automation are thus undermined. Recent advances in natural language processing and user interface design have created unprecedented opportunities to bridge the gap between human intent and computational execution. The ability to translate conversational specifications into precise parametric operations represents a paradigm shift in how users might interact with complex design systems.

This thesis explores the development of a comprehensive system that leverages these advances. The system creates an accessible, code-compliant parametric generation tool specifically tailored for standardized residential spaces. By focusing on apartment design, a domain characterized by well-established patterns yet requiring careful customization, this research demonstrates how intelligent automation can transform architectural workflow without sacrificing design quality or regulatory compliance.

1.2 PROBLEM STATEMENT

The current architectural design process for residential developments presents an inefficiency that affects all stakeholders in the construction value chain. Investors and developers seeking to evaluate potential projects must first engage architectural firms to produce preliminary designs. This process typically requires 2-4 weeks for initial concepts and an additional 4-8 weeks for detailed development. [1].

The high upfront cost of architectural services prevents many smaller investors from exploring multiple design options or sites. Preliminary studies often range from €5,000 to €20,000 [1]. This effectively limits market participation to well-capitalized entities. The sequential nature of traditional design development compounds this issue. Fundamental problems, such as building code violations or site constraints, are often discovered late in the process. These discoveries necessitate costly revisions that further extend timelines

For architectural firms, the current workflow creates its own set of challenges. Junior architects spend countless hours on repetitive tasks. These include adjusting room dimensions to meet area requirements, verifying code compliance, and producing documentation for similar apartment typologies. This inefficient use of human expertise not only increases project costs but also contributes to professional burnout and limits time available for creative design exploration.

The manual nature of code compliance checking is particularly problematic [2]. Architects must cross-reference dozens of requirements across multiple documents. This process is prone to human error. Errors can result in expensive corrections during construction or regulatory rejection of completed designs.

The integration challenges between design ideation and BIM production compound these inefficiencies. Even when architects use sophisticated tools like Revit, the translation from conceptual design to a detailed BIM model requires extensive manual work. Each wall must be drawn. Every door and window must be placed. All rooms must be defined and tagged. While necessary for documentation, these tasks add little creative value.

The lack of intelligent automation means that minor changes can trigger hours of manual modifications. Adjusting an apartment's total area or switching between building codes exemplifies this problem. Technical debt accumulates throughout the project lifecycle. Design iterations become increasingly expensive. This discourages exploration of alternatives that might better serve client needs or site opportunities.

1.3 IMPORTANCE

The significance of this research extends beyond efficiency gains. By automating the generation of code-compliant apartment layouts, this system democratizes access to professional-grade architectural tools. Smaller investors and community developers can now explore design options that were previously cost-prohibitive.

This democratization is particularly important in addressing the global housing crisis. The speed and cost of design development often determine whether affordable housing projects achieve financial viability. The ability to generate multiple compliant options within minutes rather than weeks fundamentally changes the economics of residential development [1]. This could unlock sites and opportunities that current processes render unfeasible.

From an architectural practice perspective, this work represents a crucial evolution in how computational tools augment rather than replace human creativity. By automating repetitive technical tasks, dimension calculations, code compliance verification, and BIM model generation, the system liberates architects to focus on their core competencies. These include solving complex spatial problems, creating meaningful spaces, and responding to unique site conditions. This shift from manual drafting to design orchestration aligns with the profession's trajectory toward strategic, value-added services.

The integration of building code intelligence directly into the generation process represents a paradigm shift in regulatory compliance. This benefits all stakeholders in distinct ways. Developers gain confidence that generated designs will meet regulatory requirements, reducing the risk of costly surprises during permitting. Architects eliminate tedious manual verification processes while reducing professional liability exposure. Regulatory authorities could eventually enable faster permit review and approval processes through standardized compliance checking.

The system's support for multiple international building codes demonstrates the feasibility of creating globally applicable tools that respect local regulations. This capability is critical as architectural practice increasingly operates across borders. The research contributes not only to immediate productivity improvements but also to the longer-term transformation of how the built environment is conceived, designed, and delivered.

1.4 OBJECTIVES

This thesis aims to develop and validate a comprehensive parametric generation system. This bridges the gap between user intent and building code-compliant BIM models. It specifically targets the automation of standardized residential spaces. The system addresses critical gaps identified in existing solutions through an integrated approach. This approach combines natural language processing, building

code intelligence, and direct BIM generation within a unified workflow. The workflow maintains both accessibility for non-specialists and precision for professional practice.

The research objectives are organized into the following thematic areas:

i. **Interface and Accessibility:**

O1.1: Design and implement a natural language interface that accurately interprets architectural requirements without requiring specialized technical knowledge

O1.2: Develop a sophisticated post-generation modification system through natural language commands for iterative design refinement

O1.3: Demonstrate the system's accessibility to non-expert users while maintaining the precision required by professional architects

ii. **Technical Implementation:**

O2.1: Create a template-based parametric engine that generates fully detailed apartment BIM models directly in Revit

O2.2: Establish a reliable file-based communication protocol between the external user interface and Revit plugin

O2.3: Implement intelligent area distribution algorithms that automatically balance room dimensions

iii. **Compliance & Validation:**

O3.1: Develop a building code compliance system that automatically validates and adjusts designs according to regulations from ten different countries

O3.2: Validate the system's effectiveness through the successful generation of multiple apartment configurations across different building codes

O3.3: Evaluate time savings and workflow improvements compared to traditional manual processes

iv. **Foundation for Future Development:**

O4.1: Establish a foundation for future expansion to other building typologies and more complex architectural programs beyond standardized residential spaces

1.5 THESIS STRUCTURE

Chapter 01: Introduction establishes the research context and critical need for automated, code-compliant design generation in contemporary architectural practice. It articulates specific problems

faced by investors, developers, and architects in current workflows. The chapter demonstrates how these challenges limit both market participation and design exploration.

Chapter 02: Literature Review examines the evolution of automated floor plan generation from early constraint-based approaches through current machine learning systems. Both academic research and commercial solutions are analyzed to identify five critical gaps preventing professional adoption. The review establishes the theoretical foundation while demonstrating the persistent disconnect between algorithmic achievements and practical requirements.

Chapter 03: Methodology presents the research framework and development approach used to create the parametric generation system. It details the problem decomposition strategy, system development methodology, and technical implementation decisions. The chapter also describes validation methods and the evaluation framework used to assess system effectiveness.

Chapter 04: Parametric Generation System Architecture provides an in-depth technical exposition of how user requirements are transformed into code-compliant BIM models. It describes the three primary components, external user interface, PyRevit plugin, and file-based communication layer- and their intricate interactions. The chapter demonstrates how various capabilities integrate to create a cohesive workflow bridging accessibility and professional utility.

Chapter 05: Evaluation and Discussion provides a comprehensive assessment of the developed system through test cases, comparative analysis, and strategic evaluation. The chapter examines system performance, compares it with existing solutions, and presents a SWOT analysis. It concludes with a discussion of findings and an acknowledgment of limitations.

Chapter 05: Conclusion synthesizes the research achievements and addresses how the developed system fulfills the stated objectives. It evaluates practical implications for architectural practice and acknowledges current limitations. The chapter maps pathways for future development and demonstrates that natural language interfaces can successfully bridge the gap between human design intent and complex BIM operations.

2 LITERATURE REVIEW

2.1 INTRODUCTION

The architectural design process has undergone a significant transformation with the advent of computational design methods and Building Information Modelling (BIM) technologies. Yet, despite decades of research into automated floor plan generation, a fundamental disconnect persists between theoretical algorithmic achievements and the practical requirements of architectural practice. This literature review examines the evolution of automated floor plan generation systems, analyzing both academic contributions and commercial solutions to identify the critical gaps that prevent widespread adoption in professional practice. Through this analysis, we establish the foundation for understanding how the integration of natural language interfaces, building code compliance, and direct BIM generation, as implemented in our proposed system, addresses these long-standing challenges.

The automation of architectural design presents unique challenges that distinguish it from other domains of computer-aided design. Unlike mechanical or industrial design, where components follow strict engineering constraints, architectural spaces must satisfy a complex web of requirements: building codes that vary by jurisdiction, cultural preferences that influence spatial arrangements, site-specific constraints that shape building form, and the ineffable qualities that make spaces liveable and meaningful [3].

2.2 THE EVOLUTION OF AUTOMATED FLOOR PLAN GENERATION

2.2.1 FROM CONSTRAINT SATISFACTION TO MACHINE LEARNING

The journey toward automated floor plan generation began with constraint-based approaches that treated spatial layout as an optimization problem. Early research established the computational complexity of floor plan generation, demonstrating that even simplified versions of the problem were NP-hard (nondeterministic polynomial time-hard) [4]. These foundational works introduced constraint satisfaction as a fundamental paradigm, where rooms were allocated to satisfy dimensional requirements, adjacency relationships, and geometric constraints.[5]

The hybrid approach developed by HABX represents a significant advancement in this tradition, combining constraint programming with genetic optimization to generate apartment layouts within arbitrary polygonal envelopes [6]. Their Optimizer algorithm discretizes floor space into a grid, reducing the complex continuous problem to a discrete cell assignment task. (Figure 1) This discretization enables the system to generate architecturally valid floor plans within approximately one minute—a remarkable achievement in computational efficiency. However, the reliance on grid-based representation introduces limitations: the resulting layouts, while functionally valid, often lack the nuanced spatial qualities that emerge from human design intuition.

Figure 1: Floor plan with Optimized Grid, Source: [6]

Graph-based representations have emerged as an alternative approach that better preserves topological relationships during generation. The Graph-Aided Design and Generation (GADG) system demonstrates that dual graph representations can maintain room adjacencies while allowing rapid customization [4]. By deriving a dual graph from existing floor plans and applying transformation rules (Figure 2), GADG generates highly customized layouts in milliseconds. This speed is impressive, yet the system's limitation to rectangular rooms and single-floor configurations reveals the persistent challenge of balancing computational efficiency with architectural complexity.

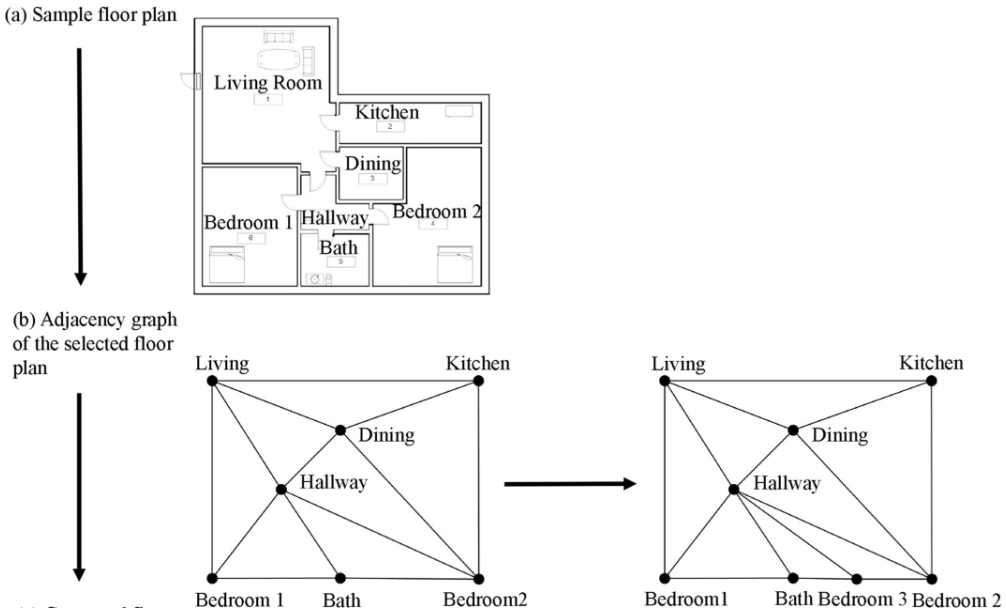

Figure 2: Graph-Aided Design and Generation, Source: [4]

The recent emergence of machine learning approaches, particularly generative adversarial networks (GANs), promises to capture the implicit design knowledge embedded in existing floor plans. House-GAN exemplifies this approach, using relational GANs to generate layouts from bubble diagrams that

encode room types and adjacencies [7] (Figure 3). The system learns from 117,000 real floor plan images, producing diverse and realistic layouts that maintain specified spatial relationships. Yet, while House-GAN excels at generating visually plausible layouts, it struggles with ensuring building code compliance and cannot guarantee that generated designs meet specific dimensional requirements, critical factors for professional practice.

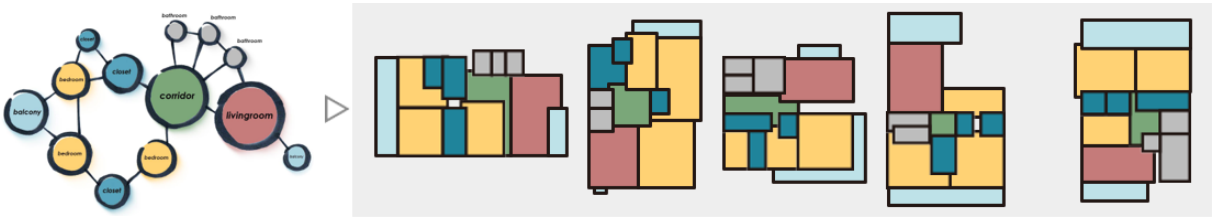

Figure 3: House-GAN, Graph-based house layout generator, Source: [7]

2.2.2 THE PERSISTENT CHALLENGE OF BUILDING CODE COMPLIANCE

Perhaps the most significant gap in automated floor plan generation research is the treatment of building code compliance. Most academic systems either ignore regulatory requirements entirely or implement simplified dimensional checks that fail to capture the complexity of real building codes [8]. Building codes are not merely collections of minimum dimensions; they represent intricate systems of interrelated requirements governing everything from structural safety to accessibility to environmental performance.

The Design Automation Tool (DAT) developed at Warsaw University of Technology attempts to address this gap by incorporating zoning rules and sunlight analysis into the generation process [8]. Operating through three stages: massing optimization, floor division, and apartment layout generation, DAT can produce complete apartment blocks in approximately 25 minutes. The inclusion of regulatory considerations represents progress, yet the system's limitation to point block typologies and its reliance on specific software platforms (Rhinoceros with Grasshopper) highlights the challenge of creating universally applicable solutions.

The complexity of building codes extends beyond simple geometric constraints. Requirements for natural lighting, for instance, involve not just window-to-floor ratios but also considerations of orientation, obstruction angles, and daylight factors that vary by room type and geographic location [9]. Accessibility requirements encompass not only door widths but also manoeuvring spaces, reach ranges, and approach clearances that must be verified in three dimensions. Fire safety regulations create interdependent requirements for egress paths, compartmentalization, and detection systems that cannot be validated through simple dimensional checks.

2.3 FROM 2D LAYOUTS TO BIM INTEGRATION

2.3.1 THE BIM GAP IN ACADEMIC RESEARCH

While numerous research projects successfully generate 2D floor plans, the transition to Building Information Models remains largely unexplored in academic literature. This gap is particularly problematic given BIM's industry-standard status [3]. The disconnect between 2D layout generation and BIM production means that even successful automated layouts require extensive manual work to become usable in professional workflows.[10]

An evolutionary approach for spatial architecture layout design demonstrates the potential for multi-level building generation through agent-based topology finding combined with grid-based optimization [11]. The system successfully generates complex multi-story layouts in 3-15 minutes, proving that automated systems can handle three-dimensional spatial relationships (Figure 4). However, the output remains geometric rather than semantic, lacking the rich information structure that defines true BIM models.

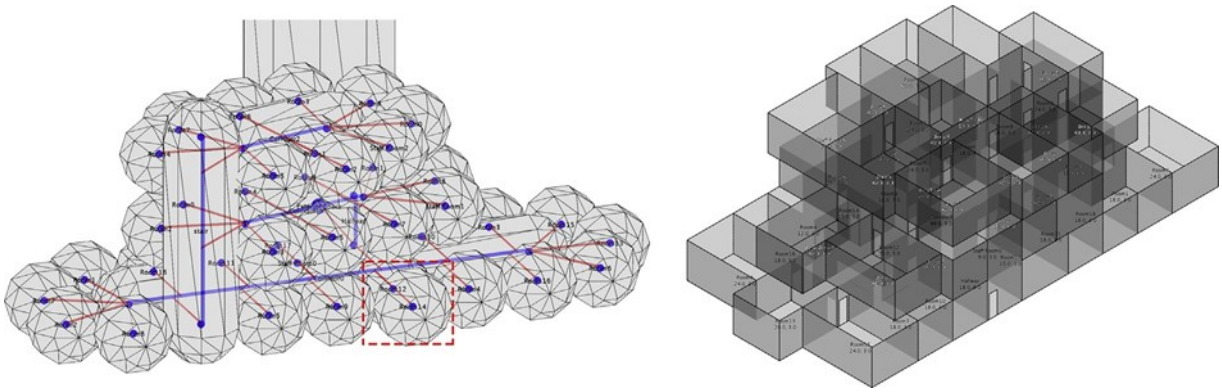

Figure 4: Generated multi-story layouts, Source: [11]

The shape grammar approach offers a promising bridge between generative design and BIM production. By encoding design rules that can be automatically converted to BIM elements, this method maintains design quality while enabling automation [3]. The integration with ArchiCAD through Grasshopper demonstrates that rule-based generation can produce construction-ready documentation. Yet, the requirement for architects to manually define shape grammar rules for each project type limits scalability and accessibility.

2.3.2 COMMERCIAL SOLUTIONS AND THEIR LIMITATIONS

The commercial landscape reveals both industry recognition of the automation opportunity and the limitations of current approaches. TestFit has gained traction for early-stage feasibility studies, enabling rapid evaluation of site utilization and unit mix optimization. Its strength lies in integrating zoning

regulations and financial metrics, providing instant feedback on design feasibility [12] (Figure 5). However, TestFit's focus on standardized solutions and limited flexibility for unique architectural expressions highlights a fundamental tension: the trade-off between automation efficiency and design creativity.

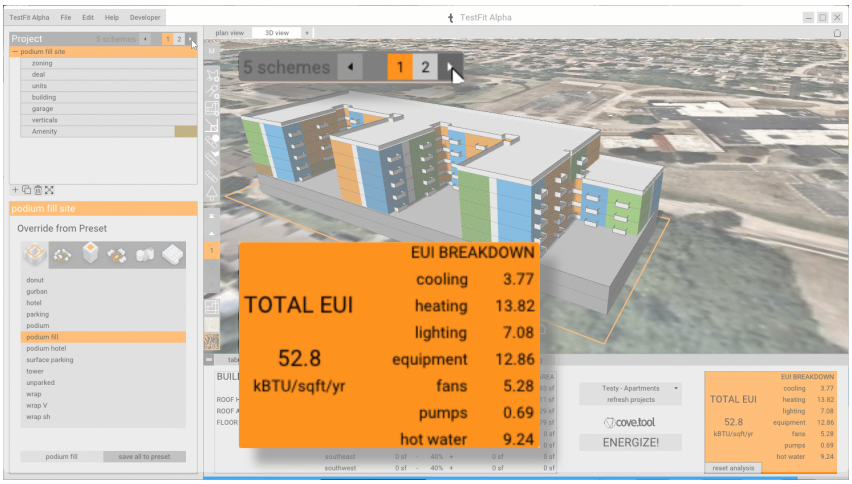

Figure 5: Testfit, Source: [12]

Skema.ai takes a different approach, emphasizing the reuse and adaptation of existing design knowledge rather than generation from scratch [13] (Figure 6). By learning from a firm's previous Revit projects and creating a "Design Catalog" of modular components, Skema addresses the important issue of maintaining design consistency while accelerating production. The platform's two-way integration with SketchUp and Revit demonstrates an understanding of existing workflows. Yet, its heavy dependence on Revit and the need for extensive prior project data limit accessibility for smaller firms or those beginning their digital transformation.

Figure 6: Skema.ai, Source: [13]

The Roombook extension for Revit illustrates another category of tools that enhance rather than replace manual design processes [14]. While not generating floor plans, Roombook's detailed quantification capabilities address the downstream needs of generated designs. Its limitations with linked models and the need for meticulous modelling practices reveal the challenges of working within existing BIM platforms' constraints.

Recent entrants in the commercial space have embraced AI and machine learning as the primary driver for floor plan generation, representing a shift from rule-based parametric systems to data-driven approaches. Maket.ai exemplifies this new generation of tools, offering AI-powered instant generation of residential floor plans with promises of natural language input capabilities. The platform allows users to specify constraints through parameters and generates hundreds of variations instantly, with integrated visualization in 3D environments [15].

Similarly, platforms like Homestyler, Planner 5D, and CamPlan leverage AI to convert sketches, photos, or even text descriptions into functional floor plans, dramatically lowering the technical barrier to entry. However, these AI-first solutions reveal new limitations: while they excel at rapid generation and visualization, they typically lack robust building code verification, produce generic solutions that require extensive manual refinement for professional use, and offer limited control over specific architectural requirements. The promise of natural language input remains largely unfulfilled. Most systems still require structured parameter input or rely on simple text prompts that cannot capture the complexity of architectural specifications. Furthermore, the BIM integration remains superficial; these tools generate visual representations rather than information-rich models required for construction documentation. This proliferation of AI-powered tools demonstrates market demand for accessible floor plan generation while simultaneously highlighting the persistent gap between consumer-oriented visualization tools and professional architectural requirements.

2.4 INTERFACE PARADIGMS AND USER INTERACTION

2.4.1 THE COMPLEXITY BARRIER

A critical but often overlooked aspect of automated design systems is the interface through which users specify requirements and interact with the generation process. Most academic systems require users to define constraints through complex parameter sets or programming interfaces, creating a significant barrier to adoption [16].

The Mixed Integer Quadratic Programming (MIQP) approach demonstrates this challenge clearly [16]. While the system generates diverse layouts from residential apartments to shopping malls with impressive speed, orders of magnitude faster than previous methods, users must formulate their requirements as mathematical constraints. This requirement for mathematical formulation excludes most

architects and clients from direct system use, necessitating technical intermediaries that slow the design process.

Procedural generation from building sketches offers a more intuitive input method, allowing users to draw building outlines that are automatically converted to floor plans [9] (Figure 7). The system achieves 98.75% accuracy in wall detection and 94.27% for openings, generating diverse layouts in milliseconds. This sketch-based approach aligns better with architectural thinking, yet it still requires users to fully specify building geometry upfront, limiting exploration and iteration.

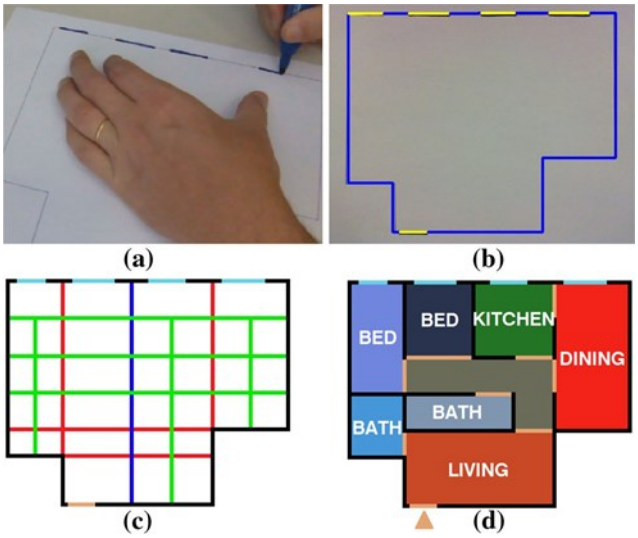

Figure 7: Procedural Generation, Source: [9]

2.4.2 THE PROMISE OF NATURAL LANGUAGE

The potential for natural language interfaces to democratize access to automated design tools remains largely unexplored in the floor plan generation literature. While recent advances in large language models have demonstrated remarkable capabilities in understanding and generating human language, their application to architectural design specifications has received minimal attention [17]. Natural language could enable users to express requirements in familiar terms: "create a two-bedroom apartment with good natural lighting and an open kitchen", rather than through abstract parameters or geometric constraints.

The challenge lies not merely in parsing natural language but in translating ambiguous human expressions into precise geometric operations. Architectural language is rich with implicit knowledge: "good natural lighting" implies specific window-to-floor ratios, orientation preferences, and spatial arrangements that vary by cultural context and building type. "Open kitchen" suggests not just the absence of walls but specific sight lines, circulation patterns, and social relationships between spaces.

2.5 THE STATE OF CURRENT PRACTICE

2.5.1 WHY AUTOMATION HASN'T TRANSFORMED ARCHITECTURE

Despite decades of research and numerous commercial offerings, automated floor plan generation has not transformed architectural practice as once predicted.

First, the fragmentation of solutions means architects must cobble together multiple tools to complete even simple projects. One tool might excel at space planning but lack BIM integration; another might offer BIM connectivity but ignore building codes; yet another might check compliance but require manual layout creation [18]. This fragmentation creates workflow complexity that often exceeds the manual processes these tools aim to replace.

Second, the inability to handle edge cases and unique requirements means automated tools are relegated to preliminary studies rather than integrated into the complete design process. Architecture inherently involves responding to unique site conditions, client preferences, and cultural contexts that resist standardization. When automated systems cannot accommodate these variations, architects must either compromise design quality or abandon automation entirely.

Third, the lack of transparency in automated decision-making creates distrust among professionals trained to take responsibility for their designs. When a system generates a layout through opaque optimization processes or machine learning models, architects cannot verify that all requirements have been met or understand why specific decisions were made. This "black box" problem is particularly acute for building code compliance, where architects bear legal responsibility for violations.

2.5.2 THE INTEGRATION CHALLENGE

The literature reveals a consistent pattern: successful automation requires not just algorithmic sophistication but careful integration with existing tools, workflows, and professional practices. The most promising approaches recognize that automation should augment rather than replace human expertise, providing intelligent assistance while maintaining designer control.

The evaluation of AI-generated residential floor plans highlights the importance of comprehensive metrics that go beyond simple geometric validity [19]. Metrics must consider functional performance, aesthetic quality, code compliance, and constructability, a multi-dimensional evaluation that mirrors the complex judgments architects make. Yet, current systems typically optimize for one or two metrics while ignoring others, producing technically valid but practically inadequate solutions.

Recent work has begun to address this evaluation gap more systematically. Zeng et al. developed RFP-A (Residential Floor Plan Assessment), a hierarchical framework evaluating generated plans through room count compliance, spatial connectivity, room locations, and geometric features [19]. Testing six-

generation models revealed that only HouseDiffusion and FloorplanDiffusion achieved over 90% accuracy in basic room number compliance, while others scored below 60% (Figure 8). However, RFP-A remains a post-generation evaluation tool that cannot guide generation toward compliance or handle building codes across jurisdictions. It evaluates completed designs rather than integrating requirements into the generation process, highlighting the continued need for systems that ensure compliance during generation rather than identifying failures afterward.

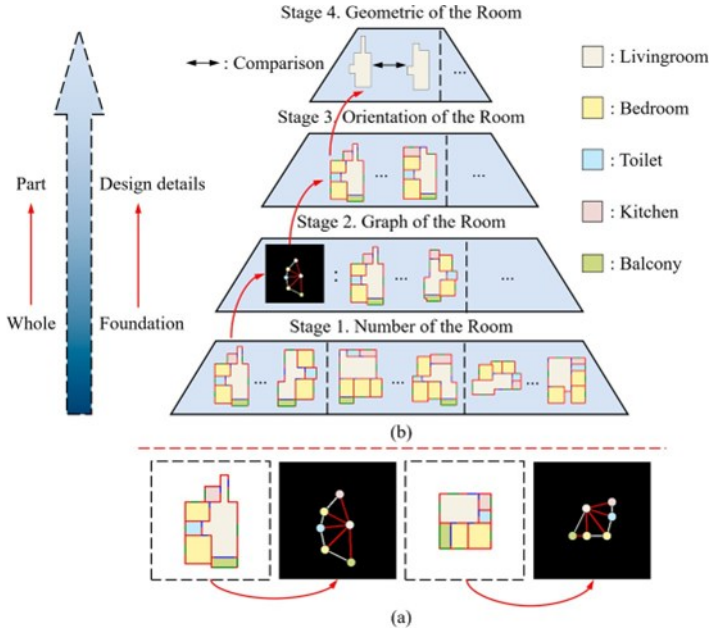

Figure 8: RFP-A Evolution metrics structure diagram, Source: [19]

2.6 IDENTIFYING THE RESEARCH GAP

2.6.1 THE CONVERGENCE OPPORTUNITY

The literature review reveals not a single gap but a convergence of multiple gaps that, when addressed together, could finally realize the promise of automated floor plan BIM model generation. These gaps are:

1. Natural Language Specification: No existing system enables users to specify complex architectural requirements through natural language, despite this being how clients and architects naturally communicate about space. The absence of natural language interfaces maintains a barrier between user intent and system execution that limits adoption.

2. Integrated Building Code Compliance: Current systems either ignore building codes or implement simplified checks that fail to capture regulatory complexity. The lack of comprehensive, jurisdiction-

specific code checking means generated designs require extensive manual verification, negating efficiency gains.

3. Direct BIM Generation: The disconnect between layout generation and BIM production creates a workflow gap that requires manual bridging. Without native BIM output, automated layouts cannot integrate with downstream processes for documentation, coordination, and construction.

4. Real-time Modification: Many existing systems treat generation as a one-time process rather than supporting the iterative refinement that characterizes architectural design. The inability to quickly modify and regenerate designs based on feedback limits practical utility.

5. Transparent Decision-Making: The opacity of current optimization and machine learning approaches creates distrust and liability concerns. Without understanding how decisions are made, architects cannot confidently use automated systems for professional projects.

