

GSOC 26: Integrate a client for building a Smart Data Dictionary web service ([issue](#))

Name: Parag Debnath

Student's email: paragforwork8@gmail.com

Phone : +91 8415811654

GitHub: <https://github.com/paragforwork>

Introduction and motivation

I'm currently a pre-final year student at National Institute of Technology Agartala, India. I am one of the lead members of the Coding society of our college. I have taken part as well as won various hackathons on different levels.

My first experience with open-source was back in 2023, when I first discovered GitHub; since then, I've developed a deep passion and respect for open source software and the value it brings to the world. I have a hands on experience on working with python and c++ codebases.

Brief summary

The **buildingSMART Data Dictionary** (bSDD) is a centralized, cloud-based web service managed by [buildingSMART](#) International. It serves as an encyclopedia of standardized, hierarchical concepts and properties used in Building Information Modeling (BIM).

Different disciplines, countries, and software platforms often use different terminology to describe the same physical object. An architect in the US might call it a "Door," a British engineer might call it a "Doorset," a Spanish contractor calls it a "Puerta".

The bSDD solves this semantic interoperability problem by separating the *name* of a concept from its *meaning*. It assigns a machine-readable Global Unique Identifier (GUID) and a specific web URL (URI) to every concept. By linking FreeCAD objects to these centralized URIs, the exported Industry Foundation Classes (IFC) data becomes universally understandable by any other openBIM software.

Right now, assigning a standardized classification to a 3D object in FreeCAD is a highly manual, multi-step process that lacks context awareness. The user types a search term (e.g., "wall" or "concrete") into the search bar of the active classification system. The tool filters the locally loaded list. The user must manually read through the resulting tree or list of hierarchical classifications, relying entirely on their own domain knowledge to decide which specific code. The Classification Manager does not care what the user has selected. Even if the user selects an "IFC Door", they still have to manually type "door" into the search bar.

Detailed Explanation And Proposal

Project Objective The primary goal of this project is to build and integrate a Python-based client for the buildingSMART Data Dictionary (bSDD) directly into FreeCAD's BIM Workbench. This integration will significantly streamline the workflow for architects and engineers by providing context-aware, automated classification suggestions for BIM objects (such as walls, slabs, and materials) based on international IFC standards.

Detailed Research of the Bonsai BIM

Since [Bonsai BIM](#) has this feature implemented. I did some research there to find out how exactly they have implemented this feature. Moreover I also have some contributions in the [IfcOpenShell](#) so I am quite familiar with the code base.

The **Bonsai BIM** implementation executes the classification process through a logical sequence of API calls and UI updates:

1. **Initialize & Fetch Domains:** The tool hits the bSDD dictionary endpoint ([GET /api/Dictionary/v1](#)) to retrieve a list of available classification systems (e.g., Uniclass, OmniClass) along with metadata like status, version, and global URI.
2. **Active Context:-** The user selects which dictionaries are "active." Crucially, these active dictionary choices are persisted within the project's IFC metadata, ensuring the model carries its classification context universally.
3. **Context-Aware Querying:** When the user searches for a class, the tool builds requests filtered not only by the active dictionary but optionally by the selected object's underlying IFC entity (e.g., filtering only classes applicable to [IfcWall](#)).
4. **Fetch Class Details:** Upon selecting a specific classification from the search results, the tool fetches the full payload from the bSDD, which includes the required class properties, allowed predefined values (enums), and related property sets.

A critical takeaway from Bonsai BIM is that applying a classification is not merely slapping a string label onto an object; it requires a strict IFC hierarchy:

Anchor Creation: Before a class can be assigned, the system ensures a parent [IfcClassification](#) entry exists at the project level (matching the bSDD dictionary URI and version).

Applying References: The tool creates an [IfcClassificationReference](#). It maps the bSDD reference code to the IFC *Identification* field, the class name to the *Name* field, and the bSDD URI to the *Location* field.

API & Performance Strategy used by the Bonsai BIM —

REST over GraphQL: Despite bSDD offering GraphQL, Bonsai BIM relies on the REST API.

Auth: While OAuth is supported, standard dictionary and class lookups are executed without active authentication flows to reduce friction.

Caching Strategy: The implementation avoids heavy, long-term database storage. Instead, it utilizes an **in-memory session cache**. As the user works, class details and property details are cached locally by their URIs. This request-on-demand approach minimizes latency during repeated interactions, reduces API load, and keeps the UI highly responsive without bloating the local storage.