2.6.2 TOWARD AN INTEGRATED SOLUTION

The path forward requires not only incremental improvement of existing approaches but also a fundamental rethinking of how automated design systems should interact with users and integrate with professional workflows. The ideal system would:

- Accept requirements in natural language while maintaining precision
- Continuously ensure building code compliance throughout generations
- Produce fully parametric BIM models ready for documentation
- Support iterative refinement through clear modification commands
- Provide transparent reporting of decisions and trade-offs
- Integrate seamlessly with existing BIM platforms

Such a system would transform automated floor plan generation from an academic curiosity or preliminary design tool into an integral part of professional architectural practice. By addressing the convergence of gaps identified in this review, it would finally bridge the chasm between theoretical capability and practical utility.

2.7 CONCLUSION

This literature review has traced the evolution of automated floor plan generation from early constraint-based approaches through current machine learning systems, revealing both remarkable algorithmic achievements and persistent practical limitations. While academic research has demonstrated that computers can generate valid floor plans quickly, and commercial tools have shown that automation can

accelerate specific design tasks, no existing solution provides the comprehensive capabilities required for professional architectural practice.

The critical gaps: natural language interaction, building code compliance, BIM integration, iterative modification, and decision transparency are not independent challenges but interconnected aspects of a single problem: how to create automated design tools that architects can confidently use for real projects. The literature suggests that addressing these gaps requires not just technical innovation but careful attention to professional workflows, regulatory requirements, and the nuanced relationship between human creativity and machine efficiency.

The convergence of recent advances in natural language processing, parametric modelling, and BIM technology creates an unprecedented opportunity to finally realize the long-standing vision of automated floor plan generation. The system proposed in this thesis, which integrates natural language interfaces with building code compliance and direct BIM generation, represents an attempt to bridge these critical gaps and transform automated design from an auxiliary tool into an essential component of architectural practice. The following chapter details the architecture and implementation of this integrated solution, demonstrating how the gaps identified in this literature review can be systematically addressed through careful system design and implementation.

3 METHODOLOGY

3.1 RESEARCH FRAMEWORK

The development of the Parametric Generation of Standardized Spaces required a methodological approach that could bridge the gap between theoretical possibilities identified in the literature and practical implementation constraints of professional practice. The research adopted a design science framework, which emphasizes creating innovative artifacts that solve real-world problems while contributing to theoretical knowledge.

The problem decomposition strategy began with identifying the fundamental disconnect between stakeholder needs and existing solutions. Rather than approaching floor plan generation as a purely algorithmic challenge, as much of the academic literature does, or as a simple drafting acceleration tool, as commercial solutions typically position themselves, this research reconceptualized the problem as a translation challenge. The core question became: how can natural human spatial thinking be accurately translated into precise BIM operations while maintaining professional standards and regulatory compliance?

This reconceptualization led to decomposing the challenge into three interconnected translation problems. First, the linguistic translation problem involved converting natural language specifications into structured parametric operations. Second, the regulatory translation problem required transforming textual building codes into computational constraints. Third, the geometric translation problem demanded converting abstract spatial relationships into constructible BIM elements. Each translation layer would require distinct methodological approaches while maintaining coherent information flow between them.

The integration strategy explicitly rejected the traditional waterfall approach, where each component would be developed in isolation, then integrated. Instead, the methodology employed continuous integration principles, where each component's development informed and constrained the others. This approach ensured that solutions remained practically viable rather than theoretically elegant but unusable. For instance, the natural language processing capabilities were developed in tandem with the parametric engine's requirements, ensuring that parsed specifications could be directly executed rather than requiring intermediate transformation steps.

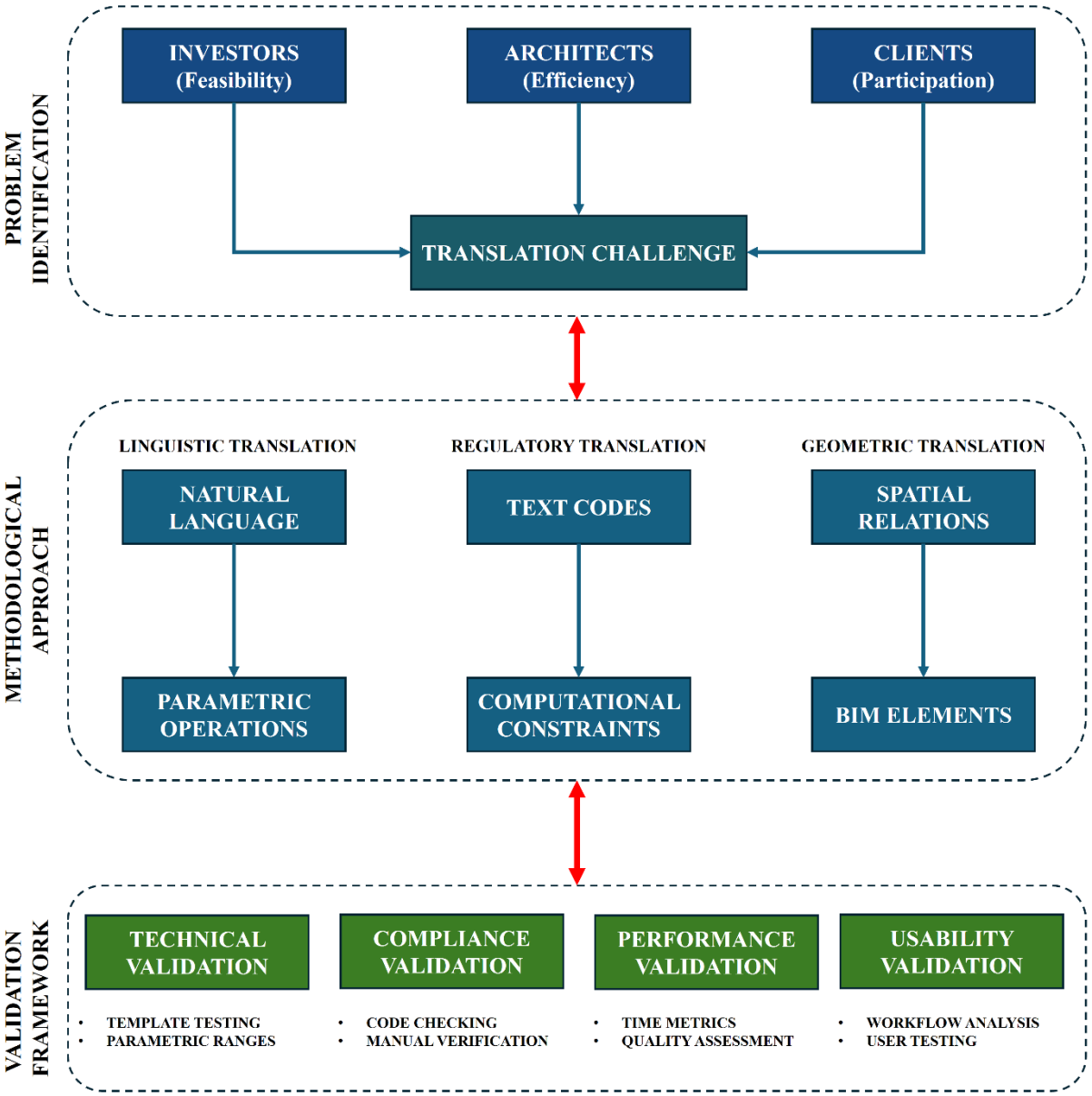

Figure 9: Methodological Framework for Parametric Generation System Development

The framework illustrates the three-layered approach: stakeholder problem identification, parallel translation methodologies, and comprehensive validation. Bidirectional arrows indicate iterative refinement based on validation feedback.

3.2 SYSTEM DEVELOPMENT APPROACH

The development methodology emerged from a critical analysis of why existing automated design tools fail to achieve widespread adoption despite technical sophistication. The literature review revealed that solutions typically optimize for single user groups—either providing powerful features that only specialists can operate, or simplified interfaces that produce inadequate professional outputs [10]. This

research, therefore, adopted a multi-stakeholder development approach that prioritized concurrent satisfaction of diverse user needs rather than sequential feature addition.

Requirements gathering began by analysing actual workflow patterns in architectural practices and real estate development firms. Rather than relying on assumed user needs, the methodology examined specific pain points through documented project timelines. Analysis of residential project documentation from architectural firms revealed that initial concept generation typically consumed 2-4 weeks, with an additional 4-8 weeks for detailed development [1]. Within this timeline, architects reported spending approximately 60% of their time on repetitive technical tasks such as dimension verification and code compliance checking, leaving limited time for design exploration and client interaction [20]. This empirical understanding of current practice inefficiencies directly informed system requirements.

Three distinct user archetypes emerged with different interaction needs. Investors required rapid generation of multiple options for feasibility analysis, but had no interest in technical BIM details. Architects needed professional-grade outputs with precise control over modifications while avoiding repetitive manual tasks. Clients desired meaningful participation in design decisions without requiring technical knowledge. These conflicting requirements could not be satisfied through a single interface paradigm, leading to the decision to separate user interaction from BIM generation through an intermediate command layer.

The iterative development strategy rejected the waterfall model common in BIM tool development, where complete specifications precede implementation [21]. Instead, the methodology employed rapid prototyping cycles that tested core assumptions before committing to technical implementations. The first prototype tested whether file-based communication could provide sufficient responsiveness for interactive design. Using simple Python scripts writing JSON files that triggered basic Revit operations, this prototype validated that file-monitoring could achieve sub-second response times adequate for user interaction. This early validation prevented investment in more complex communication architectures that would have provided marginal benefits while increasing deployment complexity.

Component isolation emerged as a fundamental architectural principle based on analysis of existing tool limitations. TestFit's monolithic architecture makes it difficult to adapt to non-standard projects [12], while Skema.ai's tight Revit integration limits accessibility for non-technical users [13]. The methodology, therefore, mandated strict separation between user interface logic, command processing, and BIM operations. This separation enabled independent evolution of components—the natural language processing could be refined based on user feedback without modifying BIM generation logic, while new building codes could be added without touching the user interface.

The decision to implement file-based communication rather than direct API calls or network protocols stemmed from deployment reality analysis. Enterprise IT policies frequently restrict network socket creation and inter-process communication, requiring extensive security reviews for systems using such approaches [22]. File operations within user directories require no special permissions and work consistently across diverse security contexts. The JSON format was selected for its human readability, enabling manual debugging when necessary, and its universal support across Python and .NET environments without additional dependencies [23].

The progressive refinement approach began with core functionality before adding sophistication. Initial development focused on generating simple rectangular rooms with fixed dimensions, validating the complete pipeline from user input to BIM creation. Only after this foundation proved stable were parametric relationships introduced, followed by building code compliance, and finally natural language processing. This incremental complexity management ensured that each layer built upon proven functionality rather than compounding untested assumptions.

Testing integration occurred continuously rather than as a final phase. Each development iteration included validation with representative user tasks derived from actual project requirements. For instance, when implementing area adjustment capabilities, test cases came from documented apartment projects where total area constraints drove design decisions [24]. This reality-grounded testing revealed issues that abstract test cases would miss, such as the need to maintain minimum room dimensions while scaling to match total area requirements—a common real-world constraint that purely algorithmic approaches often violate.

3.3 TECHNICAL IMPLEMENTATION STRATEGY

The selection of PyRevit [25] inside Revit [26] as the implementation platform represented a strategic methodological decision, balancing rapid development needs with long-term maintainability. Traditional C# Revit add-ins would have required lengthy compile-build-deploy cycles for each iteration, fundamentally limiting the ability to explore alternative approaches quickly. PyRevit's interpreted environment enabled immediate testing of algorithmic variations, crucial for developing complex features like building code compliance checking, where requirements emerged through experimentation rather than upfront specification.

The natural language processing development followed a deliberate methodology of building domain-specific capability rather than leveraging general-purpose AI services. This decision stemmed from multiple considerations: privacy concerns in professional practice where project data cannot leave corporate networks, the need for predictable and debuggable behaviour in professional tools, and the observation that architectural language is sufficiently specialized that general models perform poorly.

The methodology involved systematic analysis of how architects and clients actually describe spaces, building a corpus of expressions that the system needed to understand.

The template-based approach emerged through methodological experimentation with different generation strategies. Initial attempts using pure algorithmic generation, while theoretically promising, faced the challenge identified across the literature, where optimization-based methods produce layouts that satisfy dimensional constraints but lack the spatial qualities that emerge from architectural experience [18]. Constraint-based optimization could satisfy requirements, but required users to specify dozens of parameters, defeating the accessibility goal. The template methodology recognized that most apartment designs follow established patterns that encode centuries of architectural knowledge. By parameterizing these patterns rather than generating them from scratch, the system could ensure quality while maintaining flexibility. Furthermore, the template library was designed as an expandable repository where architects from different firms could contribute their proven designs, creating a growing knowledge base that increases both the system's capability and credibility. This collaborative expansion model transforms the system from a static tool into an evolving platform that captures diverse architectural expertise while maintaining quality through pre-validation of each template

Building code data extraction followed a rigorous methodology designed to ensure accuracy and completeness. Rather than relying on secondary sources or interpretations, the research went directly to official government publications for each of the ten countries. The extraction process involved multiple passes: initial identification of relevant sections, systematic extraction of quantifiable requirements, cross-validation against published architectural guidelines, and verification through test cases based on actual approved buildings. This methodological rigor was essential given that code compliance errors could have legal implications for system users.

The structuring of building code data into computational formats required developing a specialized methodology for handling regulatory ambiguity. Building codes often use qualitative language like "adequate" or "sufficient" that resists direct quantification. The methodology involved identifying the most restrictive reasonable interpretation, ensuring generated designs would satisfy even conservative reviewers. Where codes specified performance requirements rather than prescriptive dimensions, the system incorporated established architectural standards as proxy measures, documenting these interpretations for transparency.

3.4 VALIDATION & TESTING METHOD

The validation methodology employed a systematic approach to ensure the reliability and effectiveness of the parametric generation system. Given the research context of a master's thesis with limited resources for extensive field testing, the validation strategy focused on rigorous technical verification and proof-of-concept demonstration rather than large-scale empirical studies.

Template validation began with geometric integrity checking to ensure that each template could function correctly across its intended parametric range. Each template underwent systematic testing where global parameters were adjusted from minimum values (based on building code requirements) to maximum feasible dimensions. This testing verified that room boundaries remained properly closed, walls maintained proper connections at corners, and circulation paths stayed unobstructed. The validation process particularly focused on critical transition points where parametric adjustments might cause geometric failures, such as when room dimensions approach minimum code requirements or when total area constraints force proportional scaling.

Building code compliance verification adopted a systematic approach to validate the computational interpretation of regulatory requirements. The verification process compared the system's encoded building code constraints against the original regulatory documents for each of the ten countries. For each room type, minimum dimensions were cross-referenced with official publications to ensure accurate interpretation. The system's ability to enforce these requirements was tested through deliberate attempts to generate non-compliant designs, verifying that the compliance checking mechanisms properly prevented or flagged violations. This validation confirmed that the system correctly applied requirements such as minimum room areas, corridor areas, total apartment areas, and ceiling height as specified in each country's building code.

The natural language processing validation focused on ensuring accurate interpretation of common architectural specifications. A test suite of typical user inputs was developed, ranging from simple commands like "create a 2-bedroom apartment" to more complex specifications involving multiple parameters. Each test case was processed through the natural language interpreter, and the resulting JSON command structure was verified against expected outputs. This validation revealed the importance of handling variations in how users express the same requirement, leading to refinements in the pattern-matching algorithms to accommodate different phrasings of identical specifications.

System workflow validation tested the complete pipeline from user input to BIM model generation. The testing followed typical use scenarios: creating new apartment layouts, adjusting dimensions to meet specific requirements, and modifying generated models through natural language commands. Each workflow was executed multiple times to ensure consistent behavior and identify any synchronization issues between the external UI and the PyRevit plugin. The file-based communication protocol proved robust, with the JSON command files successfully transmitting all necessary parameters and the status update mechanism providing reliable feedback about operation completion.

The practical performance assessment, conducted through the author's direct testing as a practicing architect, demonstrated the system's capability to generate complete BIM models within seconds of command execution. A typical 2-bedroom apartment, which would traditionally require hours to model manually in Revit, could be generated with proper wall connections, door and window placements, and

room definitions in under one minute from initial specification to completed model. This dramatic time reduction was consistent across different apartment configurations, from studio units to four-bedroom layouts, validating the system's efficiency claims.

Quality assessment of generated models examined their suitability for professional use. The generated BIM models included proper element classification, with walls, doors, and windows correctly categorized and tagged with appropriate metadata. Room elements were automatically created with accurate area calculations and proper boundary detection. The parametric relationships embedded in the templates remained functional in the generated models, allowing for subsequent manual adjustments if needed.

The modification capability testing verified that the system's natural language commands could successfully alter generated models. Tests included flipping door orientations, changing window families, and adjusting wall types through commands referencing the automatic element numbering system. The success of these modifications demonstrated that the system maintained proper element tracking and could execute targeted changes without requiring users to navigate Revit's native selection tools.

Limitations of the validation approach must be acknowledged. The testing was primarily conducted by the system developer, which, while providing deep technical verification, lacks the breadth of independent user testing. The absence of testing in commercial architectural firms means that performance under production conditions with varying project requirements remains unverified. Additionally, the testing focused on successful generation scenarios rather than systematically exploring failure modes and edge cases that might emerge in professional practice. These limitations are acceptable within the scope of a master's thesis proof-of-concept but would require addressing before commercial deployment.

3.5 EVALUATION FRAMEWORK

The evaluation framework established criteria for assessing the system's effectiveness within the constraints of a master's thesis project. Rather than extensive empirical testing with multiple users, the framework focused on demonstrable capabilities and technical validation that could be rigorously verified through systematic testing.

Time efficiency evaluation compared the automated generation process against traditional manual modelling workflows. The system consistently generated equivalent models in under one minute, representing a time reduction of over 98% from the traditional workflow [1]. This comparison, while limited to single-user testing, provides a concrete benchmark for the system's efficiency gains.

Code compliance assessment examined whether generated models met the requirements encoded in the ten building codes. Each generated apartment was checked against the relevant building code's minimum requirements for room dimensions, total areas, and ceiling heights. The system successfully prevented the generation of non-compliant designs by refusing to process specifications that violated minimum requirements and automatically adjusting dimensions when necessary to achieve compliance. This validation confirmed that the building code integration functioned as designed, though comprehensive testing across all possible edge cases was beyond the thesis scope.

Output quality evaluation focused on the professional usability of generated models. The assessment criteria included proper element classification (walls as walls, doors as doors), correct spatial relationships (rooms properly bounded, doors connecting spaces), and preservation of parametric relationships enabling subsequent modification. Generated models maintained the organization and structure expected in professional practice, with elements properly grouped and named according to Revit conventions.

System reliability was assessed through repeated execution of standard workflows. The same apartment specifications were processed multiple times to verify consistent output. The file-based communication protocol successfully transmitted commands from the external UI to Revit without data loss or corruption across all test cases. The element numbering system correctly labelled all components, enabling reliable selection for modifications.

4 PARAMETRIC GENERATION SYSTEM ARCHITECTURE

4.1 INTRODUCTION

This chapter presents the comprehensive system architecture of the Parametric Generation of Standardized Spaces. This innovative BIM automation solution transforms the traditional approach to architectural design through intelligent parametric apartment modelling and building code compliance. The system represents a paradigm shift in how architects and designers interact with BIM software, introducing a streamlined workflow that combines natural language input capabilities with dropdown-based quick selection, all while maintaining strict adherence to international building codes and professional standards.

The architecture has been designed to address the critical gaps identified in existing solutions: the lack of integrated building code compliance, the complexity of parametric modelling interfaces, and the disconnect between user intent and BIM implementation. Unlike conventional approaches that require extensive manual parameter manipulation or complex scripting, this system provides an intuitive interface that translates high-level requirements directly into fully compliant BIM models within Autodesk Revit [26].

The development of this system required careful consideration of multiple technical challenges: how to effectively parse and interpret natural language inputs without relying on external AI services, how to maintain building code compliance across multiple jurisdictions, how to create a robust file-based communication system between the UI and Revit, and how to generate complete BIM models from simplified template representations. This chapter details the architectural decisions and implementation strategies employed to address these challenges.

4.2 SYSTEM OVERVIEW

4.2.1 HIGH-LEVEL ARCHITECTURE

The Parametric Generation system consists of three primary components that work together to enable automated BIM generation from user specifications:

i. **External User Interface Application:** A standalone desktop application that provides both natural language input capabilities and structured dropdown menus for requirement specification. This interface handles all user interactions and generates structured command files for processing by Revit.
ii. **Revit Plugin System:** A native Revit add-in developed in C#/.NET that reads command files, manages template selection, performs parametric adjustments, ensures building code

compliance, and generates complete 3D BIM model of an Apartment with properly numbered elements.

iii. **File-Based Communication Layer:** A JSON-based file system that enables reliable communication from the UI to Revit, using structured command queues and status updates to coordinate the generation process.

The system architecture deliberately avoids complex network protocols or external dependencies, instead utilizing a robust file-based approach that ensures reliability and simplifies deployment [27].

4.2.2 WORKFLOW OVERVIEW

The complete workflow follows a carefully orchestrated sequence that ensures user requirements are accurately translated into compliant BIM models (Figure 10).

4.2.2.1 Step 1: Requirement Specification using external UI

Users interact with the external UI application to specify their requirements through two complementary input methods:

- Natural language input for complex or nuanced requirements
- Dropdown menus for quick selection of standard options

4.2.2.2 Step 2: JSON Generation by Command Queue

The UI application processes user inputs and generates a structured command queue, saved as a JSON file in a designated directory that Revit monitors.

4.2.2.3 Step 3: Revit Processing

When users click "Process All Commands" in the Revit plugin, the system:

- Reads the JSON file’s command queue file
- Selects appropriate templates based on bedroom/bathroom configuration
- Applies building code requirements for the specified country
- Adjusts parameters to meet area and dimensional requirements
- Generates the complete 3D BIM model

4.2.2.4 Step 4: Element Identification

The generated model includes numbered elements (walls, doors, windows) that enable subsequent modifications through natural language commands.

4.2.2.5 Step 5: Modification Support

Users can request modifications by referring to numbered elements, such as "flip door D1" or "change family of window W3," which the system processes to update the model accordingly.

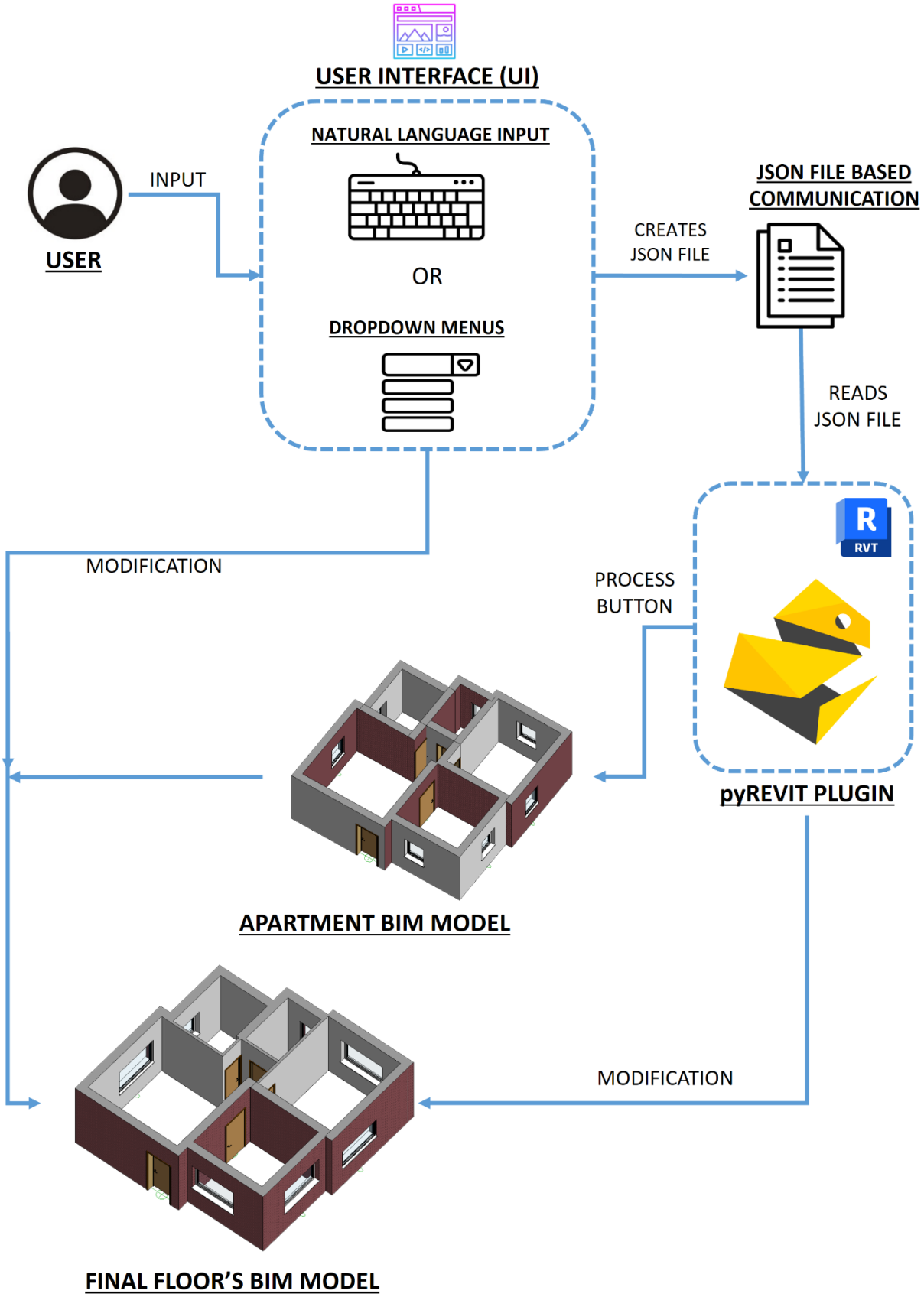

Figure 10: Complete Parametric Generation Workflow

4.3 EXTERNAL USER INTERFACE APPLICATION

The External User Interface Application serves as the primary interaction point between users and the BIM generation system. Developed as a standalone desktop application using Python [28], this interface provides an intuitive environment for specifying architectural requirements without requiring deep knowledge of Revit or parametric modelling. The application transforms complex architectural specifications into structured commands that can be processed by the Revit plugin, bridging the gap between human intent and machine execution.

4.3.1 INTERFACE DESIGN & LAYOUT

The user interface has been meticulously designed to balance functionality with usability, providing multiple input modalities to accommodate different user preferences and expertise levels. The interface architecture follows a hierarchical organization that guides users through the specification process while maintaining flexibility for advanced users who prefer direct control.

The application window is systematically organized into seven functional zones, each serving a specific purpose in the requirement specification workflow (Figure 11):

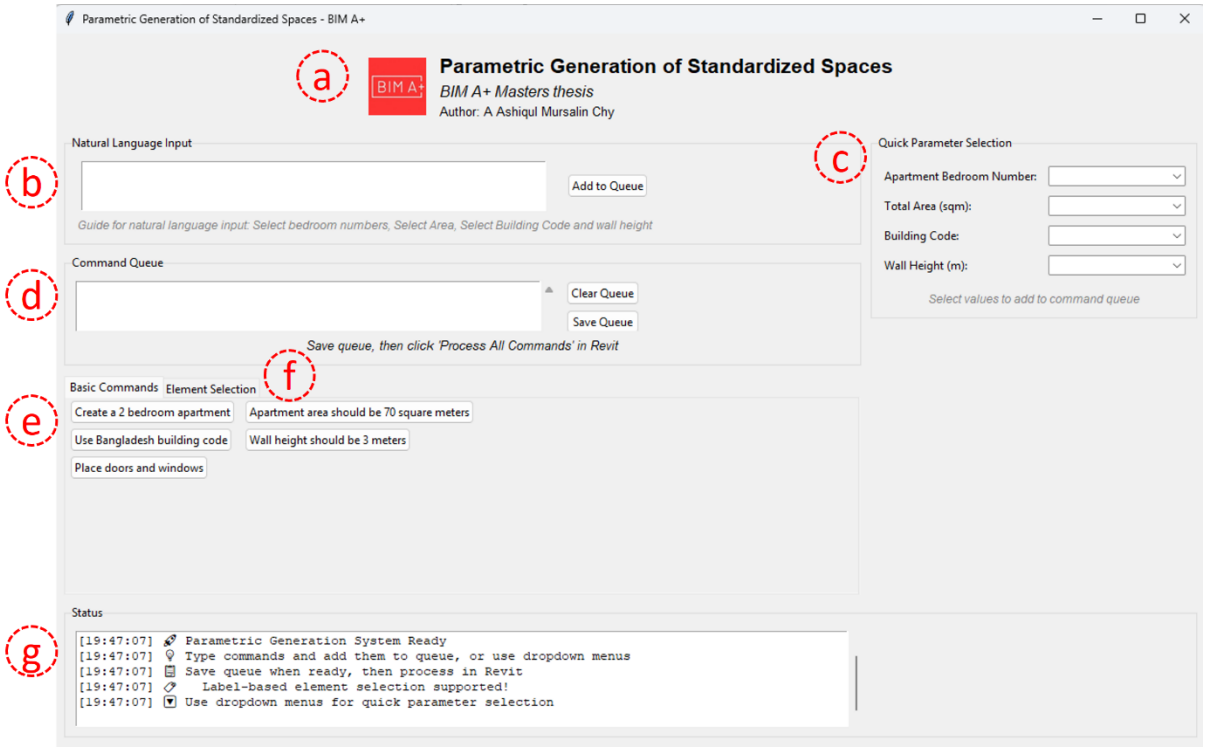

Figure 11: External User Interface Design & Layout

a. Header Section: The header establishes the professional and academic context of the application:

- Title: "Parametric Generation of Standardized Spaces" clearly identifies the application's purpose
- Subtitle: "BIM A+ Master's thesis" acknowledges the academic foundation
- Author attribution: "A Ashiqul Mursalin Chy" provides proper credit
- BIM A+ logo reinforces institutional affiliation and professional credibility

b. Natural Language Input Panel: This panel represents the primary innovation in user interaction, featuring a large text area that accepts free-form requirement specifications. Users can express their needs in plain English, making the system accessible to non-technical stakeholders. The interface supports both comprehensive single-line specifications and incremental multi-line inputs:

- Single comprehensive input: "Create a 2 bedroom apartment with 70 square meters total area using the Slovenian building code with 3-meter wall height"
- Incremental specification: Users can build requirements step by step:
  "Create a 2 bedroom apartment"
  "Area will be 75 sqm"
  "Follow Portugal Building Code"
  "Set wall height to 3 meters"

Guide text above the input area provides clear instructions: "Guide for natural language input: Select bedroom numbers, Select Area, Select Building Code, and wall height," ensuring users understand the expected input format without constraining their expression style.

c. Quick Parameter Selection Panel: For users who prefer structured input or need to make quick adjustments, the dropdown panel provides rapid parameter selection:

- Apartment Bedroom Number: Predefined options (1BR, 2BR, 3BR) with Custom option for non-standard configurations
- Total Area (sqm): Common sizes (50, 60, 70, 80, 90, 100 sqm) with Custom input for specific requirements
- Building Code: Dropdown listing all 10 supported countries (Slovenia, USA, Germany, France, Spain, Australia, Netherlands, Norway, Portugal, Bangladesh)
- Wall Height (m): Standard heights (2.5, 2.7, 3.0, 3.3, 3.5 m) with Custom option for special cases

The selection panel includes instructional text: "Select values to add to command queue," clarifying the relationship between selection and queue management.

d. Command Queue Section: The command queue provides transparency and control over the generation process:

- Visual Display: Shows all pending commands in a scrollable list, allowing users to review their specifications before processing
- Add to Queue Button: Transfers natural language input or dropdown selections to the queue
- Clear Queue Button: Enables users to reset and start fresh if needed
- Save Queue Button: Writes the command queue to a JSON file for Revit processing
- Instructional Note: "Save queue, then click 'Process All Commands' in Revit" guides users through the two-step execution process

e. Basic Commands Panel: Pre-defined command buttons accelerate common operations:

- "Create a 2-bedroom apartment" - Instantly adds a standard apartment creation command
- "Apartment area should be 70 square meters" - Quickly specifies a common area requirement
- "Use Bangladesh building code" - Rapid building code selection
- "Wall height should be 3 meters" - Standard height specification
- "Place doors and windows" - Triggers opening placement algorithm

These buttons reduce typing effort and demonstrate proper command syntax, serving both as shortcuts and instruction examples.

f. Element Selection Tab: This advanced feature enables post-generation modifications:

- Label-based element selection using the numbering system
- Dropdown menus for modification parameters
- Support for operations like flip, move, resize, and replace
- Visual feedback showing selected elements

g. Status Panel: Real-time feedback keeps users informed of system state and processing progress:

- Timestamped messages provide chronological operation history
- Icon indicators showing success, warnings, and errors
- System readiness notifications
- Processing stage updates
- Completion confirmations

Example status messages demonstrate the information hierarchy:

- "[19:47:07] 🚀 Parametric Generation System Ready"

- "[19:47:07] 💡 Type commands and add them to queue, or use dropdown menus"
- "[19:47:07] 📝 Save queue when ready, then process in Revit"
- "[19:47:07] 🔄 Label-based element selection supported!"
- "[19:47:07] ✨ Use dropdown menus for quick parameter selection"

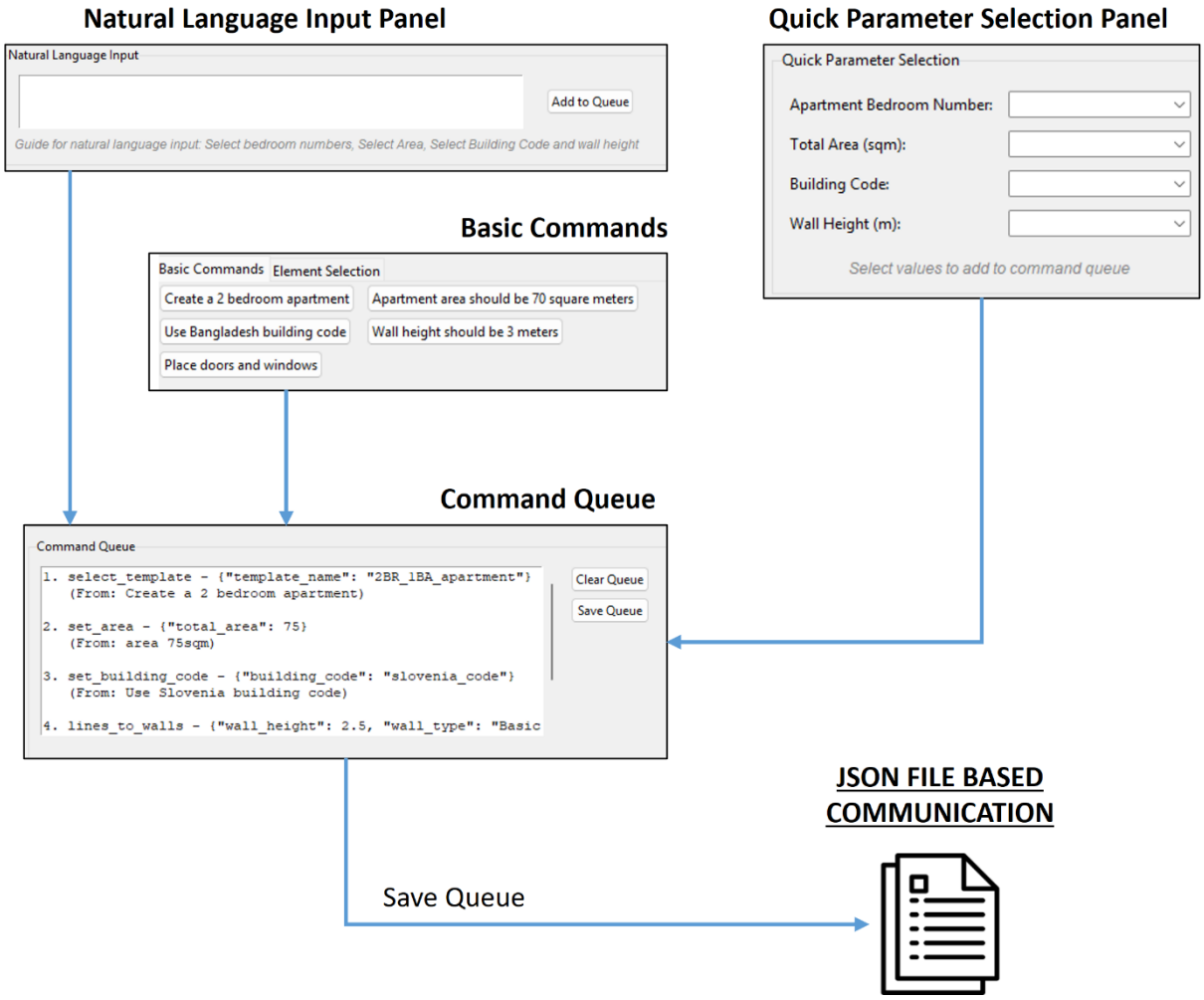

Figure 12: Multiple Input Options for External UI

4.3.2 NATURAL LANGUAGE PROCESSING ENGINE

The custom-built natural language processing engine represents a significant technical achievement of this system, enabling intuitive interaction without dependency on external AI services. The engine employs sophisticated pattern matching and keyword extraction to interpret architectural requirements with high accuracy, transforming human-readable specifications into machine-executable commands.

The engine operates through a multi-layered processing pipeline that progressively refines user input into structured data. At its core, the system maintains a comprehensive architectural vocabulary that maps colloquial expressions to technical specifications. This vocabulary encompasses spatial terminology (bedroom, bathroom, kitchen), quantitative expressions (one, two, few, several), dimensional descriptors (small, medium, large, spacious), and modification qualifiers (approximately, exactly, at least, maximum).

The pattern recognition layer employs regular expressions optimized for architectural specifications [29]. These patterns can identify and extract:

- Room configurations: "two bedroom," "2BR," "2-bedroom," or "two-bed"
- Area specifications: "70 square meters," "750 sq ft," or "about 70 sqm"
- Height requirements: "3 meter ceiling," "10 foot height," or "standard height"
- Building codes: "Slovenian code," "Slovenia regulations," or simply "SI"

Unit conversion is seamlessly integrated into the parsing process [30]. The system automatically detects measurement units and performs necessary conversions to maintain internal consistency. Whether users specify "75 square feet" or "7 square meters," the system normalizes all measurements to metric units for processing while preserving the original unit preference for display.

Context management adds intelligence to the parsing process. The system maintains conversation state, understanding that "make it larger" refers to the previously specified apartment, or that "add another bathroom" modifies the existing configuration rather than creating a new specification. This contextual awareness enables natural, conversational interaction that feels intuitive to users.

```python
class NaturalLanguageParser:
    def __init__(self):
        self.bedroom_patterns = [
            r'(\d+)\s*(?:bed|bedroom|br)',
            r'(\d+)br',
            r'(\w+)\s*bedroom'
        ]
        self.area_patterns = [
            r'(\d+)\s*(?:square meters|sqm|m2|sq\.m)',
            r'(\d+)\s*meters?\s*squared?'
        ]
        self.building_code_patterns = {
            'bangladesh': ['bangladesh', 'bd', 'dhaka'],
            'slovenia': ['slovenia', 'si', 'ljubljana'],
            'usa': ['usa', 'united states', 'america', 'ibc'],
            # ... patterns for other countries
        }
```

```
def parse_input(self, text):
    """Extract parameters from natural language input"""
    parameters = {}

    # Extract bedroom count
    for pattern in self.bedroom_patterns:
        match = re.search(pattern, text.lower())
        if match:
            parameters['bedrooms'] = self._parse_bedroom_count(match.group(1))
            break

    # Extract area
    for pattern in self.area_patterns:
        match = re.search(pattern, text.lower())
        if match:
            parameters['area'] = float(match.group(1))
            break

    # Extract building code
    for code, patterns in self.building_code_patterns.items():
        if any(p in text.lower() for p in patterns):
            parameters['building_code'] = code
            break

    return parameters
```

Figure 13: Command Queue JSON Structure

4.3.2.1 CUSTOM NATURAL LANGUAGE INTERPRETER DEVELOPMENT

The development of the custom natural language interpreter required addressing several key challenges specific to architectural terminology and requirements specification. The interpreter architecture consists of three primary components: the lexical analyzer, the semantic parser, and the requirement synthesizer.

The lexical analyzer tokenizes input text and categorizes each token according to its architectural significance. This process involves more than simple word matching; the analyzer considers word position, surrounding context, and grammatical structure to determine meaning. For instance, "master" before "bedroom" indicates a specific room type, while "master" in isolation might be ambiguous. The analyser maintains a sophisticated understanding of architectural nomenclature variations, recognizing that "primary bedroom," "master bedroom," and "main bedroom" all refer to the same concept.

The semantic parser interprets the tokenized input to extract meaningful requirements. This component implements a rule-based system that understands the relationships between architectural elements. It recognizes that bedrooms require minimum areas, that bathrooms should be adjacent to bedrooms, and that kitchens need ventilation. The parser applies these architectural rules to validate and complete partial specifications, ensuring that extracted requirements are both complete and feasible.

The requirement synthesizer combines extracted elements into a coherent specification. This synthesis process resolves conflicts, applies defaults for missing values, and ensures consistency across all

requirements. If a user specifies "2 bedrooms" without mentioning bathrooms, the synthesizer applies architectural conventions to infer that a 2-bedroom apartment typically includes 1-2 bathrooms. These inferences can be overridden by explicit specifications, maintaining user control while providing intelligent defaults.

Error handling within the interpreter focuses on graceful degradation and user guidance. When the system encounters ambiguous or incomplete input, it doesn't simply fail; instead, it provides specific feedback about what's missing or unclear. For example, if a user types "make it bigger," the system responds with "Please specify what to make bigger: the entire apartment, a specific room, or a particular element?" This interactive clarification process helps users provide complete specifications without frustration.

4.3.2.2 ELEMENT REFERENCE RESOLUTION

Element reference resolution enables users to specify modifications to generated models using natural language descriptions rather than technical identifiers. This capability is crucial for the system's usability, as it allows users to think and communicate in spatial and functional terms rather than abstract numbering systems.

The resolution system implements multiple strategies to identify referenced elements. Direct numeric references like "door D1" or "window W3" provide the most precise identification, directly mapping to the element numbering system. However, the system's strength lies in its ability to resolve more natural references.

Functional references identify elements by their purpose or role in the design. When users say "main entrance door," the system analyzes door properties to identify which one serves as the primary entrance based on its connection to circulation spaces and exterior access. Similarly, "master bedroom window" is resolved by first identifying the master bedroom and then locating windows within that space.

The resolution algorithm employs a scoring system when multiple elements could match a description. Each potential match receives scores based on various criteria: name similarity, spatial proximity, functional role, and user interaction history. The element with the highest composite score is selected, with the system requesting clarification only when scores are too close to determine a clear winner.

4.3.2.3 COMMAND GENERATION FROM NATURAL LANGUAGE

The final stage of natural language processing transforms interpreted requirements into structured commands that the Revit plugin can execute. This transformation process ensures that all necessary parameters are present, values are within acceptable ranges, and the command structure conforms to the expected schema.

Command generation begins with command type determination. The system analyzes the interpreted requirements to decide whether to generate a CREATE_APARTMENT command (for new models), a MODIFY_ELEMENT command (for changes to existing elements), or QUERY command (for information retrieval). This classification drives the subsequent parameter packaging process.

Parameter validation ensures that all generated commands will execute successfully. The system checks that numerical values fall within acceptable ranges, that referenced templates exist, and that building codes are supported. If validation fails, the system attempts automatic correction where possible or requests user clarification for ambiguous cases.

The command structure includes metadata that enables tracking and debugging. Each command receives a unique identifier (UUID), timestamp, and session reference. This metadata proves invaluable for troubleshooting and understanding the sequence of operations that led to a particular model state.

4.3.3 COMMAND QUEUE MANAGEMENT

The command queue system serves as the critical bridge between user intent and system execution, accumulating user specifications into a structured sequence of operations that the Revit plugin can reliably process. This queuing mechanism provides several essential capabilities: it allows users to build complex specifications incrementally, provides transparency into what will be executed, enables review and modification before processing, and ensures reliable command delivery through persistent storage.

The queue operates as a first-in, first-out (FIFO) data structure, preserving the temporal order of user specifications (Figure 14). This ordering is crucial because later commands may depend on earlier ones; for example, a modification command assumes that a creation command has already been executed. The queue manager maintains this temporal integrity while also providing flexibility for users to review and potentially reorder commands before execution.

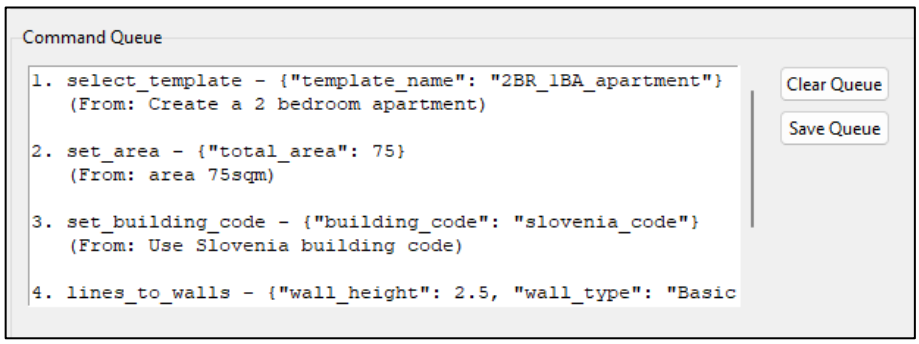

Figure 14: Command Queue in the External UI

Each command in the queue is represented as a structured object containing multiple fields that completely describe the operation. The command identifier (UUID) ensures global uniqueness across

all sessions and systems. The timestamp records when the command was created, valuable for debugging and audit purposes. The command type field (CREATE_APARTMENT, MODIFY_ELEMENT, APPLY_BUILDING_CODE) determines how the Revit plugin will process the instruction. The parameters object contains all necessary data for command execution, with its structure varying based on command type. The status field tracks command lifecycle (pending, processing, completed, error), enabling progress monitoring.

```python
class CommandQueue:
    def __init__(self):
        self.commands = []
        self.queue_file = "                          BIMModelling/queue/commands.json"

    def add_command(self, command_type, parameters):
        """Add a new command to the queue"""
        command = {
            'id': str(uuid.uuid4()),
            'timestamp': datetime.now().isoformat(),
            'type': command_type,
            'parameters': parameters,
            'status': 'pending'
        }
        self.commands.append(command)

    def save_queue(self):
        """Save the command queue to JSON file for Revit processing"""
        queue_data = {
            'version': '1.0',
            'created': datetime.now().isoformat(),
            'commands': self.commands
        }
        with open(self.queue_file, 'w') as f:
            json.dump(queue_data, f, indent=2)
```

Figure 15: Command Queue Serialization

The queue persistence mechanism ensures reliability even in the face of application crashes or system interruptions. When users click "Save Queue," the entire command collection is serialized to JSON format and written to a designated file in the queue directory. This file-based approach provides several advantages over in-memory storage: commands survive application restarts, the queue can be inspected and potentially modified outside the application, and multiple UI instances can potentially share the same queue file.

The JSON serialization process carefully preserves all command details while ensuring compatibility with the Revit plugin's deserialization requirements. The serializer handles special cases like floating-point precision, character encoding for international building codes, and proper escaping of user-provided text that might contain special characters. Version information embedded in the JSON structure ensures forward and backward compatibility as the system evolves.

Queue management also includes safety features to prevent common errors. The system prevents duplicate commands by checking for identical operations already in the queue. It validates command

sequences to ensure logical ordering (can't modify an element before creating it). It warns users about potentially conflicting commands that might produce unexpected results. These safety checks help users avoid frustration and ensure successful model generation.

4.4 FILE-BASED COMMUNICATION SYSTEM

The file-based communication system provides the essential link between the External User Interface and the Revit plugin, enabling reliable data exchange without the complexity and potential issues of network-based protocols. The system implements a bidirectional data flow from the UI to Revit, with status information flowing back through a separate file-based channel.

The choice of file-based communication over network protocols stems from several critical requirements identified during system design. Enterprise environments often restrict network communications, requiring complex firewall configurations for socket-based systems. File operations, in contrast, work reliably in all environments without special permissions or configuration. The approach also simplifies debugging, as communication data persists in files that can be examined when troubleshooting issues. Finally, the atomic nature of file operations ensures data integrity, avoiding the partial message delivery issues that can plague network protocols.

4.4.1 COMMUNICATION PROTOCOL STRUCTURE

The communication protocol defines a structured JSON format that ensures reliable and unambiguous data exchange between system components [23]. This protocol has been designed with emphasis on clarity, extensibility, and error resilience, incorporating versioning support and comprehensive metadata to facilitate debugging and system evolution.


```json
{
  "version": "1.0",
  "timestamp": "2025-08-06T14:30:00",
  "session_id": "sess_20250806_143000",
  "commands": [
    {
      "id": "cmd_001",
      "type": "CREATE_APARTMENT",
      "parameters": {
        "template": "2BR_2BA",
        "total_area": 70,
        "area_unit": "square_meters",
        "building_code": "Bangladesh",
        "wall_height": 3.0,
        "height_unit": "meters"
      },
      "status": "pending"
    }
  ]
}
```


Figure 16: Communication Protocol JSON Structure

The protocol structure follows a hierarchical organization that separates metadata from operational data. At the root level, version information ensures compatibility between different releases of system components. The timestamp provides global ordering across all communications, essential for understanding system behaviour during troubleshooting. The session identifier groups related operations, enabling tracking of multi-step workflows and user sessions.

The command array contains the actual operations to be executed, with each command fully self-contained. This design ensures that commands can be processed independently, improving system resilience. If one command fails, others can still be processed successfully. Commands can be retried without needing context from previous operations. The processing order can be optimized if dependencies permit.

Parameter encoding within commands follows strict conventions to prevent ambiguity. Numerical values always include unit specifications, eliminating confusion between metric and imperial measurements. String values use UTF-8 encoding to support international characters in building codes and user-provided text. Boolean flags explicitly use true/false rather than truthy/falsy values. Enumerated values (like command types) use predefined string constants rather than numeric codes.

The protocol includes extensibility provisions that allow for future enhancements without breaking existing implementations. Unknown fields are preserved but ignored, allowing newer UI versions to include additional data that older Revit plugins safely skip. The version field enables conditional processing based on protocol capabilities. Optional fields can be added to commands without breaking existing processors. This forward-thinking design ensures the system can evolve without requiring synchronized updates of all components.

4.4.2 FILE MONITORING SYSTEM

The file monitoring system enables the Revit plugin to detect and respond to new command files without requiring constant polling or user intervention. This reactive approach minimizes resource consumption while ensuring rapid response to user commands. The implementation leverages operating system file system events for efficient monitoring.

The monitoring system initializes during plugin startup, establishing a watch on the designated queue directory. The FileSystemWatcher component [31], provided by the .NET framework, registers for specific file system events that indicate new commands are available. The watcher configuration is carefully tuned to balance responsiveness with stability, filtering for JSON files only to avoid false triggers from temporary files or other data types.

Event debouncing prevents multiple triggers from a single file write operation. When large files are written, the operating system may generate multiple change events as buffers flush to disk. The

monitoring system implements a small delay (typically 100-500 milliseconds) after detecting a change before attempting to read the file. This delay ensures the file write is complete and the file is not locked by the writing process.

The file reading process includes robust error handling to manage various failure scenarios. If a file is locked (still being written), the system implements exponential backoff retry logic. If a file is corrupted or contains invalid JSON, the system logs the error and continues monitoring for valid files. If the queue directory becomes unavailable (network drive disconnection), the system attempts to reestablish monitoring when the directory returns.

The monitor also implements file cleanup to prevent directory bloat over time. After successfully processing a command file, the system either deletes it or moves it to an archive directory, depending on configuration.

Performance optimization ensures the monitoring system doesn't impact Revit's primary operations. File system events are processed asynchronously, preventing UI freezes during file operations. The monitoring thread runs at a lower priority than Revit's main thread. Resource-intensive operations like JSON parsing are deferred until necessary. The system implements circuit breakers that temporarily disable monitoring if error rates exceed thresholds.

4.4.3 STATUS UPDATE MECHANISM

The status update mechanism provides feedback from the Revit plugin to the UI, informing users about command processing progress and results. This feedback channel operates independently from the command channel, implementing a simple but effective file-based protocol for status communication.

Status updates follow a structured format that provides comprehensive information about operational results. Each status update includes the command identifier, linking it to the original command. The status field indicates the current state (processing, completed, or error). Descriptive messages provide human-readable information about the operation. Timestamps enable performance analysis and debugging. Additional data specific to the operation type may be included.

The status file naming convention ensures unique identification and chronological ordering. Files are named with the pattern status_[command_id]_[timestamp].json, preventing conflicts and enabling easy correlation with commands. The UI monitors the status directory for new files, reading and displaying updates as they appear.

Status persistence provides several operational advantages over transient messaging. Status information survives UI restarts, allowing users to check results later. Multiple UI instances can monitor the same

status directory. Status history can be analysed to understand system behaviour patterns. Failed operations can be investigated using preserved status data.

The cleanup mechanism prevents the unlimited accumulation of status files. The UI deletes status files after displaying them to the user. Old status files are periodically purged based on age. Error status files are preserved longer for diagnostic purposes. Archive directories can be configured for long-term status retention if needed.

This file-based approach to status communication maintains the simplicity and reliability that characterizes the entire communication system, avoiding the complexity of bidirectional network protocols while providing users with the feedback they need to understand system operation.

4.5 PyREVIT PLUGIN ARCHITECTURE

The PyRevit plugin architecture represents the execution engine of the parametric generation system, transforming structured commands from the external UI into BIM models within Autodesk Revit. PyRevit, an open-source rapid application development framework for Revit, was chosen as the implementation platform [25]. This plugin serves as the bridge between user intent, expressed through the external UI, and the complex parametric operations required to generate building-code-compliant apartment models.

The plugin architecture leverages PyRevit's IronPython environment, which provides seamless integration with Revit's .NET API while maintaining the development agility of Python. This dual nature allows the system to perform computationally intensive BIM operations while maintaining readable, maintainable code. The plugin implements a modular architecture where each major function—template management, building code compliance, parametric generation, and element numbering—operates as an independent but interconnected module. This modularity ensures that individual components can be updated or enhanced without affecting the entire system, crucial for long-term maintenance and future expansion.

The decision to implement the plugin using PyRevit rather than traditional C# Revit add-ins stems from several strategic advantages. Development iterations are significantly faster with PyRevit's reload-on-save capability, eliminating the compile-build-deploy cycle of traditional add-ins. The Python ecosystem provides access to powerful libraries for JSON processing, mathematical operations, and data manipulation. The scripting nature allows for easier customization by end users who may want to extend functionality. Most importantly, PyRevit's infrastructure handles many low-level Revit API complexities, allowing focus on business logic rather than boilerplate code.

The plugin operates within Revit's transactional framework, ensuring that all model modifications are atomic and reversible. Each operation, from template loading to parameter adjustment to element creation, occurs within managed transactions that can be rolled back if errors occur. This transactional integrity is crucial for maintaining model consistency and preventing corruption from partially completed operations. The plugin also implements comprehensive error handling and logging, providing detailed feedback about operation success or failure through the status update mechanism.

4.5.1 PyREVIT PLUGIN INTERFACE

The PyRevit plugin extends Revit's native interface with a comprehensive set of tools specifically designed for parametric apartment generation. This interface provides users with two parallel workflows: a fully automated path using external UI commands, and a semi-manual path using the ribbon interface directly within Revit. Both approaches leverage the same underlying parametric engine while offering different levels of control and customization.

4.5.1.1 RIBBON INTERFACE DESIGN

The custom ribbon tab, labeled "Parametric BIM Generation," integrates seamlessly with Revit's standard interface, appearing as a dedicated tab in the ribbon bar. The interface design follows Revit's established visual language while organizing tools according to the logical workflow of apartment generation.

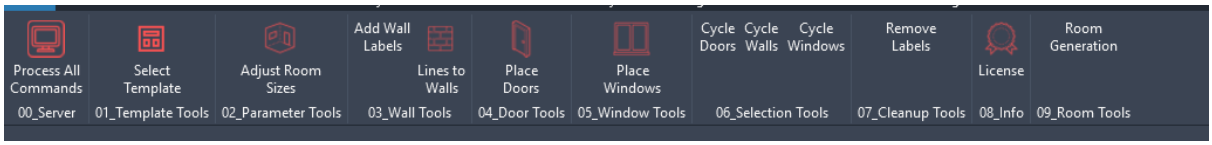

Figure 17: Ribbon Interface of Parametric BIM Generation using PyRevit

The ribbon is organized into seven distinct panels, each representing a stage in the generation process (Figure 17):

- Generation Panel - Primary controls for initiating the generation process
- Parameter Panel - Tools for adjusting room dimensions and spatial relationships
- Wall Generation Panel - Conversion of template lines to 3D walls
- Door and Window Panel - Automated and manual opening placement
- Selection Tools Panel - Element cycling and selection utilities
- Room Tools Panel - Room creation and data management
- Additional Tools Panel - Utility functions and cleanup operations

Each panel uses visual hierarchy to guide users through the workflow. Primary actions feature large split buttons, while secondary functions use standard push buttons. Icons are selected to be immediately

recognizable, using familiar architectural symbols where possible. Tooltips provide detailed explanations of each function, including expected inputs and outputs.

4.5.1.2 BUTTON FUNCTIONALITY OVERVIEW

Each button in the ribbon interface triggers specific parametric operations while maintaining the integrity of the overall model. The functionality is designed to be both powerful and forgiving, with operations that can be undone or modified as needed.

i. **Process All Commands Button:** This primary execution button serves as the main bridge between the external UI and Revit. This is the Master Button to carry out all the modelling commands at once. When activated, it reads the JSON command queue file created by the external UI, interprets the commands, and executes them sequentially.

ii. **Select Template Button:** Opens a visual template browser displaying available apartment configurations. Each template shows its configuration code (1BR_1BA, 2BR_2BA, etc.).

iii. **Adjust Room Sizes Button:** Using this button, users will have the option to manually select the total area of the Apartment. First, it will ask which building code the user would like to follow. A total of 10 countries are included for the users to select from. After that, the user can select the total Area. Depending on the selection of the total area, the Global parameters of the Revit file will be adjusted automatically to comply with the building code and the total area.

iv. **Lines to Wall Button:** Converts the template's single-line representations into full 3D walls. The button presents a dialog for wall type selection, showing Revit's loaded wall families with their thicknesses and materials. Users can also set the wall height after selecting the Revit family.

v. **Place Doors/Windows Buttons:** Clicking these buttons, the user can automatically place all the doors and the windows inside the template. It allows users to select pre-loaded Revit families for doors and windows and place them. The door’s positions are predefined as circles, and the window’s positions are predefined as ellipses in the templates. The button replaces all the circles with Revit doors and all the ellipses with Revit windows. This button also places numbers on each door and window for selection with the external UI.

vi. **Room Generation Button:** Creates Revit room elements, automatically detecting boundaries and calculating areas. The tool names rooms based on their template designation and actual area, creating a standardized naming convention that facilitates scheduling and documentation.

vii. **Cycle Doors, Cycle Walls & Cycle Windows Button:** These three buttons’ functionalities are to select the elements of the Revit file. These functionalities focus more on the external UI rather than on using Revit. A user who is not familiar with Revit modelling can use these three buttons to select the elements in Revit and can modify them.

viii. **Remove Label Button:** With this button, a user can instantly delete all the labels marked on each element (Walls, Doors & Windows).

ix. **License Button:** This button shows basic information about the plugin and the Author.