Current FreeCAD BIM classification implementation

Currently, a user must manually manage classifications:

1. The user opens the Classification Manager from the BIM tools.
2. The dialog loads static classification systems (e.g., Uniclass) strictly from locally stored files.
3. The user manually searches the tree view, selects a class, and clicks "Apply."
4. FreeCAD writes the combined system and code into either the `StandardCode` or `Classification` property.
5. Data Storage (Legacy): The actual classification data is persisted as a single string property named `StandardCode` within the `ArchComponent`, `ArchMaterial`, and `ArchEquipment` base classes.
6. IFC Export Mapping (Legacy): Upon export, `exportIFC.py` reads the `StandardCode` string, dynamically splits it, and generates the necessary `IfcClassification`, `IfcClassificationReference`, and `IfcRelAssociatesClassification` entities.
7. For IFC-native documents, the system bypasses `StandardCode`. Instead, `ifc_classification.py` uses `IfcOpenShell` API calls to synchronize classification strings directly with the underlying IFC entities in real time whenever an object is edited (`ifc_objects.py`).d

While the current system successfully generates the core IFC entities (`IfcClassificationReference`, etc.) upon export, it relies on several architectural simplifications that will need to be addressed to support the bSDD:-

No Unified Data Model: The split between `StandardCode` (Legacy) and `Classification` (NativeIFC) creates two separate pipelines. More importantly, both pipelines reduce complex classification data into a **single text string**.

Loss of Structured Metadata: Because the system only stores a combined text payload, it drops critical metadata such as the system URI, version, edition, and reference URI.

Fragile Parsing: The legacy exporter relies on simple string splitting (e.g., `classification.split(" ", 1)`) to separate the system name from the code. This is brittle and prone to failure if the user alters the text.

No Multi-Class Support: The current UI and property models effectively restrict an object to a single classification string, whereas the IFC standard fully supports associating an object with multiple classification systems simultaneously.

Detailed Proposal

1. Asynchronous bSDD Client and Caching

I will implement a non-blocking, Qt-based API client to ensure that HTTP requests never freeze the main FreeCAD thread. To achieve this, I will develop a new `BsddClient` class that inherits from `QObject`, which I will integrate directly into `BimClassification.py` (or a dedicated adjacent module to keep the codebase clean).

For the networking layer, I will utilize PySide's `QNetworkAccessManager` to handle asynchronous HTTP GET requests. I will carefully connect the `reply.finished` signals to my dedicated parser slots to handle the JSON responses. I will program my client to support three primary endpoints:

- `/api/Dictionary/v1` to fetch available classification domains.
- `/api/Search/v1` to execute context-aware searches for concepts.
- `/api/Concept/v1` to retrieve deep payloads, including properties, relations, and enums.

To minimize network latency and reduce the load on the bSDD API, I will build an efficient in-memory session caching system. I will structure this cache into three distinct tiers: a `dict_cache` indexed by the Dictionary URI, a `search_cache` indexed by a tuple of `(query, active_dicts, ifc_class)`, and a `concept_cache` indexed by the Concept URI.

2. UI Extension & Context Persistence

I will significantly extend the current user interface by introducing new Qt widgets specifically designed for bSDD operations. I will build and integrate a provider toggle, an active dictionary checklist, an asynchronous search box, a hierarchical results tree, and a property preview table.

To ensure the UI feels highly responsive, I will implement event handling with search debouncing. By utilizing a `QTimer` set to 250–400ms, I will throttle API requests while the user is actively typing.

Furthermore, I will implement robust state persistence logic. I will write functions to save the user's active dictionary selections directly into the `IfcProject` metadata. I will design this so that whenever a user reopens their model, their active dictionary context is automatically detected and restored.

3. Strict Data Contract & IFC Mapping

I will establish a strict data contract to pass structured information seamlessly from my new UI layer down to the underlying IFC mapping layer. Whenever a user selects a bSDD concept, I will configure the UI to construct a standardized payload containing:

- The **dictionary** metadata (name, uri, version).
- The **class** metadata (code, name, uri).
- The **ifc_filter** (the applicable IFC entity).
- The **properties** (a parsed list of required properties, data types, and enum constraints).

I will intentionally route this payload directly to new functions I created in the NativeIFC module. By doing this, I will successfully bypass the legacy **StandardCode** single-string property, preserving the rich metadata provided by the bSDD.

I will execute the core data modeling changes inside **ifc_classification.py**. Here, I will introduce new semantic writer functions engineered to strictly enforce the correct IFC schema hierarchy for all incoming bSDD data.

4. Validation & Backward Compatibility

In the final phase of my project, I will focus entirely on testing and quality assurance to ensure my new pipeline does not disrupt any existing architectural workflows.

I am committed to legacy preservation: I will leave the existing **StandardCode** and offline classification workflows fully intact so that users relying on traditional, non-bSDD processes experience absolutely no regressions.

Finally, I will execute a strict testing protocol. I will personally verify that my code correctly generates and links the required IFC entities, specifically writing checks for **IfcClassification**, **IfcClassificationReference**, and **IfcRelAssociatesClassification** to guarantee total IFC compliance.