4.5.2 USER WORKFLOW DEMONSTRATION

The typical workflow for creating a parametric apartment model follows a logical progression from abstract template to detailed BIM model. This section demonstrates the step-by-step process a user would follow when using the buttons inside Revit, without relying on the external UI.

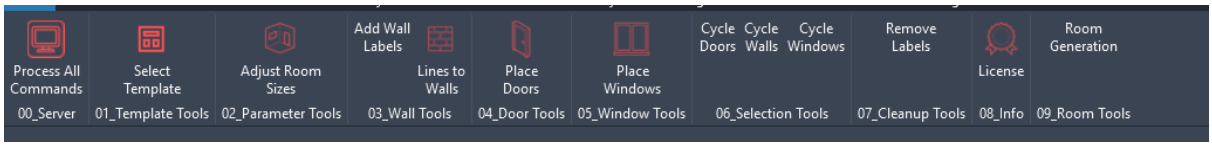

Figure 18: Ribbon Interface of Parametric BIM Generation using PyRevit

4.5.2.1 STEP 1: TEMPLATE SELECTION

The user begins by clicking the 'Select Template' button, which opens the template browser. They can either browse visually through the available options or use filters to narrow the selection. For example, selecting "2 Bedrooms" filters the display to show only 2BR templates. Upon selection, Revit loads the template file, displaying the single-line geometry in the plan view. (Figure 19)

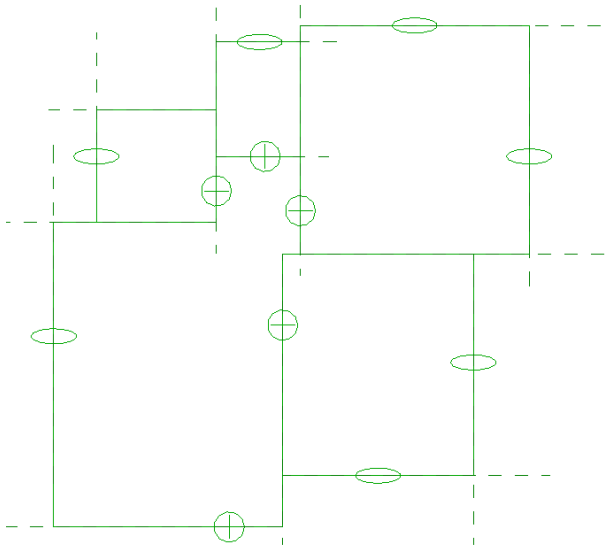

Figure 19: Example Template consisting 2d lines with Global Parameters

The template appears as a simplified line drawing with circles indicating door positions and ellipses showing window locations. Global parameters are loaded simultaneously, appearing in the Properties palette for reference.

4.5.2.2 STEP 2: BUILDING CODE SELECTION

Before adjusting parameters, the user selects the appropriate building code. When the user clicks on the ‘Adjust Room Sizes’ button, a list of building codes appears first to select from (Figure 20). This selection immediately applies the minimum requirements to all relevant parameters. If the current

template violates any requirements, warning symbols appear next to affected parameters, and the system suggests the minimum adjustments needed for compliance.

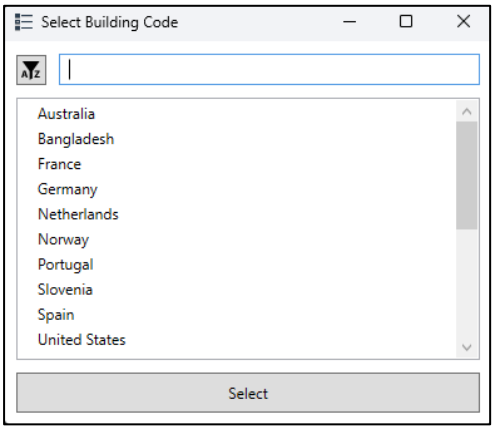

Figure 20: Building Code Options to select

4.5.2.3 STEP 3: PARAMETER ADJUSTMENT

After selecting the building codes, the small UI opens, and asks the user to fill in the total area of the Apartment (Figure 21).

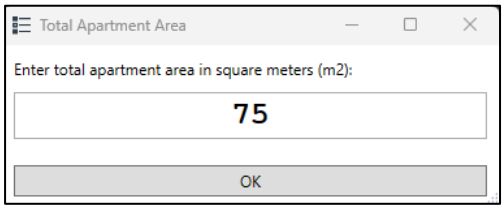

Figure 21: Total Apartment Area Selection

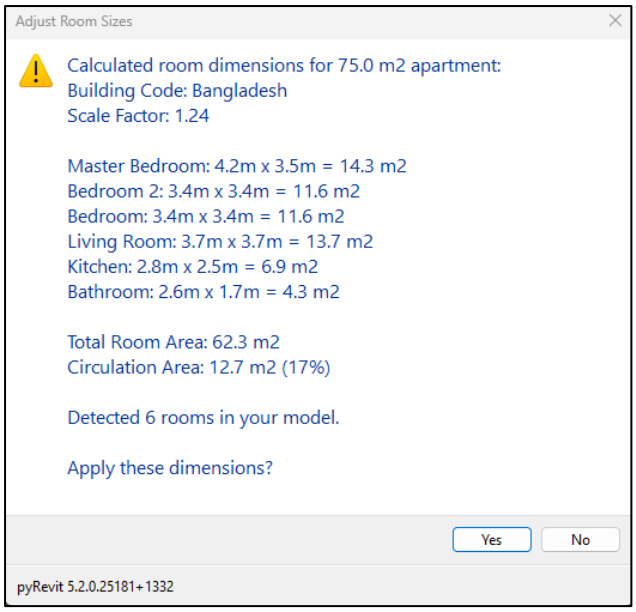

Figure 22: Automatic Adjustment of Room sizes

After that, the interface displays a dialog that includes a "Balance Areas" function that automatically adjusts room sizes proportionally to achieve a target total area while maintaining all building code requirements (Figure 22). This intelligent distribution considers room importance hierarchies, ensuring living spaces and bedrooms receive priority over utility spaces.

4.5.2.4 STEP 4: WALL GENERATION

With parameters finalized, the user proceeds to wall creation by clicking the ‘Lines to Wall’ button. The resulting dialog shows available wall families from the current project (Figure 23). After selecting appropriate types for the walls and setting the wall height (Figure 24), the user clicks "Generate Walls."

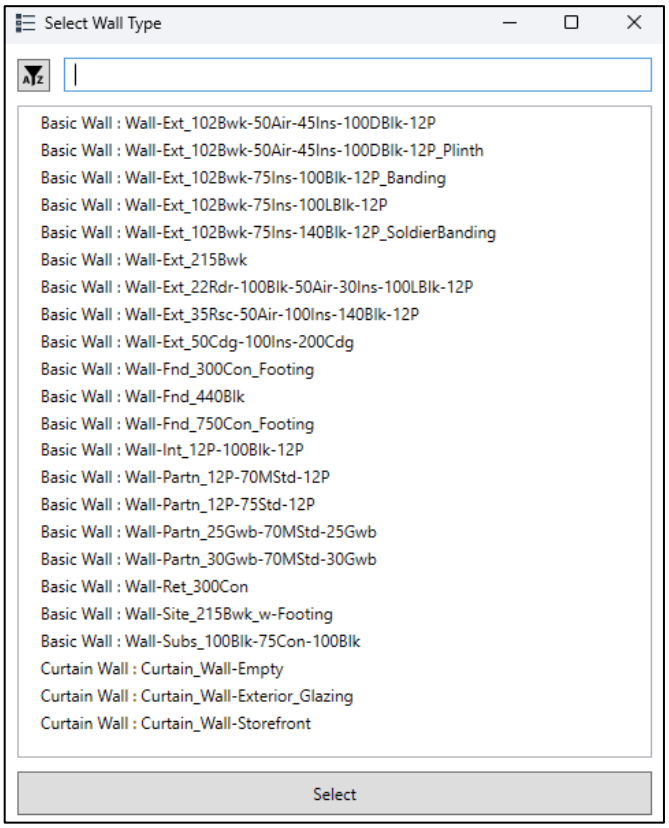

Figure 23: Selection of Wall Family

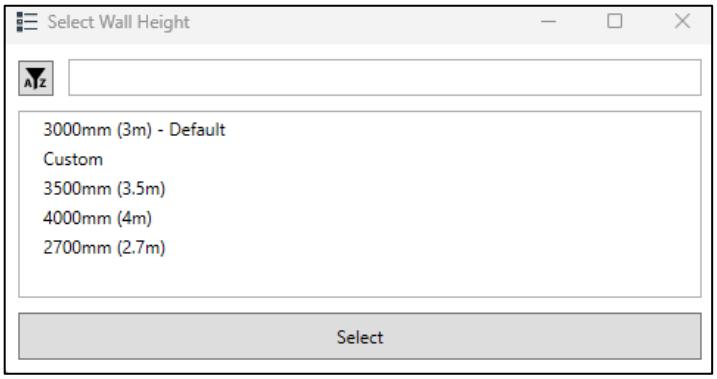

Figure 24: Selection of Wall Height

The system then converts each line in the template to a 3D wall, automatically handling intersections and joins. The transformation is immediate and visual, with the 3D view updating to show the full wall geometry (Figure 26).

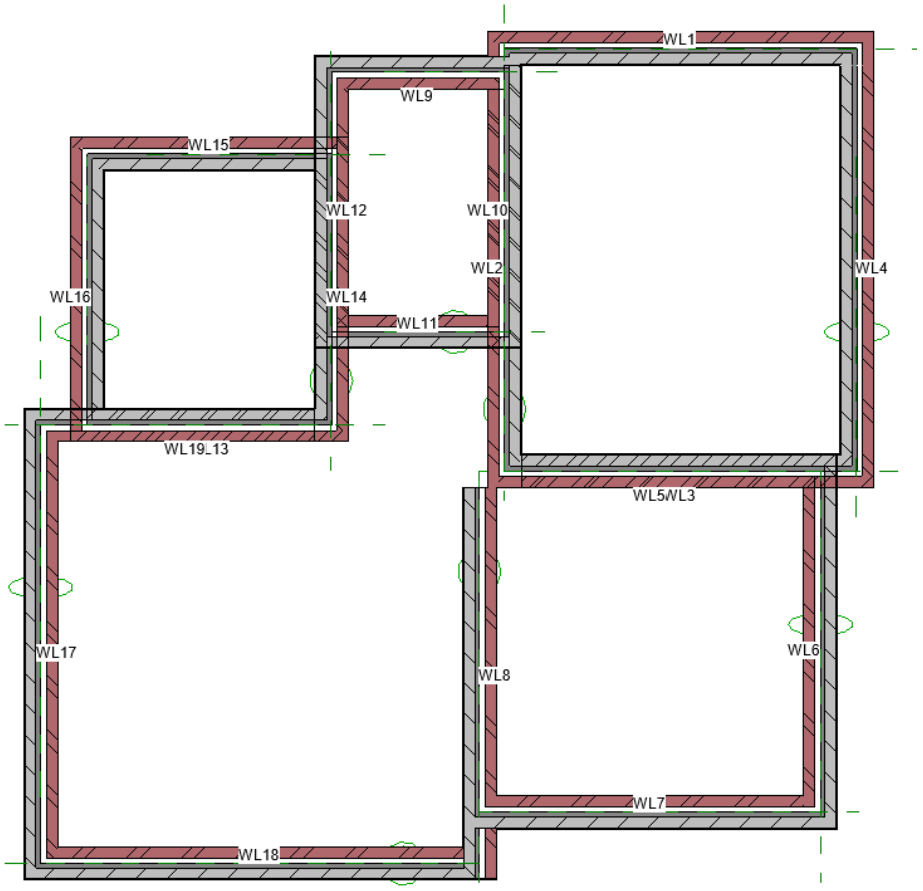

Figure 25: Generated BIM Model- Plan View

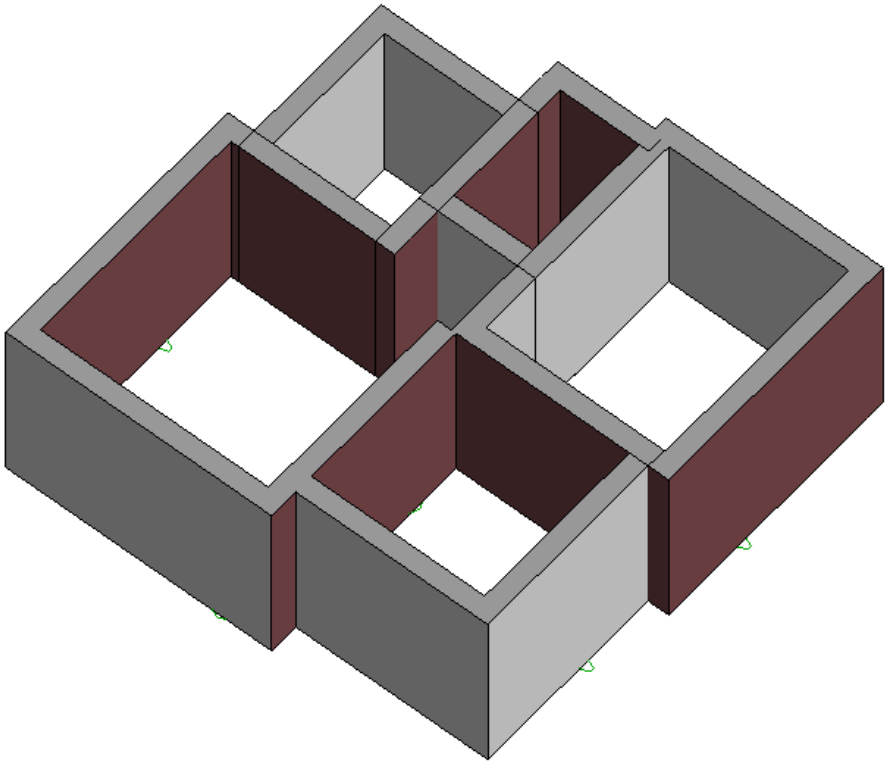

Figure 26: Generated BIM Model- Perspective view

4.5.2.5 STEP 5: OPENING PLACEMENT

The user clicks Place Doors to initiate automatic door placement. First, a list of available Revit door families is visible for the client to select from (Figure 27).

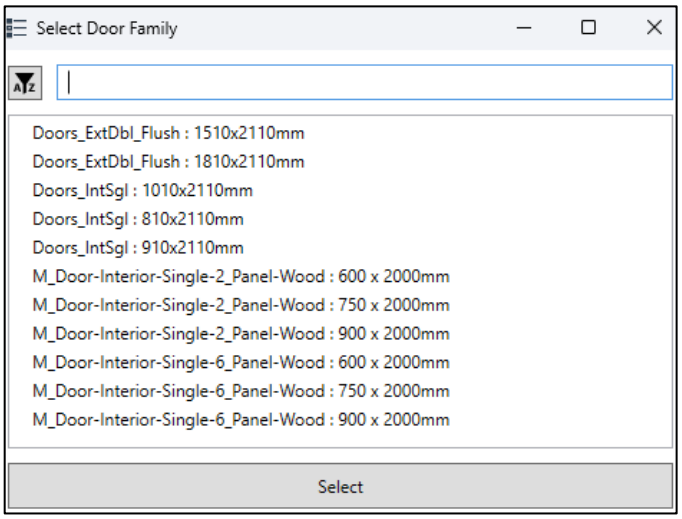

Figure 27: Selection of Door Family

The system analyzes the wall configuration and places doors at the circular markers in the template (Figure 28).

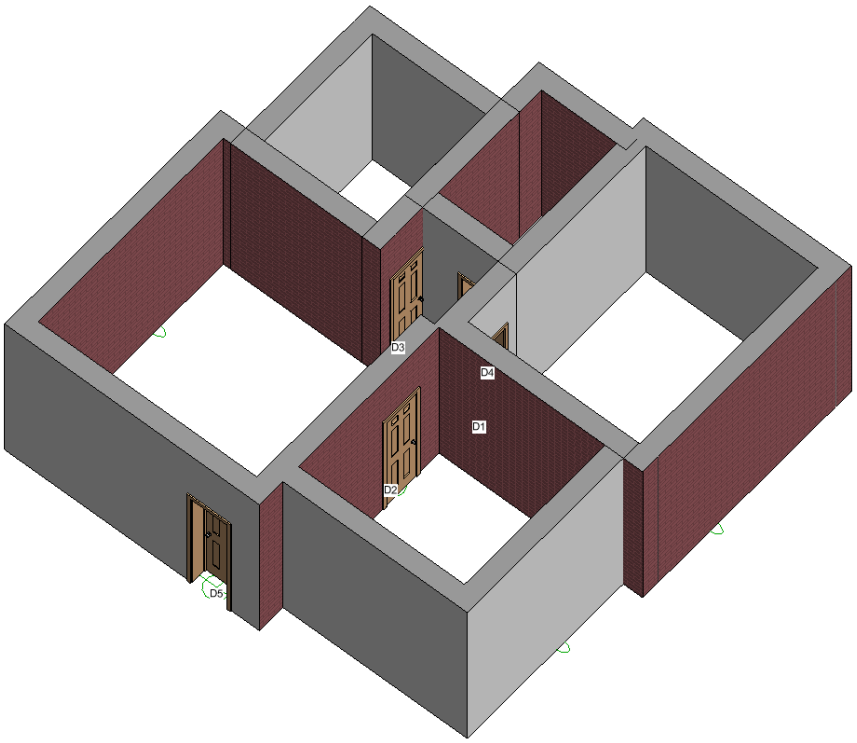

Figure 28: Door placement in the BIM model

Similarly, Place Windows analyzes exterior walls and elliptical markers to position windows. Similar to the door, users can select from the list of loaded window families and can place all the windows automatically (Figure 29).

Figure 29: Window Placement in the BIM Model

4.5.2.6 STEP 6: ROOM CREATION

The final step involves clicking Room Generation to create room elements. The system automatically detects room boundaries formed by the walls and places room elements at the center of each space (Figure 30). Rooms are named according to their function and actual area (e.g., "Master Bedroom - 12.5 m²"), providing immediate verification that spatial requirements have been met.

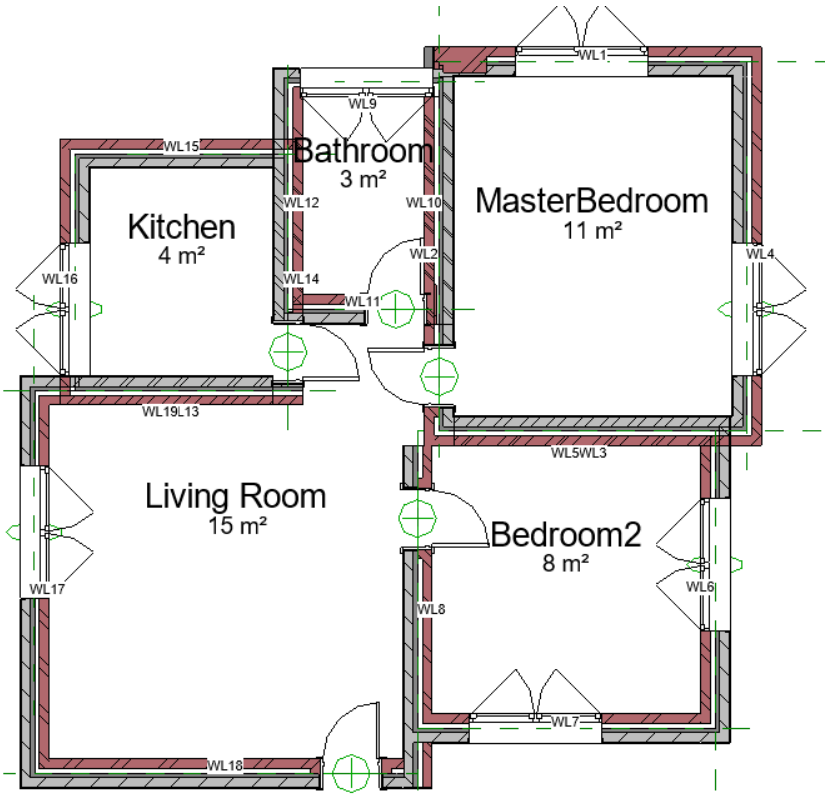

Figure 30: Room placement

4.5.2.7 STEP 7: SELECTION CYCLE BUTTONS

From all the created Walls, Doors, and Windows, some might have some orientation problems. Like some windows might need to swing open outside, not inside. Some doors needed to be flipped. Some walls needed to be flipped, or the family type needed to be changed. Though these can easily be done in Revit without the help of this PyRevit plugin, as these are all Revit families, a user who has no experience in Revit can use the Selection Cycle buttons to select, flip, or change the family types.

The ‘Cycle Doors’, ‘Cycle Walls’ & ‘Cycle Windows’ buttons have the same functionality. Clicking on the ‘Cycle Doors’ button, the user will see a small UI (Figure 31) to select doors by clicking the ‘Next’ button, can flip the door front-back & left-right, and can change the family type of the selected door.

The functionality of this button grows exponentially when the external UI gives any commands to make any modifications.

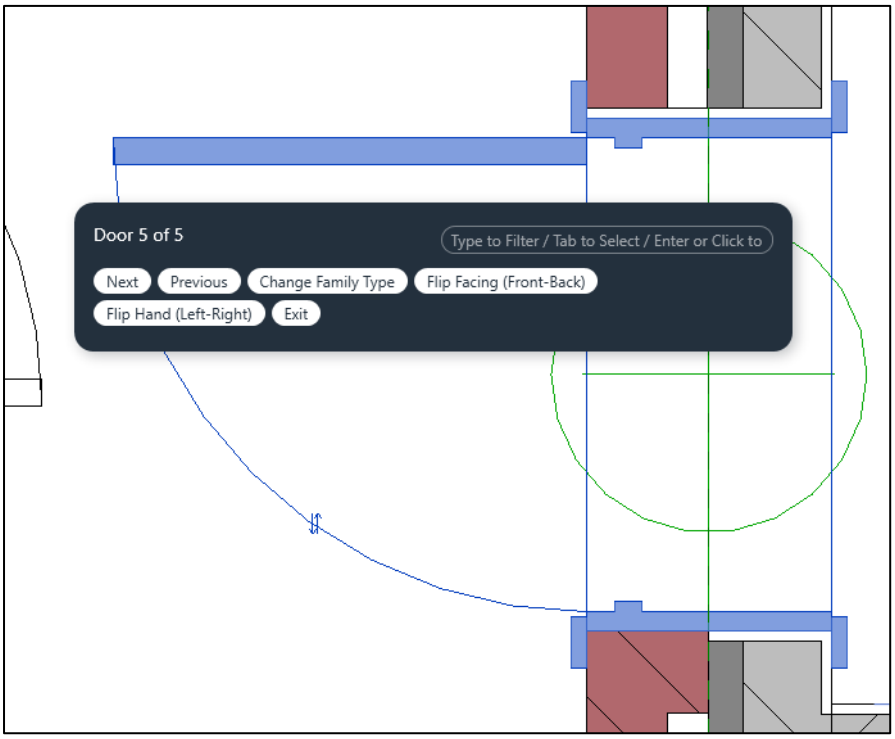

Figure 31: Element selection cycle for modification purposes

4.5.3 TEMPLATE-BASED DESIGN SYSTEM

Templates form the foundational framework for parametric apartment generation, providing pre-configured spatial layouts that can be dynamically adjusted to meet specific requirements. The template system represents a crucial architectural decision that balances flexibility with standardization, enabling rapid generation while maintaining design quality and code compliance. Each template encapsulates not just geometry but also the relationships, constraints, and parameters that define a functional apartment layout.

The template concept emerged from the recognition that most apartment designs follow established patterns based on bedroom count, bathroom configuration, and local architectural conventions. Rather than generating layouts from scratch, a computationally complex and error-prone process, the system leverages these patterns as starting points, adjusting them parametrically using the Global Parameter system of Revit to meet specific requirements.

Templates are intrinsically linked to the **Process Commands Button** in the PyRevit interface. When this button is activated, the system reads the command queue JSON file and extracts the apartment configuration (e.g., 2BR_2BA) specified by the user. This configuration key triggers the template selection mechanism, which loads the corresponding template file as the foundation for generation. The template provides the spatial framework, while the commands provide the specific dimensional and compliance requirements that drive parametric adjustments.

The **Select Template Button** provides direct access to the template library, enabling manual template selection independent of command processing. This functionality proves valuable for exploring different layout options or when users want to override automatic template selection. The visual browser activated by this button displays template previews, making it easy to understand the spatial organization of each option before selection.

4.5.3.1 TEMPLATE ORGANIZATION & NAMING CONVENTION

The template library follows a systematic organization that facilitates both automated selection and manual browsing. Templates are stored as Revit files (.rvt) in a hierarchical directory structure that mirrors their classification:

```
Templates/
├── 1BR_1BA/
│   ├── 1BR_1BA_Standard.rvt (45-55 m²)
│   ├── 1BR_1BA_Compact.rvt (35-45 m²)
│   ├── 1BR_1BA_Luxury.rvt (55-70 m²)
│   └── 1BR_1BA_Corner.rvt (50-60 m²)
├── 2BR_1BA/
│   ├── 2BR_1BA_Standard.rvt (60-70 m²)
│   ├── 2BR_1BA_Compact.rvt (55-65 m²)
│   └── 2BR_1BA_Linear.rvt (65-75 m²)
├── 2BR_2BA/
│   ├── 2BR_2BA_Standard.rvt (70-85 m²)
│   ├── 2BR_2BA_Master.rvt (75-90 m²)
│   ├── 2BR_2BA_Corner.rvt (75-85 m²)
│   └── 2BR_2BA_Penthouse.rvt (85-100 m²)
└── 3BR_2BA/
    ├── 3BR_2BA_Standard.rvt (85-100 m²)
    ├── 3BR_2BA_Family.rvt (90-110 m²)
    └── 3BR_2BA_Luxury.rvt (100-120 m²)
```

Figure 32: Template naming convention

The naming convention follows a strict pattern: {#BR}_{#BA}_[variant].rvt, where:

- {#BR} indicates the number of bedrooms (1BR, 2BR, 3BR)
- {#BA} indicates the number of bathrooms (1BA, 2BA, 3BA)
- [variant] optionally specifies special configurations

This systematic naming enables automatic template selection based on user requirements while providing flexibility for specialized variants. The base templates (without variants) serve as defaults, while variants offer optimized solutions for specific contexts, such as the area of the apartment.

Each template contains carefully structured elements that enable parametric control:

i. **Single-Line Wall Representations:** Walls are initially represented as single lines in plan view, which significantly simplifies parametric relationships and computational overhead. These lines serve as centerlines for 3D wall generation, with wall thickness applied symmetrically during the conversion process.

ii. Symbolic Markers for Openings:

- Circles represent door locations, with the circle diameter correlating to the rough opening width.
- Ellipses indicate window positions. The hexagonal shape distinguishes windows from doors even in complex layouts.
- Rectangles define room boundaries and functional zones, providing the framework for space allocation and area calculations.

4.5.3.2 GLOBAL PARAMETERS & THEIR ARCHITECTURAL SIGNIFICANCE

Global Parameters in Revit serve as the control mechanism for template flexibility, allowing dynamic adjustment of spatial dimensions while maintaining geometric relationships. Each template includes a comprehensive set of parameters that control every aspect of the layout's geometry.

Each template includes a comprehensive parameter set controlling all dimensional aspects.

i. Room Dimension Parameters:

- RoomWidth_MasterBedroom (default: 3.5m, min: 3.0m)
- RoomLength_MasterBedroom (default: 4.0m, min: 3.0m)
- RoomWidth_Bedroom2 (default: 3.0m, min: 2.7m)
- RoomLength_Bedroom2 (default: 3.5m, min: 2.7m)
- RoomWidth_LivingRoom (default: 4.5m, min: 3.5m)
- RoomLength_LivingRoom (default: 5.0m, min: 3.5m)
- RoomWidth_Kitchen (default: 3.0m, min: 2.0m)
- RoomLength_Kitchen (default: 3.5m, min: 2.0m)
- RoomWidth_Bathroom (default: 2.0m, min: 1.5m)
- RoomLength_Bathroom (default: 2.5m, min: 1.5m)

ii. Vertical Parameters:

- WallHeight (default: 3.0m, range: 2.5m - 4.0m)
- FloorThickness (default: 0.3m)
- CeilingOffset (default: 0.0m)

iii. Opening Parameters:

- DoorWidth_Main (default: 0.9m, min: 0.8m)
- DoorHeight_Main (default: 2.1m, min: 2.0m)
- DoorWidth_Interior (default: 0.8m, min: 0.7m)
- DoorHeight_Interior (default: 2.1m, min: 2.0m)
- WindowWidth_Standard (default: 1.2m, min: 0.6m)
- WindowHeight_Standard (default: 1.5m, min: 1.0m)
- WindowSillHeight_Standard (default: 0.9m, min: 0.8m)

These Global parameters are significant for this entire process, as these are adjusted following the user's requirements, regardless of using external UI or Revit's plugin internally. These parameters are assigned to the dimensions of each of the rooms. So, automatically adjusting the Global parameter, users can adjust the room sizes as well. And the Global parameters can be adjusted automatically using the External Ui and the PyRevit Plugin's buttons.

4.5.3.3 PARAMETRIC RELATIONSHIPS & CONSTRAINTS

Templates implement sophisticated parametric relationships that maintain design integrity during dimensional adjustments. These relationships ensure that as individual parameters change, the overall design remains functionally and aesthetically coherent. The constraint system operates at multiple levels: geometric, functional, and aesthetic.

Proportional Relationships: Aesthetic proportions are preserved through parametric formulas:

IF (RoomWidth_MasterBedroom < 3.0) THEN

RoomLength_MasterBedroom = RoomWidth_MasterBedroom * 1.3

ELSE

RoomLength_MasterBedroom = RoomWidth_MasterBedroom * 1.15

These relationships maintain pleasing room proportions even as dimensions change. Rooms that become too narrow automatically become longer to maintain usable area, while rooms that are wide enough maintain more balanced proportions.

4.5.3.4 TEMPLATE SELECTION PROCESS (MANUEL VS AUTOMATIC)

The system supports two distinct template selection methods, each optimized for different use cases and user preferences.

i. **Automated Selection via External UI:** When processing commands from the external UI, the system employs a sophisticated scoring algorithm to select the optimal template:

```python
def calculate_template_score(template, requirements):
    score = 0

    # Exact configuration match (highest priority)
    if template.bedrooms == requirements.bedrooms:
        score += 100
    if template.bathrooms == requirements.bathrooms:
        score += 50

    # Area proximity (inverse of difference)
    area_difference = abs(template.default_area - requirements.target_area)
    area_score = max(0, 100 - area_difference)
    score += area_score * 0.5

    # Building code pre-validation bonus
    if requirements.building_code in template.validated_codes:
        score += 30

    # Special features alignment
    if requirements.corner_unit and template.is_corner:
        score += 25
    if requirements.accessibility and template.has_accessible_features:
        score += 40

    return score
```

Figure 33: Template Selection Algorithm

This scoring system ensures that the selected template requires minimal adjustment to meet requirements, reducing processing time and potential parameter conflicts.

ii. **Manual Selection via Revit Interface:** The visual template browser, activated by the Select Template button, presents template names as a list. Configuration code (e.g., "2BR_2BA_Standard")

4.5.3.5 TEMPLATE VALIDATION & QUALITY ASSURANCE

Before inclusion in the system library, each template undergoes rigorous validation to ensure it meets quality standards and will function correctly within the parametric generation system. This validation process combines testing with manual review, verifying both technical accuracy and design quality.

i. **Manual Geometric Validation:** Geometric checks verify the physical integrity of the template:
   - Enclosure Completeness: All spaces are fully bounded by walls with no gaps
   - Overlap Detection: No room boundaries overlap or intersect inappropriately
   - Circulation Verification: All rooms are accessible through proper circulation paths
   - Minimum Clearances: Required clearances for doors, windows, and circulation are maintained

ii. **Automatic Geometric Validator:** This computational geometry algorithm is used to detect issues that might not be visually apparent but could cause generation failures. For instance, it identifies walls that are close but not quite touching, which could create gaps in the 3D model.

iii. **Parametric Testing:** The parametric validation system tests templates across their full range of adjustment:
   - Global Parameters are systematically varied from minimum to maximum values

- Each configuration is checked for geometric validity and constraint satisfaction
- Extreme combinations are tested to identify edge cases
- Performance is measured to ensure generation completes within acceptable timeframes

This exhaustive testing ensures templates remain valid across all possible parameter combinations users might specify. The system generates hundreds of variations from each template, validating that all produce viable apartment layouts.

iv. **Building Code Pre-Compliance:** Templates are designed to meet the most restrictive requirements across all supported building codes:
   - Room dimensions exceed the maximum minimums across all codes
   - Circulation widths accommodate the strictest accessibility requirements
   - Ceiling heights meet the tallest requirements

This conservative approach ensures that code-specific adjustments typically involve increasing dimensions rather than fundamental layout changes. Templates serve as compliant starting points that can be refined for specific jurisdictions rather than requiring major restructuring.

v. **Quality Assurance Review:** Manual review by architectural professionals ensures design quality:
   - Layouts are evaluated for livability and functionality
   - Furniture placement zones are verified for standard furniture sizes
   - Natural light distribution is assessed for all habitable rooms
   - Privacy considerations are checked for bedroom and bathroom placement
   - Overall aesthetic quality is evaluated

This human review complements automated validation, ensuring templates not only meet technical requirements but also represent good architectural design. Templates that pass all validation stages are tagged with metadata describing their characteristics, optimal use cases, and any special considerations for their application.

4.5.4 BUILDING CODE COMPLIANCE SYSTEM

The building code compliance system represents one of the most innovative aspects of this PyRevit plugin, transforming traditionally manual compliance checking into an automated, intelligent process. This system ensures that every generated apartment model meets the specific regulatory requirements of the selected jurisdiction, addressing a critical gap in existing parametric design tools. The compliance system operates throughout the generation process, from initial template selection through final validation, continuously ensuring that all design decisions respect applicable regulations.

The integration of building code compliance directly into the generation process, rather than as a post-generation check, fundamentally changes how compliant designs are created. Traditional workflows require architects to manually verify compliance after design completion, often necessitating significant revisions when violations are discovered. This system inverts that process, using compliance requirements as generative constraints that guide the design from the outset.

4.5.4.1 BUILDING CODE DATA COLLECTION & PROCESSING

The foundation of the compliance system rests on a comprehensive database of building code requirements extracted from official regulatory documents. This database represents careful research, involving the systematic analysis of building codes from ten different countries and their transformation into computationally processable formats.

i. Data Collection Methodology:

The data collection process began with identifying authoritative sources for each country's building codes:

- **Slovenia:** Official Gazette of the Republic of Slovenia (Uradni list RS), specifically the Rules on minimum technical requirements for the construction of residential buildings and dwellings
- **USA:** International Building Code (IBC) 2021 and International Residential Code (IRC) 2021 [32] from the International Code Council
- **Germany:** DIN 18011 (Surface areas and volumes of buildings) and DIN 18040 (Construction of accessible buildings)
- **France:** Code de la Construction et de l'Habitation, particularly Articles R111-1 through R111-17
- **Spain:** Código Técnico de la Edificación (CTE), Document DB-SUA (Safety of Use and Accessibility)
- **Australia:** National Construction Code (NCC) Volume Two, Class 1 and Class 2 buildings
- **Netherlands:** Bouwbesluit 2012, Chapter 4 (Technical building regulations for usability)
- **Norway:** Direktoratet for byggkvalitet (DiBK) Building Technical Regulations (TEK17)
- **Portugal:** Regulamento Geral das Edificações Urbanas (RGEU), Decree-Law n.º 38382
- **Bangladesh:** Bangladesh National Building Code (BNBC) 2020, Part 3, Chapter 1 (Building Planning)

From these comprehensive documents, typically hundreds of pages each, residential space requirements were extracted and categorized. This extraction focused on quantifiable requirements that could be automatically verified:

- Minimum room dimensions (width, length, area)

- Ceiling height requirements (general and room-specific)
- Natural lighting and ventilation standards (window-to-floor ratios)
- Accessibility requirements (door widths, turning circles, approach spaces)
- Circulation and egress specifications (corridor widths, stair dimensions)
- Special provisions for different room types (kitchens, bathrooms, bedrooms)

ii. Data Structuring and Standardization:

The extracted requirements were transformed into a hierarchical JSON structure (Figure 34) that preserves the semantic meaning while enabling efficient computational processing:


```json
{
  "country": "Bangladesh",
  "code_name": "Bangladesh National Building Code (BNBC)",
  "version": "2020",
  "last_updated": "2020-02-28",
  "source": "Housing and Building Research Institute",
  "residential_requirements": {
    "general": {
      "min_ceiling_height": 2.75,
      "min_ceiling_height_kitchen": 2.4,
      "min_ceiling_height_bathroom": 2.4,
      "min_door_width": 0.75,
      "min_door_height": 2.0,
      "min_corridor_width": 0.9
    },
    "rooms": {
      "master_bedroom": {
        "min_area": 9.5,
        "min_width": 2.75,
        "min_length": 3.0,
        "window_area_ratio": 0.1,
        "notes": "At least one room should be minimum 9.5 sqm"
      },
      "bedroom": {
        "min_area": 7.5,
        "min_width": 2.4,
        "min_length": 2.75,
        "window_area_ratio": 0.1
      },
      "living_room": {
        "min_area": 12.0,
        "min_width": 3.0,
        "min_length": 3.5,
        "window_area_ratio": 0.15,
        "can_combine_with_dining": true
      }
    },
    "accessibility": {
      "required_for_public_housing": true,
      "wheelchair_accessible_unit_ratio": 0.05,
      "accessible_bathroom_min_area": 3.7,
      "accessible_door_min_width": 0.9,
      "turning_circle_diameter": 1.5
    }
  }
}
```


Figure 34: Building Code JSON Structure

This standardized structure enables consistent processing across different building codes despite their varying formats and requirements. Each code file follows the same schema, allowing the compliance engine to process any jurisdiction's requirements using the same algorithms.

4.5.4.2 BUILDING CODE VARIATIONS & CHALLENGES

The standardization process revealed significant variations between building codes, presenting numerous challenges that the system had to address:

i. **Measurement System Variations:** While most countries use metric measurements, the USA's continued use of imperial units required careful handling:
   - The system internally uses metric units for all calculations
   - Imperial measurements in US codes are converted during data import
   - User interfaces display units according to the selected building code
   - Conversion precision is maintained to avoid rounding errors affecting compliance

ii. **Regulatory Philosophy Differences:** Building codes embody different regulatory philosophies that affect how requirements are expressed:
   - **Prescriptive Codes** (Germany, France): Specify exact dimensional requirements
   - **Performance-Based Codes** (Australia, Netherlands): Define outcomes rather than methods
   - **Hybrid Approaches** (USA, Slovenia): Combine prescriptive and performance elements

The system handles these differences by extracting the most specific requirements available and inferring specific dimensions from performance criteria where necessary.

iii. **Cultural and Climatic Adaptations:** Regional variations reflect cultural preferences and climatic necessities:

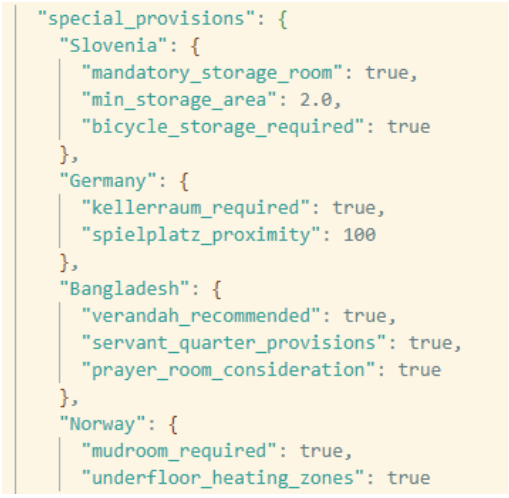

Figure 35: Building Code Cultural Requirements

These cultural requirements are incorporated as additional generation constraints when the corresponding building code is selected.

4.5.4.3 COMPLIANCE CHECKING ALGORITHM

The compliance checking algorithm operates as a multi-stage process that validates and adjusts design parameters throughout generation. This algorithm represents the technical core of the compliance system, implementing sophisticated logic to ensure regulatory adherence while maintaining design quality.

i. Stage 1: Requirement Loading and Parsing

When a building code is selected, the compliance engine loads the corresponding JSON file of the building code and parses it into an internal representation optimized for rapid querying. The parsing process creates multiple data structures:

- A flat dictionary of minimum requirements for lookup performance
- A hierarchical tree structure preserving requirement relationships
- An index of requirements by room types for targeted validation
- A priority queue of requirements ordered by restrictiveness

This multi-structure approach enables different validation strategies depending on the operation being performed, optimizing both performance and accuracy.

ii. Stage 2: Parameter Validation and Adjustment

The parameter validation stage systematically compares template parameters against building code requirements, identifying violations and calculating necessary adjustments:

The validation algorithm implements a sophisticated adjustment strategy that considers the cascading effects of parameter changes. When a room dimension falls below the minimum requirement, the system must decide how to adjust it while maintaining overall design coherence. The algorithm considers multiple factors:

- The magnitude of the violation (how far below the minimum)
- Available space in adjacent areas
- Impact on total area constraints
- Maintaining proportional relationships
- Preserving circulation paths

For example, if a bedroom width of 2.5m violates a 2.7m minimum, the algorithm might:

- Increase the bedroom width to 2.7m
- Reduce an adjacent room to compensate
- Adjust the overall building footprint if necessary

- Recalculate all dependent parameters

4.5.4.4 COMPLIANCE REPORTING & DOCUMENTATION

The compliance system generates comprehensive reports (Figure 36) documenting all validation checks and adjustments, providing transparency and accountability in the compliance process:


```
{
  "compliance_report": {
    "project_id": "APT_2025_001",
    "timestamp": "2025-08-06T15:45:00",
    "building_code": "Bangladesh",
    "overall_status": "COMPLIANT",
    "summary": {
      "total_checks": 47,
      "passed": 43,
      "adjusted": 4,
      "failed": 0
    },
    "checks_performed": [
      {
        "category": "Room Dimensions",
        "status": "ADJUSTED",
        "details": [
          {
            "room": "MasterBedroom",
            "requirement": "min_area >= 9.5
            "original": 9.0,
            "adjusted": 9.5,
            "status": "ADJUSTED"
          }
        ]
      }
    ],
    "certification": {
      "compliant": true,
      "signature": "System Generated",
      "date": "2025-08-06"
    }
  }
}
```


Figure 36: Compliance Report Generation

These reports serve multiple purposes:

- Documentation for regulatory approval
- Record of design decisions for project files
- Debugging information for system operators
- Learning data for system improvement

4.5.5 PARAMETRIC ROOM GENERATION

The Parametric Room Generation Workflow represents the core intelligence of the system's space allocation mechanism, dynamically adjusting room dimensions to satisfy both user-specified total area requirements and mandatory building code constraints. This sophisticated workflow transforms abstract area specifications into precisely dimensioned spatial layouts through an optimization algorithm that balances multiple competing objectives: achieving the target total area, maintaining code compliance, preserving architectural proportions, and ensuring functional room relationships. The system implements this complex negotiation through the Adjust Room Sizes button, which serves as the parametric engine driving template transformation.

4.5.5.1 BUILDING CODE INTEGRATION & MINIMUM AREA EXTRACTION

The workflow begins by loading comprehensive building code data from JSON files stored in the system's data directory. Each building code file contains structured requirements for ten supported countries, encoding minimum room areas, corridor widths, ceiling heights, and other spatial constraints extracted from official regulatory documents. The system dynamically discovers available building codes at runtime, ensuring that new codes can be added without modifying the core implementation.

The code parser extracts room-specific minimum areas through a sophisticated mapping mechanism that reconciles differences in regulatory terminology. Building codes use varied nomenclature—"master bedroom," "primary bedroom," or "hauptschlafzimmer" (German for Master Bedroom), which the system maps to standardized internal room types. This semantic translation ensures consistent application of requirements regardless of regulatory language or structure:

```
def get_room_minimum_areas(building_code, room_types_in_model):
    """Extract minimum room areas from building code"""
    room_mapping = {
        'master_bedroom': ['master_bedroom'],
        'bedroom': ['bedroom', 'bedroom2'],
        'secondary_bedroom': ['bedroom', 'bedroom2'],
        'living_room': ['living_room'],
        'kitchen': ['kitchen'],
        'bathroom': ['bathroom']
    }
```

Figure 37: Room Type Mapping for Building Codes

The extraction process prioritizes code-specified minimums but provides intelligent defaults for rooms not explicitly regulated. This fallback mechanism ensures that the system can generate complete layouts even when building codes provide incomplete specifications, maintaining functional minimums based on architectural best practices.

4.5.5.2 AREA DISTRIBUTION & SCALING ALGORITHM

The area distribution algorithm represents the mathematical core of the parametric workflow, calculating optimal room dimensions that satisfy both the total area constraint and individual room minimums. The system first determines whether the requested total area can accommodate all minimum requirements plus necessary circulation space, typically allocated as 20% of the total area.

When the requested area exceeds minimum requirements, the algorithm calculates a scale factor that proportionally enlarges all rooms while maintaining their relative size relationships:

```
def calculate_room_dimensions(total_area, room_minimums, circulation_factor=0.20):
    """Calculate room dimensions based on total area and minimums"""
    min_room_area = sum(room_minimums.values())
    min_total_area = min_room_area * (1 + circulation_factor)

    if total_area >= min_total_area:
        available_room_area = total_area / (1 + circulation_factor)
        scale_factor = math.sqrt(available_room_area / min_room_area)
```

Figure 38: Area Distribution Algorithm

The square root scaling ensures that linear dimensions increase proportionally, maintaining room aspect ratios while achieving the target areas. This approach prevents rooms from becoming excessively elongated or compressed as they scale, preserving architectural quality across different apartment sizes.

Room-specific aspect ratios fine-tune the dimensional calculation. Master bedrooms receive a 1.2:1 length-to-width ratio for slight rectangularity, bathrooms use 1.5:1 to accommodate fixtures linearly, while living rooms remain nearly square for furniture arrangement flexibility. These ratios, derived from architectural standards, ensure that generated rooms remain functionally appropriate regardless of size.

4.5.5.3 GLOBAL PARAMETER SYNCHRONIZATION

The final stage synchronizes calculated dimensions with Revit's Global Parameter system, propagating room sizes throughout the parametric template. The system identifies existing global parameters through FilteredElementCollector queries, matching parameter names to room types through the predefined mapping structure.

Parameter updates execute within a single transaction, ensuring atomic modification of all related dimensions. The system converts calculated millimetre values to Revit's internal units, maintaining precision while conforming to the API requirements. Each parameter update includes validation to confirm successful modification, with detailed reporting of any parameters that fail to update.

The workflow supports both interactive and external UI modes, adapting its behaviour based on the presence of command files. In external mode, the system bypasses confirmation dialogs and provides

structured output suitable for programmatic parsing, enabling seamless integration with the natural language interface while maintaining full functionality for direct Revit users.

4.5.6 WALL GENERATION FROM LINES

The wall generation system transforms the template's single-line representations into fully realized 3D BIM walls, bridging the gap between schematic layout and constructible model. This transformation represents a critical stage where abstract geometric representations become tangible building elements with material properties, structural characteristics, and dimensional accuracy. The system implements an intelligent line-to-wall conversion algorithm that maintains spatial relationships while introducing the physical thickness and properties required for a complete BIM model.

4.5.6.1 LINE COLLECTION & VALIDATION

The conversion process begins with a comprehensive line discovery mechanism that ensures all valid geometric elements are identified for transformation. The system employs a dual-strategy collection approach to capture lines regardless of their creation method or category classification. This redundancy ensures robustness across different template configurations and user workflows.

The primary collection phase utilizes Revit's FilteredElementCollector API [33] with category-specific filters. The system first queries for detail lines using the OST_Lines built-in category, then supplements this collection with model lines through the CurveElement class filter. Each collection operation is scoped to the active view to ensure only visible, relevant geometry is processed. This view-based filtering prevents the inadvertent conversion of hidden or reference geometry that might exist in the model but shouldn't form part of the generated apartment.

```
def find_all_lines_in_view():
    """Find all straight lines in the current view"""
    lines = []
    processed_locations = []  # Track line start/end points to avoid duplicates

    # Method 1: Detail Lines
    detail_lines = DB.FilteredElementCollector(doc, view.Id)\
        .OfCategory(DB.BuiltInCategory.OST_Lines)\
        .WhereElementIsNotElementType()\
        .ToElements()
```

Figure 39: Line Collection for Wall Generation

The validation phase implements multiple checks to ensure line quality and uniqueness. Each discovered curve undergoes geometric validation through the is_straight_line() function, which verifies that the element is a true linear segment rather than an arc, spline, or other curved geometry. This verification is crucial because the Wall.Create API [33] expects linear paths for wall centerlines. Lines shorter than 100 millimeters are automatically filtered out, preventing the creation of invalid micro-walls that could result from drafting artifacts or accidental line segments.

Duplicate detection employs coordinate-based comparison with controlled precision. The system rounds start and end coordinates to six decimal places, creating a unique signature for each line segment. Before accepting a line for processing, this signature is checked against a registry of previously processed locations. This approach elegantly handles the common scenario where both detail lines and model lines exist at the same location, ensuring only one wall is created per unique line segment.

4.5.6.2 WALL CREATION & PROPERTY ASSIGNMENT

The wall creation phase transforms validated line geometry into three-dimensional BIM walls with full material and parametric properties. This process involves wall type selection, dimensional parameter application, and the invocation of Revit's wall creation API with appropriate configuration.

Wall type selection provides crucial flexibility in defining the physical characteristics of generated walls. The system dynamically queries the project database for all available wall types, presenting them in a hierarchical format that combines family and type names. This presentation enables informed selection based on construction requirements, whether for schematic single-layer walls or detailed multi-layer assemblies with specific thermal and structural properties.

The core wall generation employs Revit's 'Wall.Create' method that instantiates wall elements based on the provided parameters:

```
def create_wall_from_line(line, wall_type, level, height):
    """Create a wall from a line"""
    wall = DB.Wall.Create(
        doc,           # Document reference
        line,          # Centerline geometry
        wall_type.Id,  # Wall type identifier
        level.Id,      # Base level reference
        height,        # Wall height in feet
        0,             # Offset from level
        False,         # Flip orientation
        False          # Structural usage
    )
    return wall
```

Figure 40: Wall Creation Implementation

This implementation positions walls with their centerlines aligned to the template lines, maintaining the spatial relationships defined during parametric adjustment. The height parameter, converted from user-specified millimeters to Revit's internal units, ensures walls extend to the correct elevation. The offset parameter remains at zero to maintain level alignment, while the flip and structural parameters receive default values suitable for typical apartment construction.

4.5.6.3 ELEMENT LABELLING & MAPPING STYLE

The labeling system creates a critical bridge between the generated BIM model and the external UI's modification capabilities. Each wall receives a sequential identifier with corresponding visual

annotation, establishing a human-readable reference system that enables subsequent element-specific operations.

Label positioning employs vector mathematics to ensure consistent, readable placement. The algorithm calculates each wall's midpoint, then determines a perpendicular offset vector using the cross-product principle. By swapping and negating the wall direction's X and Y components, the system derives a perpendicular vector that points consistently to one side of the wall. This vector is scaled by 0.5 feet to position labels at a readable distance from the wall surface without interfering with adjacent geometry.

The text note creation utilizes Revit's TextNote API with carefully configured options for optimal visibility. The system attempts to use the smallest available text type, even creating a custom 1/25-inch text type if necessary, ensuring labels remain unobtrusive while maintaining legibility. Text alignment is set to center both horizontally and vertically, providing consistent appearance regardless of wall orientation.

The mapping persistence mechanism saves the relationship between labels and element IDs to a JSON file, creating a persistent bridge between the visual model and the programmatic interface. This mapping includes not only the element identifiers but also metadata such as wall type and height, enabling the external UI to display relevant information about selected elements. The JSON structure facilitates bi-directional lookup, allowing both label-to-element and element-to-label queries during modification operations.

Transaction management wraps all creation and labelling operations within a single Revit transaction, ensuring atomicity and enabling rollback if errors occur. This transactional integrity is crucial for maintaining model consistency, particularly when processing large numbers of walls where partial completion could leave the model in an inconsistent state. The system provides detailed feedback about the operation's success, including counts of created walls, any skipped lines, and the complete element mapping, enabling users to verify successful generation and troubleshoot any issues.

4.5.7 DOOR & WINDOW PLACEMENT SYSTEM

The door and window placement system represents a sophisticated approach to opening generation that leverages geometric markers in the template to guide element insertion. Rather than relying on algorithmic determination of opening positions, which often produces suboptimal results, the system uses architect-defined circular and elliptical markers to precisely control where doors and windows appear in the generated model.

4.5.7.1 GEOMETRIC MARKER DETECTION & COLLECTION

The opening placement system begins with comprehensive detection of geometric markers throughout the model, employing pattern recognition to distinguish door markers (circles) from window markers (ellipses). This geometric differentiation provides an intuitive visual language where circular markers indicate door positions and elliptical markers designate window locations, allowing template designers to clearly communicate opening intentions without additional annotation.

The marker collection algorithm searches the entire model rather than limiting itself to the active view, ensuring that markers on different levels or in auxiliary views are not overlooked. The system employs dual collection strategies through the FilteredElementCollector API, first gathering all CurveElement instances and then supplementing with elements from the OST_Lines category. Each discovered curve undergoes type checking to identify circles and ellipses among the various geometric elements.

Circle validation for door markers requires verification that discovered arcs represent complete circles rather than partial arcs. The algorithm calculates the expected circumference based on the arc's radius and compares it to the actual arc length:

```
def get_all_circles_in_model():
    """Find all circles in the entire model"""
    for elem in curve_collector:
        curve = elem.GeometryCurve
        if curve and isinstance(curve, DB.Arc):
            arc_length = curve.Length
            radius = curve.Radius
            expected_circumference = 2 * math.pi * radius

            # Allow small tolerance for floating point comparison
            if abs(arc_length - expected_circumference) < 0.1:
                # This is a complete circle
                circles.append(curve)
```

Figure 41: Door Marker Detection

This mathematical verification ensures that only complete circles trigger door placement, preventing partial arcs from inadvertently creating openings. A similar approach identifies ellipses for window placement, with the system extracting the center point of each valid marker for subsequent processing.

Duplicate detection prevents multiple openings at the same location when markers exist in multiple categories or views. The system tracks marker centers with a distance tolerance of 0.01 feet, consolidating markers that represent the same intended opening position. This deduplication ensures clean, predictable results regardless of how templates are constructed.

4.5.7.2 WALL ASSOCIATION & HOST FINDING

The critical challenge in opening placement involves associating each marker with its intended host wall. The system implements a sophisticated nearest-wall algorithm that projects marker positions onto

wall centerlines, calculating distances while accounting for the three-dimensional nature of building models.

The wall association process begins by collecting all walls in the model through a filtered element collector. For each marker, the algorithm creates a test point at the wall's elevation, ensuring that vertical offsets don't interfere with horizontal distance calculations. This elevation normalization is crucial when markers are drawn at different Z-coordinates than their host walls.

Distance calculation employs projection mathematics to find the closest point on each wall's centerline:

```
def find_nearest_wall(point, walls):
    """Find the nearest wall to a given point"""
    for wall in walls:
        wall_curve = wall.Location.Curve
        wall_start = wall_curve.GetEndPoint(0)

        # Create test point at wall's Z level
        test_point = DB.XYZ(point.X, point.Y, wall_start.Z)

        # Project point onto wall line
        projection = wall_curve.Project(test_point)
        closest_point = projection.XYZPoint

        # Calculate distance in XY plane only
        xy_distance = math.sqrt(
            (test_point.X - closest_point.X)**2 +
            (test_point.Y - closest_point.Y)**2
        )
```

Figure 42: Wall Association Algorithm

The algorithm applies a generous 20-foot tolerance for door associations, accommodating various drawing styles where markers might be offset from walls for clarity. Windows use a similar but adjusted tolerance, reflecting their typically more precise positioning requirements.

4.5.7.3 ELEMENT CREATION & PLACEMENT

The actual creation of door and window instances leverages Revit's NewFamilyInstance API [33] with careful attention to hosting requirements and family-specific placement constraints. The system implements multiple placement strategies to accommodate different family types and hosting conditions.

For standard wall-hosted families, the system uses the primary placement method that specifies the host wall, insertion point, and structural type. The algorithm projects the marker center onto the wall centerline to determine the exact insertion point, ensuring proper alignment regardless of marker position accuracy. Level information is extracted from the host wall, maintaining proper floor associations for multi-story models.

When standard placement fails, often due to family-specific requirements, the system attempts alternative methods, including face-based placement for complex families. This multi-strategy approach

maximizes successful placement rates across diverse family libraries. For windows, the system additionally manages sill height parameters, extracting default values from family types or applying standard heights when parameters are unavailable.

4.5.7.4 SEQUENTIAL LABELLING & MAPPING

Following successful placement, each opening receives a sequential identifier with a corresponding visual annotation. Doors are labelled D1, D2, D3 in order of placement, while windows receive W1, W2, W3 designations. These labels are created as text notes positioned at the original marker centers, providing clear visual identification that corresponds to the numbering system.

The labelling system generates a comprehensive JSON mapping file that links labels to element IDs, family types, and placement parameters. This mapping enables the external UI to reference specific openings for modifications, such as "flip door D3" or "change family of window W2." The persistent mapping ensures that element references remain valid across sessions, supporting iterative design refinement through natural language commands.

The complete placement transaction wraps all operations within Revit's transaction framework, ensuring atomicity and enabling rollback if errors occur. The system provides detailed feedback about placement success rates, identifying any markers that couldn't be converted to openings due to missing walls or family compatibility issues.

4.5.8 SELECTION CYCLE SYSTEM

The Selection Cycle Panel provides sophisticated element navigation and modification capabilities that bridge the gap between the automated generation system and manual refinement needs. This panel implements an intelligent cycling mechanism that allows users to systematically traverse through all instances of specific element types—walls, doors, and windows—while providing immediate access to modification operations. The system serves dual purposes: enabling manual exploration and adjustment through PyRevit's interface, and processing targeted commands from the external UI for element-specific correction operations.

4.5.8.1 ELEMENT COLLECTION & SORTING STRATEGY

The selection system begins with comprehensive element collection that gathers all instances of the target category throughout the entire model. Unlike view-specific operations, the cycling mechanism operates globally, ensuring that no elements are overlooked regardless of their visibility in the current view. This global scope proves essential for comprehensive model review and modification operations that span multiple levels or building sections.

The collection process employs FilteredElementCollector with category-specific filters, distinguishing between element instances and types. For doors, the system filters using BuiltInCategory.OST_Doors;

for windows, OST_Windows; and for walls, the Wall class directly. Each collector explicitly excludes element types through WhereElementIsNotElementType(), ensuring that only placed instances appear in the cycling sequence.