Project Timelines

Period	Weeks	Task	Description
April May 1 – May 24	Community Bonding Period	Architecture Finalization & UI Mockups	I will collaborate with my mentors to finalize the system architecture, design the graphical interface mockups for the updated classification dialog, and establish the exact data payload structures expected from the external dictionary service.
May 25 – June 7	Weeks 1 & 2	Async API Client & Caching System	I will develop a non-blocking network client to securely communicate with the online dictionary's endpoints. To optimize performance and reduce API load, I will implement a multi-layered, in-memory caching system to temporarily store dictionary domains, search queries, and concept details during an active session.

June 8- June 21	Week 3 & 4	UI Extension & Async Event Handling	I will expand the existing classification interface by adding new interactive elements, including provider toggles, live search bars, and hierarchical result trees. I will implement input throttling (debouncing) to ensure the main application remains fully responsive while fetching live web data.
June 22-June 28	Week 5	Data Contract & Context Persistence	I will establish UI-to-NativeIFC structured data transfer and implement persistence of dictionary selections in IfcProject metadata.
June 29- July 5	Week 6	Core IFC Semantic Mapping Engine	I will build the underlying semantic engine responsible for translating the downloaded dictionary concepts into strictly compliant IFC reference entities. This includes logic to correctly anchor the chosen classification systems securely to the project document
July 6 -July 12	Week 7(Midterm)	Integration & Midterm Submission	I will connect the frontend interface directly to the semantic engine, enabling a complete end-to-end workflow: from searching a concept online to successfully linking it to a 3D object in the viewport. I will submit my Midterm Evaluation demonstrating this core functionality
July 13- July 26	Week 8 & 9	Automated Property Set Generation	I will implement the logic for dynamic property generation. When a user applies a classification, my code will parse the required attributes, correctly map the external data types to native application types, enforce restricted value lists, and automatically attach the corresponding standard property sets to the physical object.
July 27-Aug 2	Week 10	Legacy Compatibility & Fallback Handling	I will implement offline-safe error handling and ensure backward compatibility with StandardCode and XML/IFC classification workflows.
Aug 3- Aug 9	Week 11	Testing & Bug Fixing	I will perform regression and IFC round-trip testing to ensure persistence of bSDD URIs, references, and Property Sets.
Aug 10 - Aug 16	Week 12	Documentation & Buffer	I will finalize documentation, write a FreeCAD Wiki user guide, clean up code, and address unforeseen issues.
Aug 17- Aug 24	Week 13	Final Evaluation & Submission	I will submit the final squashed PR with complete bSDD integration and complete GSoC evaluation requirements.

Past Contributions

To get familiar with the FreeCAD, I made the following contributions and bugfixes:

These are the list of my merged PRs:-

1. <https://github.com/FreeCAD/FreeCAD/pull/28123>
2. <https://github.com/FreeCAD/FreeCAD/pull/28036>
3. <https://github.com/FreeCAD/FreeCAD/pull/27966>
4. <https://github.com/FreeCAD/FreeCAD/pull/27844>
5. <https://github.com/FreeCAD/FreeCAD/pull/27785>
6. <https://github.com/FreeCAD/FreeCAD/pull/27748>

These are the list of my PRs that are under considerations but contains some approved labels:-

1. <https://github.com/FreeCAD/FreeCAD/pull/27951>
2. <https://github.com/FreeCAD/FreeCAD/pull/28114>
3. <https://github.com/FreeCAD/FreeCAD/pull/28216>

Some of my other contributions that are under review :-

1. <https://github.com/FreeCAD/FreeCAD/pull/28369>
2. <https://github.com/FreeCAD/FreeCAD/pull/28319>
3. <https://github.com/FreeCAD/FreeCAD/pull/28299>
4. <https://github.com/FreeCAD/FreeCAD/pull/28147>
5. <https://github.com/FreeCAD/FreeCAD/pull/28090>
6. <https://github.com/FreeCAD/FreeCAD/pull/28021>
7. <https://github.com/FreeCAD/FreeCAD/pull/27994>
8. <https://github.com/FreeCAD/FreeCAD/pull/27985>
9. <https://github.com/FreeCAD/FreeCAD/pull/27584>

Currently I am brainstorming to improve this [PR](#) that solves this [issue](#).

Time Availability

During the summer, I have no other commitments and will be available to work full-time on the project - for at least 35-40 hours per week.

From May 8 to May 16 (part of Community Bonding Period), I'll be having my semester End-term examinations.

I will be fully available to work from May 17 onwards.

Beginning from July 25, university classes will commence, however I will allocate sufficient time to work atleast 35 hours a week.