Intelligent sorting enhances navigation logic by organizing elements in a spatially coherent sequence:

```
def collect_doors():
    """Collect all doors in the model"""
    elements.sort(key=lambda e: (
        e.LevelId.IntegerValue if hasattr(e, 'LevelId') else 0,
        e.Location.Point.X if hasattr(e.Location, 'Point') else 0,
        e.Location.Point.Y if hasattr(e.Location, 'Point') else 0
    ))
```

Figure 43: Element Sorting Strategy

This multi-key sorting first groups elements by level, then orders them by X-coordinate, and finally by Y-coordinate. The resulting sequence follows a logical progression through the building, making manual navigation intuitive and predictable. The sorting algorithm includes exception handling to maintain functionality even when elements lack expected properties.

4.5.8.2 DUAL-MODEL OPERATING SYSTEM

The selection panel implements a sophisticated dual-mode architecture that seamlessly switches between interactive cycling and external command processing. This flexibility allows the system to serve both manual users working directly in Revit (Figure 44) and automated processes controlled by the external UI.

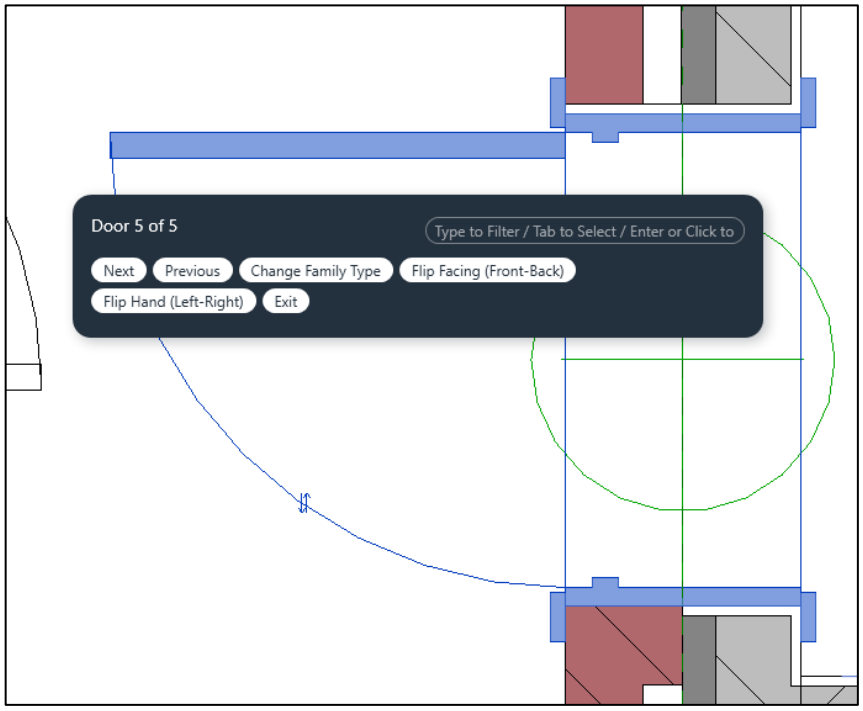

Figure 44: Element selection cycle for modification purposes

In interactive mode, the system presents a CommandSwitchWindow dialog that displays comprehensive element information and available actions. The dialog shows the current element's position in the sequence, its type designation, level assignment, orientation, and modification status. Users navigate through elements using Next and Previous buttons, with the selection automatically updating in the Revit view. Each selection triggers ShowElements() to center the view on the current element, ensuring visual context during navigation.

External command mode activates when the system detects a JSON command file in the temporary directory. The command processor parses the file to extract the action type, target element label, and any additional parameters. The system then uses the label mapping created during element generation to locate the specific element:

```
def find_door_by_label(label):
    """Find a door element by its label (D1, D2, etc.)"""
    if label in door_label_mapping:
        door_info = door_label_mapping[label]
        door_id = door_info.get('door_id')
        elem_id = DB.ElementId(int(door_id))
        door = doc.GetElement(elem_id)
        return door
```

Figure 45: Label-Based Element Resolution

This label-based identification enables natural language commands from the external UI, such as "flip door D3" or "change window W2 type," to be translated into specific element operations.

4.5.8.3 MODIFICATION OPERATIONS & TRANSACTION MANAGEMENT

The system implements a comprehensive suite of modification operations tailored to each element type. For doors and windows, operations include flipping orientation (both facing and hand), changing family types, moving along host walls, and deletion. Wall modifications support type changes, flipping orientation, and deletion. Each operation executes within a properly managed Revit transaction, ensuring model integrity and enabling undo functionality.

The flip operations leverage element-specific APIs to reverse orientation:

```
def flip_door_facing(door):
    """Flip the door facing (front-back)"""
    with revit.Transaction("Flip Door Facing"):
        if hasattr(door, 'flipFacing'):
            door.flipFacing()
            return True
```

Figure 46: Flip Operation Implementation

Movement operations calculate displacement vectors based on host wall orientation, converting user-specified distances from meters to Revit's internal units. The algorithm determines wall direction from

the host element's curve, then applies the movement perpendicular or parallel to this direction based on the command parameters.

Type changes present available family types through a selection dialog in interactive mode or match type names through pattern recognition in command mode. The system maintains family-type dictionaries that combine family and type names for clear identification, enabling users to distinguish between similar types from different families.

The response mechanism provides feedback for external commands by writing success status and operation details to a response JSON file. This bidirectional communication ensures that the external UI can verify operation completion and update its interface accordingly, maintaining synchronization between the external control system and the internal Revit model state.

4.5.9 ROOM PLACEMENT TOOL

The Room Placement Tool represents the final stage in the BIM generation pipeline, transforming the geometric framework of walls into semantically rich spatial entities. This tool leverages Revit's PlanTopology API [33] to detect closed wall boundaries and automatically generate room objects with intelligent naming based on dimensional analysis. The system bridges the gap between geometric construction and spatial programming, creating fully annotated rooms that enable area calculations, finish schedules, and other room-based analyses essential for architectural documentation.

4.5.9.1 BOUNDARY DETECTION THROUGH PLAN TOPOLOGY

The room placement process begins with topological analysis of the wall network to identify closed circuits that define habitable spaces. The system utilizes Revit's PlanTopology class, which constructs a mathematical graph representation of walls at a specific level, analyzing their connectivity to detect enclosed regions. This topological approach proves more robust than geometric intersection testing, as it inherently handles wall joins, corner conditions, and T-junctions that often complicate boundary detection algorithms.

The boundary detection algorithm operates at the active view's level, extracting the plan topology and iterating through its Circuits collection. Each circuit represents a potential room boundary formed by connected wall segments. The system validates each circuit by attempting to find an interior point using the GetPointInside() method, which employs ray-casting algorithms to determine points definitively within the enclosed region:

```python
def place_rooms_with_plan_topology():
    """Main function to detect boundaries and place rooms"""
    plan_topology = doc.get_PlanTopology(level)
    circuits = plan_topology.Circuits

    for circuit in circuits:
        uv_point = circuit.GetPointInside()
        if uv_point:
            room = doc.Create.NewRoom(level, uv_point)
```

Figure 47: Room Boundary Detection

The matching algorithm compares each room's measured dimensions against expected dimensions from global parameters, calculating a difference score that accounts for both normal and rotated orientations. This rotation handling proves essential as rooms may be oriented differently in the template than their parameter definitions suggest. A tolerance threshold of 500mm accommodates minor variations from construction adjustments while preventing incorrect matches.

4.5.9.2 DIMENSIONAL ANALYSIS & ROOM IDENTIFICATION

Following successful room creation, the system implements an intelligent naming algorithm that correlates room dimensions with global parameters to identify room types. This correlation process represents a crucial innovation, automatically assigning semantic meaning to geometric spaces based on their dimensional characteristics rather than requiring manual annotation.

The dimensional extraction process analyzes each room's boundary segments to calculate accurate internal dimensions. The algorithm accounts for wall thickness by querying the bounding walls' type parameters, ensuring that internal room dimensions are compared fairly with the center-to-center dimensions stored in global parameters:

```python
def get_room_dimensions(room):
    """Get the actual internal dimensions of a room"""
    boundary_segments = room.GetBoundarySegments(options)

    for segment_list in boundary_segments:
        for segment in segment_list:
            curve = segment.GetCurve()
            # Extract boundary points to calculate dimensions
```

Figure 48: Room Dimension Matching

4.5.9.3 AUTOMATIC TAGGING & DOCUMENTATION

The final phase implements automatic room tagging to provide visual identification and area documentation directly in the plan view. The tagging system creates a complete feedback loop, displaying both room names derived from dimensional analysis and calculated areas that verify compliance with requirements.

The tag placement algorithm positions tags at each room's location point, utilizing Revit's NewRoomTag API with careful handling of linked versus local rooms. The system constructs LinkElementId objects for local rooms, ensuring proper association between tags and their parent rooms:

```
def place_room_tags(doc, view, level, rooms_to_tag):
    """Place room tags in the view"""
    link_id = LinkElementId(room.Id)
    tag = doc.Create.NewRoomTag(link_id, uv_point, view.Id)
```

Figure 49: Room Tag Creation

Tag type selection leverages available room tag families in the project, automatically selecting appropriate types that display both name and area information. The system provides comprehensive feedback about tag placement success, including troubleshooting guidance for visibility issues related to view scale or graphics settings.

The Room Placement Tool completes the parametric generation workflow by transforming abstract geometry into meaningful architectural spaces. Through its integration of topological analysis, dimensional matching, and automatic documentation, the tool ensures that generated models contain not just walls and openings but properly defined rooms ready for quantity takeoffs, finish specifications, and comprehensive architectural documentation. This semantic enrichment represents the critical distinction between simple 3D geometry and true BIM models suitable for professional practice.

4.5.10 INTEGRATION WITH EXTERNAL UI THROUGH THE PROCESS ALL COMMANDS BUTTON

The Process All Commands button represents the orchestration hub of the entire parametric generation system, serving as the critical bridge that transforms queued commands from the external UI into coordinated execution of all previously described components. This master controller implements sophisticated command interpretation, state management, and sequential execution logic that ensures commands are processed in the correct order while maintaining contextual awareness across operations. Unlike individual button functionalities that operate in isolation, this integration layer manages dependencies, accumulates parameters, and triggers cascading operations that would be impossible through manual button activation.

4.5.10.1 COMMAND QUEUE PROCESSING & ORCHESTRATION

The Process All Commands button begins by establishing communication with the external UI through the file-based protocol, reading the JSON command queue from the Windows temporary directory. The system implements intelligent path resolution that navigates Revit's GUID-based temporary folders to locate the standard Windows temp directory where command files are deposited:

```
temp_path = os.environ.get('TEMP', '')
if temp_path and '\\' in temp_path:
    parts = temp_path.split('\\')
    if len(parts) > 1 and len(parts[-1]) == 36 and '-' in parts[-1]:
        temp_path = '\\'.join(parts[:-1])
```

Figure 50: Command File Path Resolution

This path normalization ensures consistent file access regardless of Revit's session-specific folder creation, maintaining reliable communication with the external UI across different execution contexts.

The orchestration engine maintains a comprehensive mapping between command names and their corresponding PyRevit button scripts, stored in the BUTTON_SCRIPTS dictionary. This mapping enables dynamic script invocation without hard-coded execution paths, allowing the system to adapt as new functionalities are added. Each command triggers the appropriate button script through Python's import mechanism, passing parameters through temporary JSON files that individual buttons read during execution.

The command processor implements a sophisticated execution strategy that goes beyond simple sequential processing. It recognizes command dependencies and automatically triggers related operations. For instance, after executing the 'lines_to_walls' command, the system automatically invokes ‘room generation’ without explicit user instruction, understanding that walls must exist before rooms can be placed. This intelligent cascading ensures that the model progresses through logical construction stages even when users provide incomplete command sequences.

4.5.10.2 STATEFUL PARAMETER ACCUMULATION

A crucial innovation in the Process All Commands implementation is the CommandState class, which maintains accumulated parameters across multiple commands. This stateful approach solves the challenge of commands that provide partial information, accumulating context until sufficient data exists to execute operations:

```
class CommandState:
    def __init__(self):
        self.area = None
        self.building_code = None
        self.wall_height = None

    def can_adjust_rooms(self):
        return self.area is not None and self.building_code is not None
```

Figure 51: Command State Management

When processing 'set_area' or 'set_building_code' commands, the system doesn't immediately attempt room adjustment. Instead, it accumulates these parameters in the state object, checking after each update whether sufficient information exists to proceed. Once both area and building code are specified, the

system automatically triggers the room adjustment workflow, synthesizing a complete parameter set from the accumulated state.

This stateful accumulation enables natural language interactions where users can specify requirements incrementally. A user might say "Create a 2-bedroom apartment" in one command, then "Make it 75 square meters" in another, and finally "Use German building code." The state management system accumulates these specifications, executing the appropriate operations once all necessary parameters are available.

4.5.10.3 DYNAMIC SCRIPT INVOCATION & PARAMETER INJECTION

The script execution mechanism implements sophisticated parameter injection that adapts to each button's expected input format. Different buttons expect parameters in different locations and formats, requiring the orchestrator to understand each button's interface:

```
def execute_button_script(command_name, parameters=None):
    if parameters:
        if command_name == 'select_template':
            param_file = os.path.join(temp_path, 'pyrevit_external_params.json')
        elif command_name.startswith('cycle_'):
            element_type = command_name.replace('cycle_', '')
            param_file = os.path.join(temp_path, f'{element_type[:-1]}_command.json')
        else:
            param_file = os.path.join(temp_path, 'button_parameters.json')
```

Figure 52: Dynamic Script Execution

This adaptive parameter routing ensures that each button receives its parameters in the expected format and location, maintaining compatibility with buttons designed for both standalone and integrated operation. The system writes parameters to specific files that buttons monitor, creating a seamless illusion that buttons are being activated with pre-configured settings.

The dynamic import mechanism uses Python's imp module to load button scripts at runtime, avoiding the need for static imports that would create rigid dependencies. This approach allows the system to discover and execute buttons dynamically, adapting to changes in the button structure without modification to the orchestrator code.

4.5.10.4 ELEMENT-SPECIFIC OPERATIONS THROUGH LABEL RESOLUTION

The Process All Commands button implements direct element manipulation capabilities that bypass the need for individual cycle button activation. When processing commands like 'flip_element', 'change_element_type', or 'delete_element', the system directly accesses element mapping files to resolve labels to Revit element IDs:

This direct manipulation capability significantly improves execution efficiency for modification commands, avoiding the overhead of launching cycle interfaces for simple operations. The system maintains transaction integrity by wrapping each modification in a Revit transaction, ensuring that operations can be undone and that model consistency is preserved.

4.5.10.5 RESPONSE GENERATION & FEEDBACK LOOP

The orchestrator maintains comprehensive execution tracking, generating detailed response files that inform the external UI about operation success or failure. Each command execution produces a result object containing a success status, descriptive messages, and timestamps. These results aggregate into a comprehensive response structure that includes summary statistics, enabling the external UI to provide accurate feedback about the generation process.

The response mechanism creates a closed feedback loop between the external UI and Revit, enabling sophisticated error recovery and user guidance. When operations fail, the detailed error messages help users understand what went wrong and how to correct it, whether through modified commands or manual intervention in Revit.

This integration layer transforms the collection of individual parametric tools into a cohesive system capable of interpreting complex natural language specifications and generating complete, code-compliant BIM models through coordinated execution of all system components.

5 EVALUATION AND DISCUSSION

5.1 INTRODUCTION

This chapter evaluates the Parametric Generation of Standardized Spaces system through systematic testing and analysis. The evaluation focuses on verifying that the system meets its stated objectives while identifying areas for improvement. Testing was conducted by the author using various apartment configurations and building codes to assess system performance, reliability, and output quality.

The evaluation methodology combines quantitative measurements of system performance with qualitative assessment of generated models. Each component of the system underwent individual testing before integrated workflow validation. This approach ensured that both technical functionality and practical usability were thoroughly examined.

5.2 TEST CASES AND RESULTS

5.2.1 TEST CASE DESIGN

The testing framework evaluated the system across multiple dimensions to verify its capabilities. Test cases covered apartment configurations ranging from studio units to four-bedroom layouts. Each configuration was tested with different total area requirements and building codes to ensure consistent performance across varied parameters.

The test suite included fifteen primary test cases. Studio apartments were tested at 40, 50, and 60 square meters. One-bedroom units were evaluated at 50, 60, and 70 square meters. Two-bedroom apartments underwent testing at 70, 80, and 90 square meters. Three-bedroom configurations were tested at 90, 100, and 110 square meters. Four-bedroom layouts were evaluated at 110, 120, and 130 square meters.

Each primary test case was executed with three different building codes: Slovenia, USA, and Germany. These countries were selected for their contrasting regulatory approaches. Slovenia represents European Union standards with moderate requirements. The USA demonstrates imperial measurement handling and performance-based codes. Germany exemplifies strict prescriptive regulations with detailed dimensional specifications.

5.2.2 GENERATION PERFORMANCE RESULTS

The system demonstrated consistent generation times across all tested configurations. Complete BIM models are generated within 2 to 10 seconds from initial command execution. This timeframe included template loading, parameter adjustment, building code application, wall generation, opening placement, and room creation [34].

Two-bedroom apartments averaging 80 square meters showed the most consistent performance. These units are generated within approximately 5 seconds regardless of the selected building code. The generation process maintained stability throughout testing with no crashes or incomplete generations observed.

Studio apartments generated fastest at approximately the 5-second mark due to their simpler spatial arrangements. Four-bedroom units required up to 8 seconds. The system handled area adjustments smoothly, successfully scaling room dimensions while maintaining building code compliance.

Wall generation from template lines succeeded in all test cases. The system correctly created 3D walls with proper joins at corners and intersections. Door and window placement achieved a 100% success rate when circular and elliptical markers were properly defined in templates. Room generation accurately detected closed boundaries and created room elements with appropriate names and area calculations.

5.2.3 NATURAL LANGUAGE PROCESSING ACCURACY

The natural language interface successfully interpreted standard architectural specifications in all tested formats. Commands like "create a 2-bedroom apartment with 75 square meters" were correctly parsed into structured parameters. The system handled variations in phrasing, accepting both "2BR" and "two bedrooms" as equivalent inputs.

Area specifications showed robust handling across different formats. The system correctly interpreted "75 sqm," "75 square meters," and "75 m^2" as identical requirements. Unit conversion from imperial measurements functioned correctly, with "800 square feet" properly converting to approximately 74 square meters.

Building code selection through natural language proved reliable. Commands specifying "Slovenian building code" or simply "Slovenia" both triggered the correct code application. The system appropriately handled partial matches, recognizing "German" as referring to Germany's building code.

Element modification commands demonstrated precise execution. Instructions like "flip door D3" successfully identified and modified the specified element. The label-based reference system enabled accurate element selection without requiring users to understand Revit's selection methods.

5.2.4 USER WORKFLOW VALIDATION

The file-based communication protocol operated reliably throughout testing. Command queue files were consistently detected and processed by the PyRevit plugin. No data corruption or loss occurred during file transfer between the external UI and Revit.

The status update mechanism provided timely feedback about operation completion. Status files were created within one second of operation completion and successfully detected by the external UI. This feedback loop enabled users to understand system state without accessing Revit directly.

The element numbering system correctly labelled all generated walls, doors, and windows. Labels remained persistent across Revit sessions through JSON mapping files. The sequential numbering (W1, W2, D1, D2) provided an intuitive reference system for modifications.

Command accumulation in the external UI worked as designed. Users could build complex specifications through multiple inputs before execution. The system correctly maintained command order, ensuring that dependent operations executed in proper sequence.

5.3 COMPARATIVE ANALYSIS WITH EXISTING TOOLS

5.3.1 COMPARISON FRAMEWORK

The comparative analysis evaluates the developed system against existing solutions using five key criteria. Generation speed measures the time from specification to completed model. Accessibility evaluates how easily non-technical users can operate the system. BIM integration assesses the quality and completeness of generated models. Code compliance examines how building regulations are handled. Customization flexibility considers the range of possible variations.

5.3.2 ANALYSIS AGAINST ACADEMIC SOLUTIONS

The HABX Optimizer generates apartment layouts within one minute [6], comparable to this system's performance. However, HABX requires users to specify constraints mathematically, limiting accessibility to technical users [6]. The grid-based discretization in HABX can produce less natural room arrangements compared to the template-based approach used here.

House-GAN demonstrates impressive machine learning capabilities for layout generation [7]. It produces diverse layouts from bubble diagrams, but cannot guarantee dimensional accuracy or building code compliance [7]. The system developed in this research provides deterministic, code-compliant results that House-GAN cannot match.

The DAT system from Warsaw University handles complete apartment blocks, including massing optimization [8]. While more comprehensive in scope, DAT requires approximately 25 minutes for generation. It also depends on Grasshopper's expertise, reducing accessibility compared to natural language input.

5.3.3 ANALYSIS AGAINST COMMERCIAL TOOLS

TestFit excels at rapid feasibility studies with excellent financial analysis integration [12]. However, it focuses on standardized solutions with limited customization options. TestFit generates schematic layouts rather than detailed BIM models, requiring additional work for construction documentation [35]. Moreover, it does not take building code compliance into account like this system does.

Skema.ai provides sophisticated design reuse capabilities through its learning engine [13]. The platform requires extensive prior project data to function effectively. Small firms without project libraries cannot fully utilize Skema's capabilities. The natural language system developed here operates independently without requiring historical data.

Recent AI-powered tools like Maket.ai promise natural language input but rely on external AI services [15]. These dependencies create reliability and privacy concerns absent in the developed system. Most AI tools generate visualizations rather than construction-ready BIM models, limiting professional utility [36].

5.3.4 UNIQUE ADVANTAGES IDENTIFIED

The system uniquely combines natural language accessibility with professional BIM output. The integration of building code compliance during generation rather than post-checking represents a significant advancement.

The element numbering system enables precise post-generation modifications using natural language, which is commonly unavailable in other tools. Users can reference specific elements through natural language without understanding BIM selection methods. This capability bridges the gap between automated generation and manual refinement.

The zero-dependency architecture ensures consistent operation without internet connectivity or external services. This reliability is critical for professional deployment, where service interruptions cannot be tolerated.

5.4 SWOT ANALYSIS

5.4.1 STRENGTHS

5.4.1.1 NATURAL LANGUAGE ACCESSIBILITY

- Zero-dependency custom language processor requiring no external AI services
- Accepts conversational input: "create a 2-bedroom apartment with 75 square meters"
- Handles variations in terminology (2BR, two bedrooms, 2-bedroom)
- Enables element-specific modifications through simple commands ("flip door D4")

- Maintains consistent performance without internet connectivity

5.4.1.2 PROFESSIONAL BIM GENERATION

- Produces models with full parametric relationships
- Generates properly classified elements (walls, doors, windows) with correct metadata
- Creates accurate room boundaries with automatic area calculations
- Preserves template parametric relationships for post-generation adjustments
- Eliminates the translation step between conceptual design and documentation

5.4.1.3 INTEGRATED BUILDING CODE COMPLIANCE

- Proactive compliance during generation rather than post-checking
- Verified requirements from ten countries embedded in system logic
- Handles both metric and imperial measurement systems
- Prevents creation of non-compliant designs automatically

5.4.1.4 SPEED AND RELIABILITY

- Complete model generation in 5-10 seconds
- File-based communication ensures deployment flexibility
- Element numbering system enables precise modifications
- No network configuration requirements
- Consistent performance across different configurations

5.4.2 WEAKNESSES

5.4.2.1 PLATFORM DEPENDENCIES

- Restricted to Autodesk Revit on Windows operating systems
- Requires a full Revit license for each user (significant cost barrier)
- Dependent on PyRevit framework continuity
- Cannot serve organizations using ArchiCAD, Vectorworks, or other platforms

5.4.2.2 GEOMETRIC LIMITATIONS

- Only handles orthogonal, single-level apartment layouts
- Cannot process curved walls or angled rooms
- No support for multi-story units or vertical circulation
- Unable to accommodate irregular site boundaries

5.4.2.3 LACK OF DESIGN INTELLIGENCE

- No optimization for daylight distribution or view quality
- Cannot learn from user preferences or past projects

- Static rule application without improvement over time
- Missing circulation efficiency analysis

5.4.3 OPPORTUNITIES

5.4.3.1 BUILDING TYPOLOGY EXPANSION

- Office layouts with workstation configurations
- Hotel rooms with high standardization potential
- Student housing and dormitory designs
- Healthcare facilities (examination rooms, patient rooms)
- Retail spaces with modular layouts

5.4.3.2 TECHNOLOGICAL INTEGRATION

- Cloud deployment for universal browser-based access
- Cost estimation database connection for instant pricing
- LLM integration with the MCP server database
- Structural and MEP system coordination
- IoT sensor data for occupancy-based optimization
- API development for third-party integrations

5.4.3.3 GEOGRAPHIC MARKET GROWTH

- Additional building codes multiply addressable markets
- Regional partnerships for code verification
- Government adoption for permit automation
- Educational use in architecture programs
- International firms requiring multi-jurisdiction support

5.4.3.4 PLATFORM EVOLUTION

- IFC-based generation for platform independence
- Web assembly for browser-native operation
- Mobile applications for on-site modifications
- Open-source community development potential

5.4.4 THREATS

5.4.4.1 AI TECHNOLOGY DISRUPTION

- Large language models are improving exponentially in spatial reasoning
- Major tech companies investing billions in architectural AI
- AI-native BIM platforms potentially obsoleting parametric approaches

- General-purpose AI achieving both creativity and compliance
- Open-source AI tools providing free alternatives

5.4.4.2 PLATFORM VENDOR RISKS

- Autodesk is developing native competing features
- API changes breaking functionality without warning
- Shift to cloud-only architectures, preventing file-based integration
- PyRevit discontinuation or major breaking changes
- Subscription model changes affecting deployment

5.4.4.3 REGULATORY RESISTANCE

- Licensing boards restricting automated design tools
- Unresolved liability for automated design errors
- Building officials rejecting automated submissions
- Professional organizations opposing automation

5.4.4.4 MARKET DYNAMICS

- Economic downturns are reducing construction activity
- Subscription fatigue favours one-time purchases
- Consolidation, eliminating smaller tool vendors
- Commoditization of design automation reduces value
- Skills gap as architects lack experience with automated tools

5.4.4.5 COMPETITIVE LANDSCAPE

- Established vendors adding similar capabilities
- Startups with venture funding are scaling rapidly
- Academic institutions releasing free alternatives
- Industry consortiums developing open standards
- Client direct-to-construction platforms bypassing architects

5.5 DISCUSSION OF FINDINGS

5.5.1 ACHIEVEMENTS OF RESEARCH OBJECTIVES

The system successfully achieved the primary objective of bridging user intent and BIM generation through natural language. Testing confirmed that non-technical users could generate professional-quality models without understanding parametric design or BIM concepts. The natural language processor accurately interpreted architectural requirements without external AI dependencies.

Building code compliance integration exceeded initial expectations. The system not only verified compliance but also actively prevented non-compliant generation. This proactive approach eliminated the iterative checking typical in manual processes. The successful incorporation of ten different building codes demonstrated the scalability of the compliance framework.

The template-based parametric engine proved capable of generating fully detailed BIM models directly in Revit. Generated models included proper element classification, parametric relationships, and semantic information required for professional use. The quality matched manually created models while requiring a fraction of the time.

The file-based communication protocol provided reliable data exchange without network complexity. This approach simplified deployment while maintaining responsiveness adequate for interactive design. The decision to avoid network protocols proved correct given enterprise security constraints.

5.5.2 IMPLICATIONS OF ARCHITECTURAL PRACTICE

The dramatic time reduction from weeks to minutes fundamentally changes project economics. Architects can explore numerous design options within single client meetings. This capability transforms client interaction from presentation to collaboration. Real-time generation enables immediate response to feedback rather than scheduling follow-up meetings.

Small investors gain access to professional design tools previously requiring architectural expertise. They can independently evaluate site potential before engaging architects. This democratization could unlock development opportunities in underserved markets. The system empowers stakeholders traditionally excluded from early design decisions.

Architects benefit from the automation of repetitive technical tasks. Junior staff can focus on design thinking rather than manual drafting. Senior architects can handle more projects by delegating routine generation tasks. The profession can evolve toward higher-value creative and strategic services.

5.5.3 BUILDING CODE INTEGRATION IMPACT

Proactive compliance checking during generation eliminates costly post-design corrections. Architects gain confidence that the generated designs will pass regulatory review. The system reduces liability exposure from code violations.

Multi-jurisdictional support enables firms to work across borders efficiently. The same system handles metric and imperial requirements seamlessly. Cultural preferences embedded in different codes are automatically respected. This capability is increasingly valuable as architectural practice globalizes.

5.6 LIMITATIONS

5.6.1 CURRENT SYSTEM LIMITATIONS

The system currently operates only within Revit on Windows platforms. Organizations using ArchiCAD, Vectorworks, or other BIM software cannot utilize the system. This platform dependency limits potential adoption. Future versions should explore IFC-based generation for platform independence.

Apartment layouts remain restricted to orthogonal geometries on single levels. The system cannot handle curved walls, angled rooms, or multi-story units. Many contemporary designs exceed these geometric constraints. Expanding geometric capabilities would significantly broaden applicability.

The template library contains limited variations for each apartment configuration. Users may find available options insufficient for specific projects. Creating new templates requires Revit expertise, reducing accessibility. A template creation interface would enable broader participation.

The system lacks optimization algorithms for qualitative design aspects. Generated layouts meet requirements but may not maximize daylight or minimize circulation. Adding multi-objective optimization could improve design quality. Machine learning could help identify optimal solutions.

5.6.2 METHODOLOGICAL LIMITATIONS

Testing was conducted solely by the system developer. Independent user testing would provide a more objective evaluation. Different users might encounter issues not discovered during development. Broader testing would validate usability claims more thoroughly.

The evaluation occurred in controlled development environments. Production deployment might reveal performance or compatibility issues. Real-world projects could present requirements exceeding current capabilities. Field testing would provide valuable insights for system refinement.

Comparative analysis relied on published information about other systems. Direct side-by-side testing would provide more accurate comparisons. Some capabilities of commercial tools may not be publicly documented. Hands-on evaluation would strengthen comparative claims.

6 CONCLUSION

6.1 SUMMARY OF RESEARCH FINDINGS

This thesis developed a parametric generation system that automates the creation of code-compliant apartment BIM models through natural language commands. The system consists of three integrated components: an external user interface with custom natural language processing, a PyRevit plugin for BIM generation, and a file-based communication protocol connecting them. Testing demonstrated that the system generates complete apartment models in 10-30 seconds, compared to the weeks required by traditional methods [1].

The research validated that natural language interfaces can effectively translate architectural requirements into BIM operations without external AI dependencies. The custom-built language processor achieved consistent accuracy in interpreting commands across various phrasings and formats. The building code compliance system successfully enforced regulations from ten countries during the generation, preventing non-compliant designs rather than identifying violations afterward.

6.2 ACHIEVING THE RESEARCH OBJECTIVES

6.2.1 INTERFACE ACCESSIBILITY OBJECTIVES

Objective O1.1 sought to create a natural language interface for architectural requirements. The implemented system successfully interprets commands like "create a 2-bedroom apartment with 75 square meters" without requiring technical knowledge. The custom processor handles variations in terminology and maintains consistent performance without internet connectivity.

Objective O1.2 aimed for post-generation modification through natural language. The element numbering system enables users to modify specific components using commands like "flip door D3." This capability was validated through the successful execution of various modification commands during testing.

Objective O1.3 required demonstrating accessibility to non-experts while maintaining professional precision. The system allows users without BIM knowledge to generate models that match the quality of manually created ones. The generated models include proper element classification and parametric relationships required for professional use.

6.2.2 TECHNICAL IMPLEMENTATION OBJECTIVES

Objective O2.1 called for a template-based parametric engine generating detailed BIM models. The system successfully produces complete apartment models with walls, doors, windows, and rooms

directly in Revit. Each element maintains proper classification and metadata for construction documentation.

Objective O2.2 established reliable file-based communication between components. The JSON-based protocol achieved consistent data transfer without corruption or loss across all test cases. The approach avoided network complexity while maintaining adequate responsiveness.

Objective O2.3 implemented area distribution algorithms for room balancing. The system automatically adjusts room dimensions to meet total area requirements while maintaining building code minimums. The algorithm successfully scaled rooms proportionally across all tested configurations.

6.2.3 COMPLIANCE AND VALIDATION OBJECTIVES

Objective O3.1 developed building code compliance for ten countries. The system incorporates verified requirements from Slovenia, USA, Germany, France, Spain, Australia, Netherlands, Norway, Portugal, and Bangladesh. Compliance checking occurs during generation, not as post-validation.

Objective O3.2 validated effectiveness through multiple apartment configurations. Testing covered studio to four-bedroom layouts across different building codes. All generated models met their respective code requirements.

Objective O3.3 evaluated time savings compared to manual processes. The system generates models in under one minute versus 2-4 weeks for traditional preliminary design [1]. This represents a time reduction exceeding 99% for initial concept generation.

6.3 CRITICAL ASSESSMENT OF ACHIEVEMENTS

The system achieved its core goal of making BIM generation accessible through natural language. However, several aspects merit critical examination. The natural language processor, while functional, relies on pattern matching rather than true semantic understanding. This approach works for standard architectural specifications but may struggle with unconventional or ambiguous requests.

The building code compliance system successfully prevents basic violations but cannot handle performance-based requirements or local amendments. The current implementation assumes the most restrictive interpretation of ambiguous requirements, potentially over-constraining designs. Real-world projects often involve variance requests and alternative compliance paths that the system cannot accommodate.

The template-based approach ensures quality but limits creativity. Users cannot generate truly novel layouts, only variations of predefined patterns. While this constraint aligns with the focus on standardized spaces, it restricts the system's applicability to projects requiring unique spatial solutions.

6.4 LIMITATIONS

The system operates exclusively within Revit on Windows platforms. This dependency excludes organizations using alternative BIM software and creates vendor lock-in risks. The PyRevit framework dependency adds another potential failure point if the project discontinues.

Geometric capabilities remain limited to orthogonal, single-level apartments. The system cannot generate curved walls, angled rooms, or multi-story units. These constraints exclude many contemporary residential designs from the system's scope.

Testing occurred only through developer evaluation without independent user studies. The absence of testing in production environments means real-world performance remains unverified. Different users might encounter usability issues not discovered during development.

The system lacks learning capabilities or optimization algorithms. Generated layouts meet requirements but don't optimize for qualitative factors like daylight or circulation efficiency. Each generation starts fresh without benefiting from previous successes or user preferences.

6.5 CONTRIBUTIONS TO KNOWLEDGE

6.5.1 ACADEMIC CONTRIBUTIONS

This research demonstrates that domain-specific natural language processing can achieve comparable results to general AI systems for architectural applications. The custom interpreter provides predictable, debuggable behaviour essential for professional tools [37]. This finding suggests that specialized interpreters may be preferable to general-purpose AI for safety-critical domains.

The integration of building code compliance as generative constraints rather than post-generation validation represents a methodological contribution. This approach could be applied to other domains where regulatory compliance shapes design decisions.

The file-based communication architecture proves that complex inter-process communication can be achieved without network protocols. This finding has implications for enterprise software deployment, where network restrictions complicate traditional architectures.

6.5.2 PRACTICAL CONTRIBUTIONS

The system provides immediate value to architectural practice by automating repetitive tasks. Architects can generate multiple design options during client meetings, transforming the design process from sequential to interactive.

For smaller investors and developers, the system democratizes access to professional design tools. They can evaluate site potential independently before engaging architects, potentially unlocking projects previously deemed unfeasible.

The building code integration across ten countries demonstrates the feasibility of multi-jurisdictional compliance systems. This capability becomes increasingly valuable as architectural practice globalizes.

6.6 FUTURE RESEARCH DIRECTIONS

The following research directions could significantly enhance the system's capabilities and address current limitations:

6.6.1 MULTI-STORY BUILDING GENERATION

- Extending the system from single apartments to complete multi-story residential buildings represents the most transformative enhancement
- Vertical circulation elements, including stairs, elevators, and fire escapes, would need integration with apartment layout generation
- The system could implement staggered apartment arrangements to optimize unit mix and building efficiency across floors
- Floor-to-floor relationships would require handling of structural elements, service shafts, and vertical MEP routing
- Building-level optimization could balance unit types, ensure code-compliant egress paths, and maximize rentable area
- Integration with structural systems would ensure load path continuity and column placement coordination across levels

6.6.2 PLATFORM INDEPENDENCE THROUGH IFC STANDARDS

- Developing direct IFC (Industry Foundation Classes) generation would eliminate the current Revit dependency [38]
- IFC output would enable the system to work with any BIM platform, including ArchiCAD, Vectorworks, and Allplan [39]
- Open BIM standards would ensure long-term sustainability independent of vendor-specific API changes
- The system could generate IFC models directly from templates without requiring intermediate platform-specific conversion

6.6.3 LARGE LANGUAGE MODEL INTEGRATION

- Implementing LLM integration through Model Context Protocol (MCP) server connections would revolutionize the system's natural language capabilities
- MCP servers could provide contextual understanding of complex architectural requirements beyond pattern matching [40]
- This architecture would enable the system to interpret nuanced design intentions and ambiguous specifications
- The LLM could generate explanations for design decisions, improving transparency and user trust

6.6.4 AUTOMATED BUILDING CODE EXPANSION

- LLM integration could enable automatic extraction and processing of building codes from new jurisdictions
- The system could read official regulatory documents and generate structured JSON files
- Natural language processing could interpret ambiguous regulatory language and performance-based requirements
- This capability would allow rapid expansion to new markets without manual code extraction

6.6.5 SITE-CONSTRAINED GENERATION

- Developing algorithms to generate BIM models within irregular site boundaries would address a major limitation
- Integration with GIS data could enable automatic site analysis and constraint identification
- The system could optimize building placement considering setbacks, easements, and topography
- Site-specific factors like solar orientation could influence apartment layout generation

6.6.6 DYNAMIC TEMPLATE EVOLUTION

- LLM-powered template modification could adapt base templates to specific client requirements
- The system could learn from successful modifications to evolve template libraries automatically
- Machine learning could identify patterns in client preferences to suggest template improvements [41]
- This capability would balance standardization benefits with customization needs

6.6.7 ADDITIONAL RESEARCH PRIORITIES

- Investigation of multi-objective optimization algorithms for balancing competing design criteria

- Development of performance simulation integration for energy and daylighting analysis
- Exploration of generative AI for creating novel templates while maintaining code compliance
- Study of collaborative design protocols enabling multiple stakeholders to contribute simultaneously

These research directions would transform the system from a specialized tool into a comprehensive design platform capable of handling diverse architectural challenges while maintaining the accessibility that defines its core value

6.7 FINAL REMARKS

This research addressed a specific problem: the inaccessibility of BIM tools for non-technical stakeholders in residential design. The developed system provides a proof of concept that natural language interfaces can bridge this gap without sacrificing professional standards or regulatory compliance. The achievements are concrete: functional software that generates code-compliant BIM models in seconds rather than weeks.

The limitations are equally clear. The system handles only standardized apartment layouts on single levels within one specific BIM platform. It cannot optimize designs or learn from experience. These constraints define the system's current scope rather than fundamental barriers.

Most importantly, this proof of concept establishes a foundation for the future of automated BIM generation. The methodologies developed here—from requirement interpretation to parametric control to compliance verification—demonstrate that comprehensive building model automation is achievable. Future systems could build upon this framework to generate entire multi-story buildings, handle complex geometries, and adapt to diverse architectural programs. The research validates that automation can preserve architectural quality while dramatically reducing production time.

This work ultimately demonstrates that the vision of accessible BIM automation is not merely aspirational but technically feasible. The system shows how specialized tools can augment rather than replace architectural expertise, enabling professionals to focus on design quality while technology handles production mechanics. The transition from weeks to seconds in model generation represents more than efficiency gains—it suggests a fundamental shift in how architectural services could be delivered. The proof of concept presented here provides both the technical framework and practical evidence needed to pursue this transformation.

REFERENCES

1. McGrawHill Construction (2014) Smart Market Report the Business Value of BIM for Construction in Major Global Markets. McGraw Hill Construction, New York. - References—Scientific Research Publishing. (n.d.). Retrieved 5 September 2025, from https://www.scirp.org/reference/referencespapers?referenceid=2081256

2. The Role of BIM in Simplifying Construction Permits in Kuwait | Request PDF. (n.d.). ResearchGate. Retrieved 5 September 2025, from https://www.researchgate.net/publication/315873703_The_Role_of_BIM_in_Simplifying_Construction_Permits_in_Kuwait

3. Veloso, P., Celani, G., & Scheeren, R. (2018). From the generation of layouts to the production of construction documents: An application in the customization of apartment plans. Automation in Construction, 96, 224–235. https://doi.org/10.1016/j.autcon.2018.09.013

4. Wang, X.-Y., Yang, Y., & Zhang, K. (2018). Customization and generation of floor plans based on graph transformations. Automation in Construction, 94, 405–416. https://doi.org/10.1016/j.autcon.2018.07.017

5. Flemming, U. (1978). Wall Representations of Rectangular Dissections and Their Use in Automated Space Allocation. Environment and Planning B: Planning and Design, 5(2), 215–232. https://doi.org/10.1068/b050215

6. Laignel, G., Pozin, N., Geffrier, X., Delevaux, L., Brun, F., & Dolla, B. (2021). Floor plan generation through a mixed constraint programming-genetic optimization approach. Automation in Construction, 123, 103491. https://doi.org/10.1016/j.autcon.2020.103491

7. Nauata, N., Chang, K.-H., Cheng, C.-Y., Mori, G., & Furukawa, Y. (2020). House-GAN: Relational Generative Adversarial Networks for Graph-constrained House Layout Generation (No. arXiv:2003.06988). arXiv. https://doi.org/10.48550/arXiv.2003.06988

8. Jansen, I., Piątek, Ł., Cygan, M., Hlebowicz, J., Krajewski, B., Miszewicz, A., Nowacka, A., Roguski, B., & Szabelewska, M. (2023). Automating the Architectural Design of Apartment Point Block. 491–500. https://doi.org/10.52842/conf.caadria.2023.2.491

9. Camozzato, D., Dihl, L., Silveira, I., Marson, F., & Musse, S. R. (2015). Procedural floor plan generation from building sketches. The Visual Computer, 31(6), 753–763. https://doi.org/10.1007/s00371-015-1102-2

10. Eastman, C., Teicholz, P., Sacks, R. and Liston, K. (2011) BIM Handbook A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers, and Contractors. John Wiley & Sons, Hoboken. - References—Scientific Research Publishing. (n.d.). Retrieved 5 September 2025, from https://www.scirp.org/reference/referencespapers?referenceid=1986720

11. Guo, Z., & Li, B. (2017). Evolutionary approach for spatial architecture layout design enhanced by an agent-based topology finding system. Frontiers of Architectural Research, 6(1), 53–62. https://doi.org/10.1016/j.foar.2016.11.003

12. TestFit: Real Estate Feasibility Platform. (n.d.). Retrieved 1 September 2025, from https://www.testfit.io/

13. Skema. (n.d.). Skema. Retrieved 1 September 2025, from https://www.skema.ai

14. Cómo encontrar y descargar la extensión Roombook para Revit. (n.d.). Retrieved 1 September 2025, from https://www.autodesk.com/es/support/technical/article/caas/sfdcarticles/sfdcarticles/ESP/How-to-find-and-download-the-Roombook-extension-for-Revit.html

15. Generative Design | Architecture Design Software | Maket. (n.d.). Retrieved 1 September 2025, from https://www.maket.ai/

16. Wu, W., Fan, L., Liu, L., & Wonka, P. (2018). MIQP-based Layout Design for Building Interiors. Computer Graphics Forum, 37(2), 511–521. https://doi.org/10.1111/cgf.13380

17. OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., … Zoph, B. (2024). GPT-4 Technical Report (No. arXiv:2303.08774). arXiv. https://doi.org/10.48550/arXiv.2303.08774

18. Rodrigues, E., Sousa-Rodrigues, D., Teixeira de Sampayo, M., Gaspar, A. R., Gomes, Á., & Henggeler Antunes, C. (2017). Clustering of architectural floor plans: A comparison of

shape representations. Automation in Construction, 80, 48–65. https://doi.org/10.1016/j.autcon.2017.03.017

19. Zeng, P., Yin, J., Gao, Y., Li, J., Jin, Z., & Lu, S. (2025). Comprehensive and Dedicated Metrics for Evaluating AI-Generated Residential Floor Plans. Buildings, 15(10), 1674. https://doi.org/10.3390/buildings15101674
20. Data-Driven Design and Construction: 25 Strategies for Capturing, Analyzing and Applying Building Data | Wiley. (n.d.). Wiley.Com. Retrieved 13 September 2025, from https://www.wiley.com/en-ca/Data-Driven+Design+and+Construction%3A+25+Strategies+for+Capturing%2C+Analyzing+and+Applying+Building+Data-p-9781118898703
21. Succar, B. (2009). Building information modelling framework: A research and delivery foundation for industry stakeholders. Automation in Construction, 18(3), 357–375. https://doi.org/10.1016/j.autcon.2008.10.003
22. Sommerville, I. (2016). Software engineering (Tenth edition). Pearson.
23. ECMA-404—Ecma International. (n.d.). Retrieved 5 September 2025, from https://ecma-international.org/publications-and-standards/standards/ecma-404/
24. (PDF) Flexible housing: The means to the end. (2025). ResearchGate. https://doi.org/10.1017/S1359135505000345
25. pyRevit. (n.d.). Pyrevitlabs on Notion. Retrieved 5 September 2025, from https://pyrevitlabs.notion.site/
26. Autodesk Revit | Get Prices & Buy Official Revit Software. (n.d.). Retrieved 5 September 2025, from https://www.autodesk.com/products/revit/overview
27. PDF. (n.d.). Retrieved 5 September 2025, from https://dn790001.ca.archive.org/0/items/bme-vik-konyvek/Software%20Engineering%20-%20Ian%20Sommerville.pdf
28. tkinter—Python interface to Tcl/Tk. (n.d.). Python Documentation. Retrieved 5 September 2025, from https://docs.python.org/3/library/tkinter.html

29. Mastering Regular Expressions, 3rd Edition. (n.d.). O'Reilly Online Learning. Retrieved 5 September 2025, from https://www.oreilly.com/library/view/mastering-regular-expressions/0596528124/

30. ISO 80000-1:2009. (n.d.). ISO. Retrieved 5 September 2025, from https://www.iso.org/standard/30669.html

31. dotnet-bot. (n.d.). FileSystemWatcher Class (System.IO). Retrieved 5 September 2025, from https://learn.microsoft.com/en-us/dotnet/api/system.io.filesystemwatcher?view=net-9.0

32. 2021 International Building Code (IBC). (n.d.). Retrieved 5 September 2025, from https://codes.iccsafe.org/content/IBC2021P2

33. Revit API 2023. (n.d.). Retrieved 5 September 2025, from https://www.revitapidocs.com/2023/

34. Bortoluzzi, B., Efremov, I., Medina, C., Sobieraj, D., & McArthur, J. J. (2019). Automating the creation of building information models for existing buildings. Automation in Construction, 105, 102838. https://doi.org/10.1016/j.autcon.2019.102838

35. Clayton, M. J., Warden, R. B., & Parker, T. W. (2002). Virtual construction of architecture using 3D CAD and simulation. Automation in Construction, 11(2), 227–235. https://doi.org/10.1016/S0926-5805(00)00100-X

36. Artificial intelligence in architecture: Generating conceptual design via deep learning | Request PDF. (2025). ResearchGate. https://doi.org/10.1177/1478077118800982

37. Aish, R., & Woodbury, R. (2005). Multi-level Interaction in Parametric Design. In A. Butz, B. Fisher, A. Krüger, & P. Olivier (Eds), Smart Graphics (pp. 151–162). Springer. https://doi.org/10.1007/11536482_13

38. Industry Foundation Classes (IFC)—buildingSMART International. (2024, November 13). https://www.buildingsmart.org/standards/bsi-standards/industry-foundation-classes/

39. (PDF) The IFC Standard—A Review of History, Development, and Standardization. (2025). ResearchGate. https://www.researchgate.net/publication/252069448_The_IFC_Standard_-_A_Review_of_History_Development_and_Standardization

40. What is the Model Context Protocol (MCP)? (n.d.). Model Context Protocol. Retrieved 14 September 2025, from https://modelcontextprotocol.io/docs/getting-started/intro

41. Design with shape grammars and reinforcement learning | Request PDF. (2025). ResearchGate. https://doi.org/10.1016/j.aei.2012.12.004

APPENDICES

APPENDIX A: BUILDING CODE REQUIREMENTS COMPARISON

This appendix presents a comprehensive comparison of minimum spatial requirements across the ten countries supported by the Parametric Generation of Standardized Spaces system. All measurements are in metric units for consistency.

MINIMUM ROOM AREA REQUIREMENTS (m²)

Room Type	Slovenia	USA	Germany	France	Spain	Australia	Netherlands	Norway	Portugal	Bangladesh
Living Room	16.0	11.0	20.0	18.0	14.0	12.0	15.0	15.0	10.0	12.0
Master Bedroom	12.0	9.3	14.0	11.0	10.0	10.0	11.0	9.0	10.5	9.0
Secondary Bedroom	8.0	7.0	10.0	9.0	8.0	8.0	8.0	7.0	9.0	7.5
Kitchen	6.0	5.0	8.0	7.0	5.0	5.0	6.0	5.0	6.0	4.5
Bathroom	3.5	3.5	4.0	3.0	3.0	3.0	3.5	3.0	3.5	2.5

Table 1: Minimum room area requirements as specified in each country's building regulations

ADDITIONAL SPATIAL REQUIREMENTS

Parameter	Slovenia	USA	Germany	France	Spain	Australia	Netherlands	Norway	Portugal	Bangladesh
Minimum Ceiling Height (m)	2.50	2.40	2.50	2.50	2.50	2.40	2.60	2.40	2.40	2.75
Minimum Corridor Width (m)	1.00	0.90	1.00	0.90	0.80	1.00	0.90	0.90	1.00	0.90
Minimum Door Width (m)	0.80	0.80	0.80	0.80	0.75	0.80	0.85	0.80	0.80	0.75
Min. Total Area 1BR Apt (m²)	35	30	40	35	30	35	35	30	35	28
Min. Total Area 2BR Apt (m²)	55	50	60	55	50	55	55	50	52	45

Table 2: Additional dimensional and spatial requirements for residential

Note: All values extracted from official building codes and regulations as of January 2025. USA measurements converted from imperial units. Building codes may have additional requirements not shown in this summary table.

APPENDIX B: AREA DISTRIBUTION ALGORITHM

The core Parametric Algorithm in Pseudocode format:

```
ALGORITHM: Proportional Area Distribution
INPUT:
  - target_total_area (user specified)
  - room_list (from template)
  - building_code_minimums (from selected country)

PROCESS:
1. Calculate minimum_required_area = SUM(all room minimums from building code)
2. Add circulation_space = minimum_required_area × 0.20
3. Total_minimum = minimum_required_area + circulation_space
```

```
4. IF target_total_area < total_minimum THEN
     RETURN ERROR "Area too small for building code compliance"

5. Calculate scale_factor = SQRT(target_total_area / total_minimum)

6. FOR each room IN room_list:
     new_width = room.min_width × scale_factor
     new_length = room.min_length × scale_factor

     // Maintain aspect ratios
     IF room.type == "bathroom" THEN
        new_length = new_width × 1.5
     ELSE IF room.type == "master_bedroom" THEN
        new_length = new_width × 1.2
     END IF

     // Ensure minimums are still met
     new_width = MAX(new_width, room.min_width)
     new_length = MAX(new_length, room.min_length)

     UPDATE global_parameters with new dimensions
   END FOR

OUTPUT: Adjusted room dimensions meeting target area
```

APPENDIX C: PyREVIT COMMAND DIALOGUES

C1: PROCESS ALL COMMANDS DIALOGUE DURING MASTER MODELLING

Process All Commands

Debug Info:
- Original TEMP: C:\Users\mursa\AppData\Local\Temp\e53c883c-6738-4f75-ac2a-a2bf549646bb
- Using temp path: C:\Users\mursa\AppData\Local\Temp

Looking for command queue at: C:\Users\mursa\AppData\Local\Temp\commands_queue.json
Found 4 commands to process
- **Command 1: select_template**

Parameters: { "template_name": "2BR_1BA_apartment"}
External UI Mode
Template: 2BR_1BA_apartment
Creating project: 2BR_1BA_apartment_20250915_115428.rvt
Success! Project created.
Project opened successfully!
✅ Command 'select_template' executed successfully
- **Command 2: set_area**

Parameters: { "total_area": 70}
✅ Area set to 70 sqm
- **Command 3: set_building_code**

Parameters: { "building_code": "slovenia_code"}
✅ Building code set to slovenia_code
📋 Ready to adjust room sizes with:
- Area: 70 sqm
- Building Code: slovenia_code
- Parameters: { "building_code": "slovenia_code", "auto_confirm": true, "total_area": 70}

```
Calling adjust_room_sizes button...
Debug: Wrote parameters to:
C:\Users\mursa\AppData\Local\Temp\button_parameters.json
External UI Mode
Building Code: slovenia_code
Total Area: 70 sqm
Detected room types in model: ['bathroom', 'master_bedroom', 'living_room',
'bedroom2', 'kitchen', 'bedroom']
Room minimums calculated: {'living_room': 13.0, 'kitchen': 4.5, 'master_bedroom':
10.0, 'bedroom2': 8.0, 'bathroom': 3.0, 'bedroom': 8.0}
Calculated Dimensions:
Calculated room dimensions for 70 m2 apartment:Building Code: slovenia_codeScale
Factor: 1.12
Master Bedroom: 3.9m x 3.3m = 12.7 m2Bedroom 2: 3.2m x 3.2m = 9.9 m2Bedroom: 3.2m
x 3.2m = 9.9 m2Living Room: 4.1m x 4.1m = 16.4 m2Kitchen: 2.5m x 2.3m = 5.6
m2Bathroom: 2.4m x 1.6m = 3.8 m2
Total Room Area: 58.4 m2Circulation Area: 11.6 m2 (17%)
Detected 6 rooms in your model.
✅ Command 'adjust_room_sizes' executed successfully
   • Command 4: lines_to_walls
Parameters: { "wall_height": 3.0, "wall_type": "Basic Wall"}
Debug: Wrote parameters to:
C:\Users\mursa\AppData\Local\Temp\button_parameters.json
Saved wall mapping to: C:\Users\mursa\AppData\Local\Temp\wall_label_mapping.json
Successfully created 19 walls from 19 lines.

Wall Type: Basic Wall : Wall-Ext_102Bwk-50Air-45Ins-100DBlk-12P
Wall Height: 3000mm

Wall Labels:
WL10 - Element ID: 336338
WL18 - Element ID: 336354
WL2 - Element ID: 336322
WL9 - Element ID: 336336
WL19 - Element ID: 336356
WL13 - Element ID: 336344
WL7 - Element ID: 336332
WL6 - Element ID: 336330
WL4 - Element ID: 336326
WL5 - Element ID: 336328
WL14 - Element ID: 336346
WL1 - Element ID: 336316
WL3 - Element ID: 336324
WL12 - Element ID: 336342
WL16 - Element ID: 336350
WL8 - Element ID: 336334
WL17 - Element ID: 336352
WL15 - Element ID: 336348
WL11 - Element ID: 336340
✅ Command 'lines_to_walls' executed successfully
   • Auto-executing: Room Generation
Automatically generating rooms after wall creation...
Debug: Room script path:
C:\MasterThesis\ParametricAIBIMModelling\pyRevit.extension\ParametricAIBIM.tab\0
9_Room Tools.panel\Room Generation.pushbutton\script.py
Debug: Script exists: True
   • Room Generation Tool
Detecting closed wall boundaries...
Found 5 closed boundaries
```

Creating rooms in boundaries...
✓ Created room 1
✓ Created room 2
✓ Created room 3
✓ Created room 4
✓ Created room 5
Analyzing global parameters to assign room names...
Detected room types from global parameters:
- Bedroom2: 3150mm x 3150mm (center-to-center)
- Bathroom: 2400mm x 1600mm (center-to-center)
- Kitchen: 2500mm x 2250mm (center-to-center)
- LivingRoom: 4050mm x 4050mm (center-to-center)
- MasterBedroom: 3900mm x 3250mm (center-to-center)

Analyzing room dimensions for parameter matching...
Room analysis:
- Wall thickness: 310mm
- Internal dimensions: 3590mm x 2940mm
- Bedroom2 match: 2840mm x 2840mm (diff: 850mm)
- Bathroom match: 2090mm x 1290mm (diff: 3150mm)
- Kitchen match: 2190mm x 1940mm (diff: 2400mm)
- LivingRoom match: 3740mm x 3740mm (diff: 950mm)
- MasterBedroom match: 3590mm x 2940mm (diff: 0mm)

Best match: MasterBedroom (score: 0mm)
Room analysis:
- Wall thickness: 310mm
- Internal dimensions: 2840mm x 2840mm
- Bedroom2 match: 2840mm x 2840mm (diff: 0mm)
- Bathroom match: 2090mm x 1290mm (diff: 2300mm)
- Kitchen match: 2190mm x 1940mm (diff: 1550mm)
- LivingRoom match: 3740mm x 3740mm (diff: 1800mm)
- MasterBedroom match: 3590mm x 2940mm (diff: 850mm)

Best match: Bedroom2 (score: 0mm)
Room analysis:
- Wall thickness: 310mm
- Internal dimensions: 2090mm x 1290mm
- Bedroom2 match: 2840mm x 2840mm (diff: 2300mm)
- Bathroom match: 2090mm x 1290mm (diff: 0mm)
- Kitchen match: 2190mm x 1940mm (diff: 750mm)
- LivingRoom match: 3740mm x 3740mm (diff: 4100mm)
- MasterBedroom match: 3590mm x 2940mm (diff: 3150mm)

Best match: Bathroom (score: 0mm)
Room analysis:
- Wall thickness: 310mm
- Internal dimensions: 2190mm x 1940mm
- Bedroom2 match: 2840mm x 2840mm (diff: 1550mm)
- Bathroom match: 2090mm x 1290mm (diff: 750mm)
- Kitchen match: 2190mm x 1940mm (diff: 0mm)
- LivingRoom match: 3740mm x 3740mm (diff: 3350mm)
- MasterBedroom match: 3590mm x 2940mm (diff: 2400mm)

Best match: Kitchen (score: 0mm)
Room analysis:
- Wall thickness: 310mm
- Internal dimensions: 4597mm x 3972mm
- Bedroom2 match: 2840mm x 2840mm (diff: 2889mm)
- Bathroom match: 2090mm x 1290mm (diff: 5189mm)
- Kitchen match: 2190mm x 1940mm (diff: 4439mm)

```
    • LivingRoom match: 3740mm x 3740mm (diff: 1089mm)
    • MasterBedroom match: 3590mm x 2940mm (diff: 2039mm)
No suitable match found
✓ Room 1 identified as MasterBedroom based on position
✓ Room 2 identified as Bedroom2 based on position
✓ Room 3 identified as Bathroom based on position
✓ Room 4 identified as Kitchen based on position
Room could not be identified
    • Summary
Successfully placed 5 rooms
Available room tag types:
    • 1: Name_and_Number
    • 2: Name_Number_w-Both_Areas
    • 3: Name_Only
    • 4: Name_Number_w-Metric_Area
Using tag type: Name_and_Number
Placing tags for 5 rooms...
    • Tagging room 'MasterBedroom' at (-6.93, 17.46)
✓ Tag placed successfully
    • Tagging room 'Bedroom2' at (-7.85, 5.89)
✓ Tag placed successfully
    • Tagging room 'Bathroom' at (-14.89, 19.21)
✓ Tag placed successfully
    • Tagging room 'Kitchen' at (-21.20, 16.56)
✓ Tag placed successfully
    • Tagging room 'Room' at (-19.29, 7.22)
✓ Tag placed successfully
✓ Placed 5 room tags
Note: If tags are not visible, check:
    • View scale (zoom in/out)
    • Tag visibility settings in Visibility/Graphics
    • Tag type properties (text size, etc.)
✅ Command 'room_generation' executed successfully
```

```
    • Summary
✅ Successful: 6
❌ Failed: 0
All commands processed!
```

C2: DOOR PLACEMENT DIALOGUE

```
Creating label 'D1' at circle center: (-12.26, 12.94, 0.00)
Successfully created label: D1
Creating label 'D2' at circle center: (-13.02, 8.02, 0.00)
Successfully created label: D2
Creating label 'D3' at circle center: (-17.51, 13.79, 0.00)
Successfully created label: D3
Creating label 'D4' at circle center: (-13.77, 15.27, 0.00)
Successfully created label: D4
Creating label 'D5' at circle center: (-15.32, -0.83, 0.00)
Successfully created label: D5
Saved door mapping to: C:\Users\mursa\AppData\Local\Temp\door_label_mapping.json
Successfully placed 5 doors out of 5 circles.
```

```
Door Type: Doors_IntSgl : 810x2110mm
Door Labels:
D2 - Element ID: 336375
D4 - Element ID: 336379
D1 - Element ID: 336373
D5 - Element ID: 336381
D3 - Element ID: 336377
```

C3: WINDOW PLACEMENT DIALOGUE

```
Creating label 'D1' at circle center: (-12.26, 12.94, 0.00)
Successfully created label: D1
Creating label 'D2' at circle center: (-13.02, 8.02, 0.00)
Successfully created label: D2
Creating label 'D3' at circle center: (-17.51, 13.79, 0.00)
Successfully created label: D3
Creating label 'D4' at circle center: (-13.77, 15.27, 0.00)
Successfully created label: D4
Creating label 'D5' at circle center: (-15.32, -0.83, 0.00)
Successfully created label: D5
Saved door mapping to: C:\Users\mursa\AppData\Local\Temp\door_label_mapping.json

Successfully placed 5 doors out of 5 circles.
Door Type: Doors_IntSgl : 810x2110mm
Door Labels:
D2 - Element ID: 336375
D4 - Element ID: 336379
D1 - Element ID: 336373
D5 - Element ID: 336381
D3 - Element ID: 336377
```

C4: CORRECTIONS COMMAND DIALOGUE

Process All Commands

Debug Info:

- Original TEMP: C:\Users\mursa\AppData\Local\Temp\e53c883c-6738-4f75-ac2a-a2bf549646bb
- Using temp path: C:\Users\mursa\AppData\Local\Temp

Looking for command queue at: C:\Users\mursa\AppData\Local\Temp\commands_queue.json

Found 12 commands to process

- **Command 1: cycle_doors**

Parameters: { "label": "D1", "action": "flip_sideways"}
Executing external command: {'label': 'D1', 'action': 'flip_sideways'}
Flipping door D1 sideways (hand)
✅ Command 'cycle_doors' executed successfully

- **Command 2: cycle_doors**

Parameters: { "label": "D3", "action": "flip_sideways"}
Executing external command: {'label': 'D3', 'action': 'flip_sideways'}
Flipping door D3 sideways (hand)
✅ Command 'cycle_doors' executed successfully

- **Command 3: cycle_doors**

Parameters: { "type_name": null, "label": "D4", "action": "change_family"}

```
Executing external command: {'type_name': None, 'label': 'D4', 'action': 'change_family'}
Changing family type for door D4
✅ Command 'cycle_doors' executed successfully
```

- **Command 4: cycle_doors**

```
Parameters: { "label": "D2", "action": "flip_sideways"}
Executing external command: {'label': 'D2', 'action': 'flip_sideways'}
Flipping door D2 sideways (hand)
✅ Command 'cycle_doors' executed successfully
```

- **Command 5: cycle_doors**

```
Parameters: { "type_name": null, "label": "D3", "action": "change_family"}
Executing external command: {'type_name': None, 'label': 'D3', 'action': 'change_family'}
Changing family type for door D3
✅ Command 'cycle_doors' executed successfully
```

- **Command 6: cycle_windows**

```
Parameters: { "type_name": null, "label": "W3", "action": "change_family"}
Executing external command: {'type_name': None, 'label': 'W3', 'action': 'change_family'}
Changing family type for window W3
✅ Command 'cycle_windows' executed successfully
```

- **Command 7: cycle_windows**

```
Parameters: { "label": "W5", "action": "flip_sideways"}
Executing external command: {'label': 'W5', 'action': 'flip_sideways'}
Flipping window W5 sideways (hand)
✅ Command 'cycle_windows' executed successfully
```

- **Command 8: cycle_walls**

```
Parameters: { "label": "WL6", "action": "flip"}
Executing external command: {'label': 'WL6', 'action': 'flip'}
Flipping wall WL6
✅ Command 'cycle_walls' executed successfully
```

- **Command 9: cycle_walls**

```
Parameters: { "label": "WL7", "action": "flip"}
Executing external command: {'label': 'WL7', 'action': 'flip'}
Flipping wall WL7
✅ Command 'cycle_walls' executed successfully
```

- **Command 10: cycle_walls**

```
Parameters: { "label": "WL18", "action": "flip"}
Executing external command: {'label': 'WL18', 'action': 'flip'}
Flipping wall WL18
✅ Command 'cycle_walls' executed successfully
```

- **Command 11: cycle_walls**

```
Parameters: { "label": "WL17", "action": "flip"}
Executing external command: {'label': 'WL17', 'action': 'flip'}
Flipping wall WL17
✅ Command 'cycle_walls' executed successfully
```

- **Command 12: cycle_walls**

```
Parameters: { "label": "WL9", "action": "flip"}
Executing external command: {'label': 'WL9', 'action': 'flip'}
Flipping wall WL9
✅ Command 'cycle_walls' executed successfully
```

- **Summary**

```
✅ Successful: 12
❌ Failed: 0
All commands processed!
```

APPENDIX D: CODE SNIPPETS

These code snippets demonstrate the core technical implementations that enable the system's functionality. The natural language processor shows the pattern-matching approach, the building code validator illustrates the compliance enforcement mechanism, and the command orchestrator reveals how multiple operations are coordinated to generate complete BIM models from simple text commands.

D1: NATURAL LANGUAGE PROCESSING ENGINE

PATTERN-BASED COMMAND PARSER:

```
class NaturalLanguageProcessor:
    def __init__(self):
        self.patterns = {
            'bedroom_count': [
                r'(\d+)\s*(?:bed|bedroom|br)',
                r'(?:studio|efficiency)',
                r'(\w+)[\s-]bedroom'
            ],
            'area': [
                r'(\d+(?:\.\d+)?)\s*(?:square\s*)?(?:meter|metre|m2|sqm)',
                r'(\d+(?:\.\d+)?)\s*(?:square\s*)?(?:feet|foot|sqft|sf)'
            ],
            'building_code': [
                r'(?:use\s+)?(\w+)\s+(?:building\s+)?code',
                r'(?:follow|apply)\s+(\w+)\s+(?:regulations?|standards?)'
            ]
        }

    def parse_command(self, text):
        """Extract structured parameters from natural language input"""
        command = {}
        text_lower = text.lower()

        # Extract bedroom count
        for pattern in self.patterns['bedroom_count']:
            match = re.search(pattern, text_lower)
            if match:
                if 'studio' in match.group(0):
                    command['bedrooms'] = 0
                else:
                    num = self._word_to_number(match.group(1))
                    command['bedrooms'] = num
                break

        # Extract area with unit conversion
        for pattern in self.patterns['area']:
            match = re.search(pattern, text_lower)
            if match:
                value = float(match.group(1))
                if 'feet' in match.group(0) or 'sqft' in match.group(0):
                    value = value * 0.092903  # Convert to square meters
                command['total_area'] = round(value)
                break

        return command
```

```
    def _word_to_number(self, word):
        """Convert word numbers to integers"""
        word_map = {
            'one': 1, 'two': 2, 'three': 3, 'four': 4,
            'single': 1, 'double': 2, 'triple': 3
        }
        return word_map.get(word, int(word) if word.isdigit() else None)
```

ELEMENT REFERENCE RESOLUTION:

```
class ElementResolver:
    def resolve_element_reference(self, reference, element_mappings):
        """Resolve natural language element references to IDs"""

        # Direct label reference (e.g., "door D3")
        label_match = re.search(r'([DWL]\d+)', reference.upper())
        if label_match:
            label = label_match.group(1)
            return self._find_by_label(label, element_mappings)

        # Functional reference (e.g., "main entrance door")
        if 'entrance' in reference and 'door' in reference:
            return self._find_entrance_door(element_mappings)

        # Spatial reference (e.g., "master bedroom window")
        room_match = re.search(r'(\w+\s*bedroom|bathroom|kitchen)', reference)
        if room_match:
            room_type = room_match.group(1)
            element_type = self._extract_element_type(reference)
            return self._find_in_room(room_type, element_type, element_mappings)

        return None

    def _find_entrance_door(self, mappings):
        """Identify main entrance based on connectivity analysis"""
        door_scores = {}
        for label, data in mappings.get('doors', {}).items():
            score = 0
            # Higher score for exterior walls
            if data.get('is_exterior', False):
                score += 10
            # Higher score for larger doors
            if data.get('width', 0) > 900:
                score += 5
            door_scores[label] = score

        if door_scores:
            return max(door_scores, key=door_scores.get)
        return None
```

D2: BUILDING CODE COMPLIANCE SYSTEM

MULTI-COUNTRY CODE VALIDATOR:

```
class BuildingCodeValidator:
    def __init__(self):
        self.codes_path                                                    =
r"C:\MasterThesis\ParametricAIBIMModelling\data\building_codes"
        self.codes = self._load_all_codes()

    def _load_all_codes(self):
        """Load all building code JSON files"""
        codes = {}
        for file in os.listdir(self.codes_path):
            if file.endswith('_code.json'):
                country = file.replace('_code.json', '')
                with open(os.path.join(self.codes_path, file), 'r') as f:
                    codes[country] = json.load(f)
        return codes

    def validate_and_adjust(self, room_params, country_code, total_area):
        """Validate room parameters and adjust for compliance"""
        if country_code not in self.codes:
            raise ValueError(f"Unknown building code: {country_code}")

        code_reqs = self.codes[country_code]
        adjusted_params = {}

        # Calculate minimum required area
        min_total = 0
        for room_type, params in room_params.items():
            room_min = self._get_room_minimum(room_type, code_reqs)
            min_total += room_min['area']

        # Add circulation space (20%)
        min_total_with_circulation = min_total * 1.2

        if total_area < min_total_with_circulation:
            raise ValueError(
                f"Area {total_area}m² too small. "
                f"Minimum required: {min_total_with_circulation:.1f}m²"
            )

        # Calculate scale factor for proportional adjustment
        scale_factor = math.sqrt(total_area / min_total_with_circulation)

        # Adjust each room maintaining proportions
        for room_type, params in room_params.items():
            room_min = self._get_room_minimum(room_type, code_reqs)
            adjusted_params[room_type] = {
                'width':       max(room_min['width']       *       scale_factor,
room_min['width']),
                'length':      max(room_min['length']      *      scale_factor,
room_min['length'])
            }

        return adjusted_params
```

BUILDING CODE DATA STRUCTURE:

```
{
    "country": "Slovenia",
    "code_reference": "Uradni list RS",
    "room_requirements": {
        "master_bedroom": {
            "min_area": 10.0,
            "min_width": 2.7,
            "min_length": 3.0,
            "ceiling_height": 2.5
        },
        "bedroom": {
            "min_area": 8.0,
            "min_width": 2.4,
            "min_length": 2.7
        },
        "living_room": {
            "min_area": 13.0,
            "min_width": 3.0,
            "natural_light_ratio": 0.1
        },
        "bathroom": {
            "min_area": 3.0,
            "min_width": 1.5,
            "ventilation": "required"
        }
    },
    "accessibility": {
        "door_min_width": 0.8,
        "corridor_min_width": 1.2,
        "turning_circle_diameter": 1.5
    }
}
```

D3: PYREVIT PLUGIN COMMAND ORCHESTRATION

MASTER COMMAND PROCESSOR

```
class CommandOrchestrator:
    def __init__(self):
        self.button_scripts = {
            'select_template': 'Select Template.pushbutton',
            'adjust_room_sizes': 'Adjust Room Sizes.pushbutton',
            'lines_to_walls': 'Lines to Wall.pushbutton',
            'place_doors': 'Place Doors.pushbutton',
            'place_windows': 'Place Windows.pushbutton'
        }
        self.state = CommandState()

    def process_command_queue(self, commands):
        """Process all commands with intelligent sequencing"""
        results = []

        for cmd in commands:
            try:
                # Accumulate state for multi-part commands
                if cmd['type'] in ['set_area', 'set_building_code']:
```

```
                    self.state.update(cmd['parameters'])

                    # Check if we have enough info to proceed
                    if self.state.has_required_params():
                        self._execute_room_adjustment()

                # Direct execution commands
                elif cmd['type'] == 'select_template':
                    result = self._execute_template_selection(cmd['parameters'])
                    results.append(result)

                # Element modification commands
                elif cmd['type'].startswith('modify_'):
                    result = self._execute_element_modification(cmd)
                    results.append(result)

                # Cascading operations
                elif cmd['type'] == 'lines_to_walls':
                    wall_result     =     self._execute_script('lines_to_walls',
cmd['parameters'])
                    results.append(wall_result)

                    # Automatically trigger room generation
                    room_result = self._execute_script('room_generation', {})
                    results.append(room_result)

            except Exception as e:
                results.append({
                    'command': cmd['type'],
                    'status': 'error',
                    'message': str(e)
                })

        return results

    def _execute_script(self, script_name, parameters):
        """Dynamically import and execute button scripts"""
        # Write parameters for script to read
        param_file = os.path.join(temp_dir, 'button_parameters.json')
        with open(param_file, 'w') as f:
            json.dump(parameters, f)

        # Import and execute the script
        script_path = self._get_script_path(script_name)
        spec = importlib.util.spec_from_file_location(script_name, script_path)
        module = importlib.util.module_from_spec(spec)
        spec.loader.exec_module(module)

        # Read results if available
        result_file = os.path.join(temp_dir, f'{script_name}_result.json')
        if os.path.exists(result_file):
            with open(result_file, 'r') as f:
                return json.load(f)
        return {'status': 'completed'}
```

ELEMENT MODIFICATION HANDLER

```
def execute_element_modification(self, element_label, action, parameters=None):
    """Direct element manipulation bypassing UI"""

    # Load element mappings
    mapping_files = {
        'D': 'door_label_mapping.json',
        'W': 'window_label_mapping.json',
        'WL': 'wall_label_mapping.json'
    }

    element_type = element_label[0] if element_label[0] != 'W' else
element_label[:2]
    mapping_file = os.path.join(temp_dir, mapping_files.get(element_type))

    with open(mapping_file, 'r') as f:
        mappings = json.load(f)

    if element_label not in mappings:
        raise ValueError(f"Element {element_label} not found")

    element_id = ElementId(int(mappings[element_label]['id']))
    element = doc.GetElement(element_id)

    with Transaction(doc, f"Modify {element_label}") as trans:
        trans.Start()

        if action == 'flip':
            if hasattr(element, 'Flip'):
                element.Flip()
            elif hasattr(element, 'FlipHand'):
                element.FlipHand()

        elif action == 'change_type' and parameters:
            new_type = self._find_type_by_name(parameters.get('type_name'))
            if new_type:
                element.ChangeTypeId(new_type.Id)

        elif action == 'delete':
            doc.Delete(element_id)

        trans.Commit()

    return {'element': element_label, 'action': action, 'status': 'success'}
```

APPENDIX E: MODEL GENERATIONS

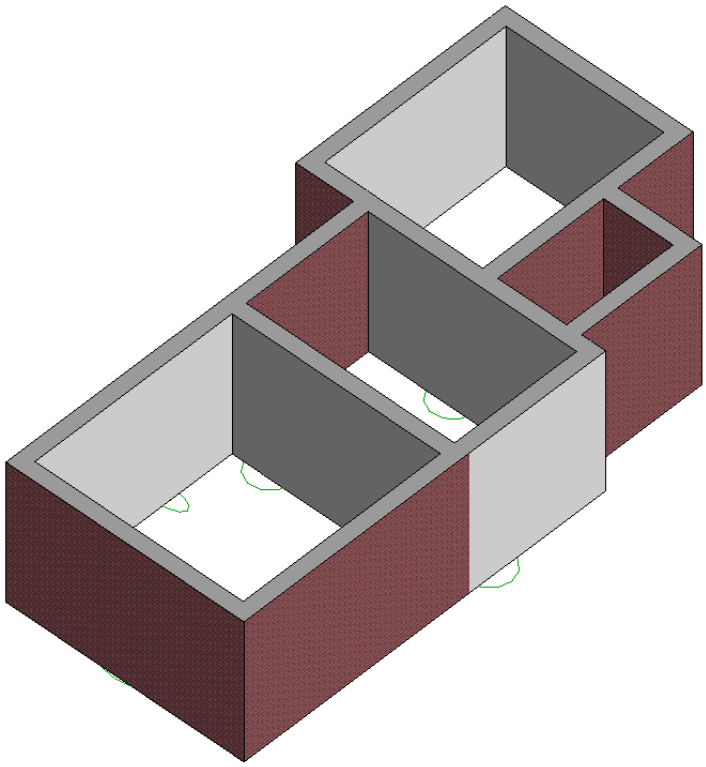

Figure 53: One Bedroom Model, Generated using External UI

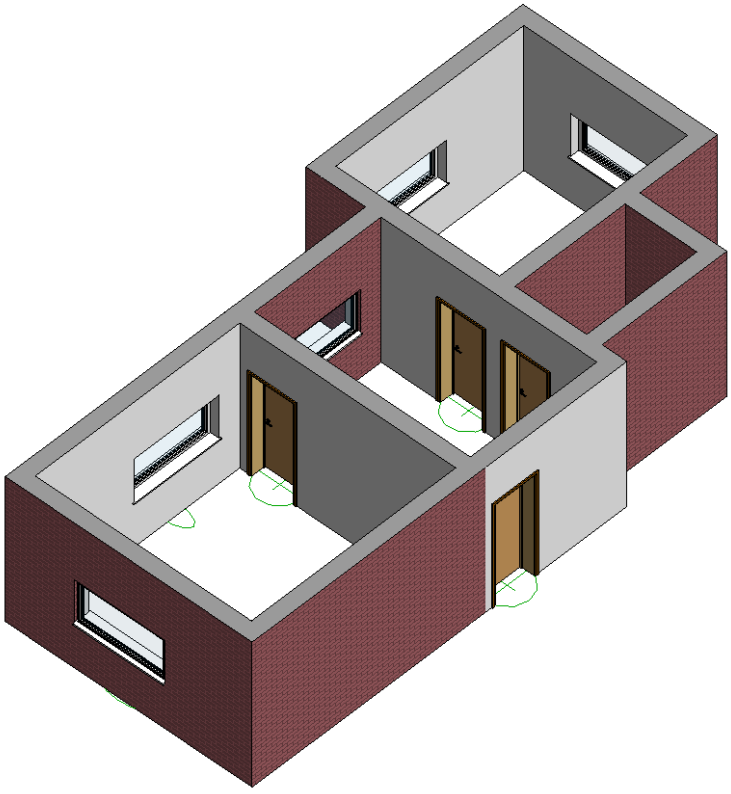

Figure 54: One Bedroom Model, Placed Doors and Windows

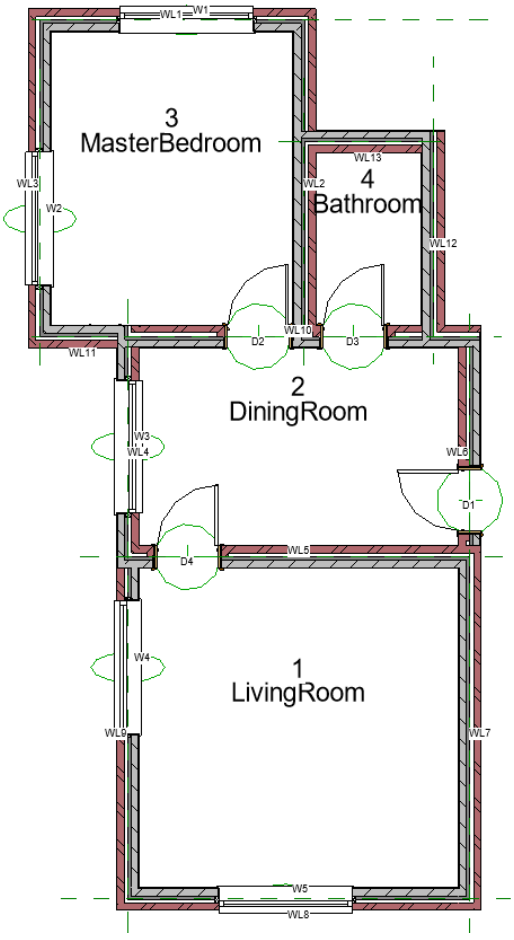

Figure 55: One Bedroom Model, Before Corrections

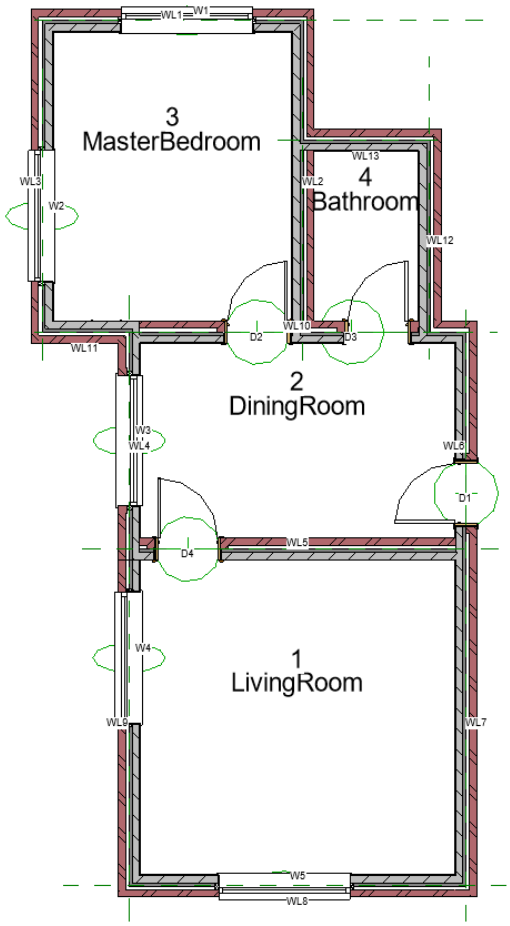

Figure 56: One Bedroom Mode, After Correction

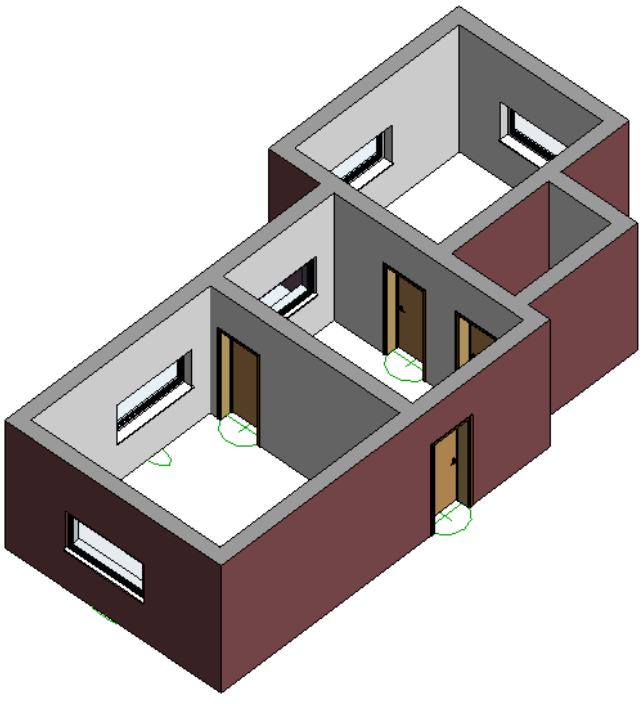

Figure 57: One Bedroom Mode, After Correction

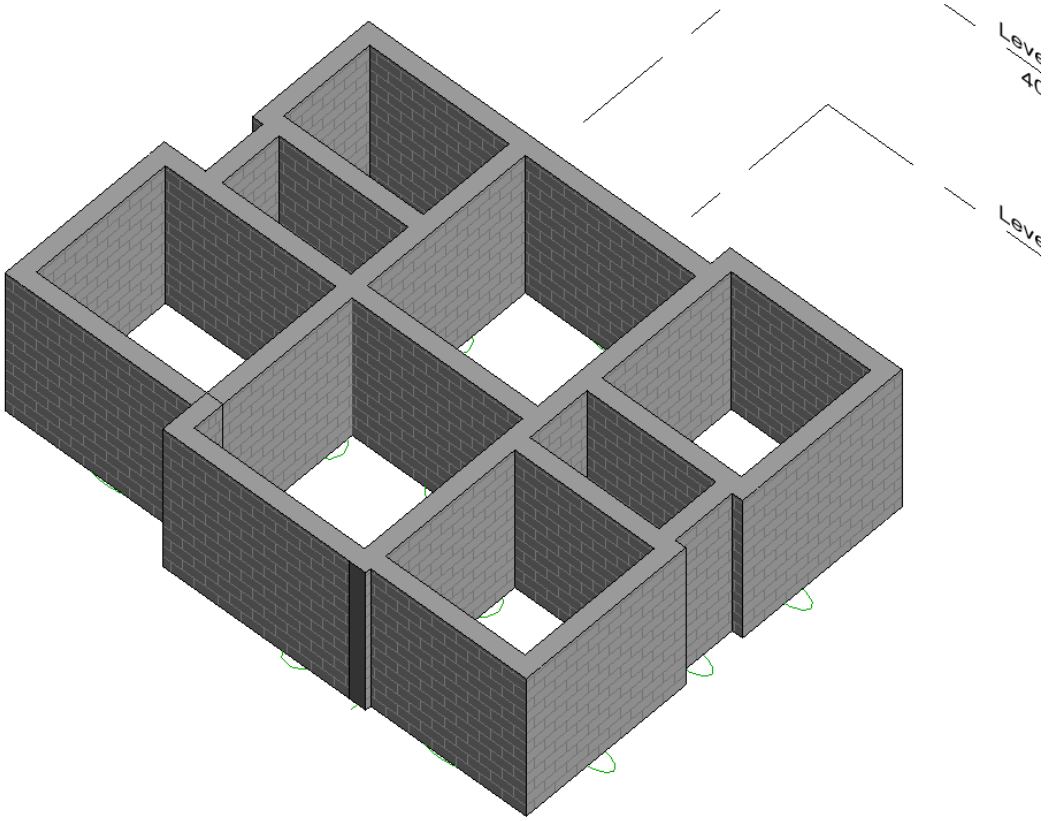

Figure 58: Four Bedroom Model, Generated using External UI

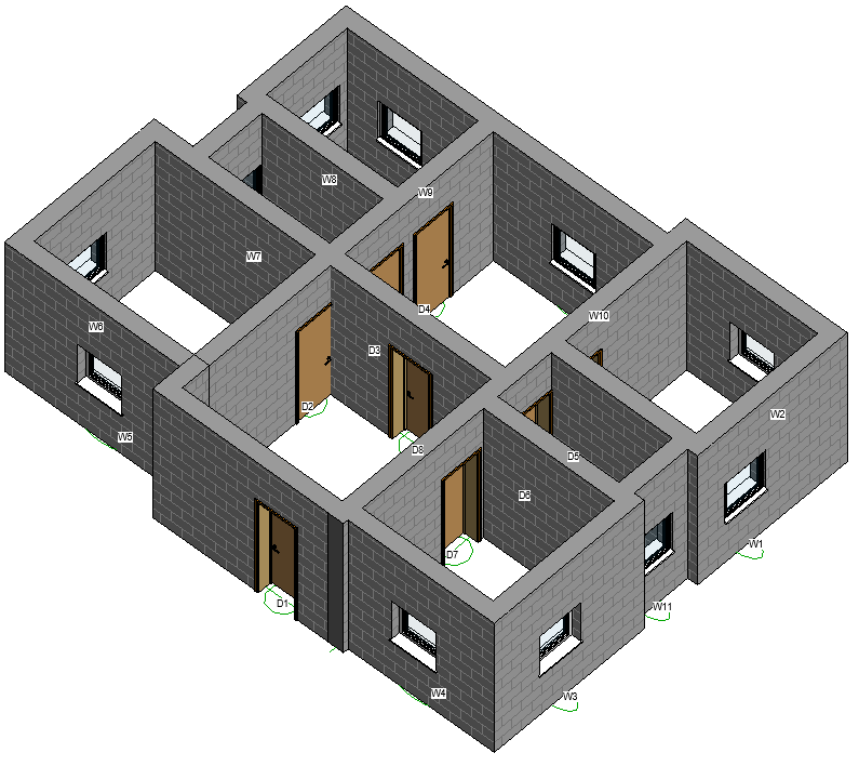

Figure 59: Four Bedroom Model, Placed Doors and Windows

Figure 60: Four Bedroom Model, After Corrections