## ***SON Impact Analysis Node Restructure of Small Cells (Based on PPTS)***

## Overview

This document outlines the changes in the SON (Self-Organizing Network) impact analysis related to the node restructure of small cells within the T-Mobile network. It highlights the modifications in node locations, equipment representation, and cell naming conventions.

## Key Changes

1. Node Locations
   * Nodes are now created as distinct rings and sites, similar to other site types in the T-Mobile network.
   * BBU (Baseband Unit) equipment will no longer be represented as distinct ring and site records; instead, they are created under the hub site in its Cabinet Equipment table.
   * Sectors/cells will be positioned at their physical locations (node site).
2. Sector/Cell Representation
   * Current Naming Convention:Example: SITE ID – LA8035BA with cells represented by different sectors (1 to 3).
   * Proposed Naming Convention:SITE ID – LA8035BA will change for each node, with cells represented by the same sector.
3. Cell Naming Convention Details
   * The new naming convention will include attributes like sector ID, carrier count, technology, spectrum, and coverage type.
   * Examples of cell names:SE01001C\_A0GPA – GSM sector
4. Importance of Cell Naming Convention
   * Ensures data integrity and accurate classification of technology and coverage types from the start of development.
   * Simplifies reporting on deployed technology and mitigates risks associated with inaccurate classifications.
5. Coverage Type Identifiers
   * Coverage types are categorized into outdoor, indoor, and mixed/special.
   * Each coverage type has specific identifiers (e.g., A for Macro, Z for Micro).

## Conclusion

The restructuring of small cells within the SON framework is aimed at enhancing operational efficiency and accuracy in network management. The new naming conventions and representation strategies will facilitate better data integrity and reporting capabilities.

# T Document: CmManager Class

### 1. Overview

The CmManager class is designed to handle data retrieval for Configuration Management (CM) using SON SDK APIs. This class provides methods to retrieve various types of attributes (including child attributes) related to DNs (Distinguished Names), using both single-threaded and multi-threaded operations. It also supports chunking DN lists to optimize performance when dealing with large sets of DNs.

### 2. Purpose

The CmManager class facilitates interaction with the EMS (Element Management System) for retrieving attributes and child attributes of Configuration Management data. It provides an efficient way to process large DN lists by utilizing threading, ensuring the operation is optimized for both small and large datasets.

### 3. Dependencies

* enet\_ems.ems. EMSManager: This class is used to interact with the EMS to fetch attributes and child attributes.
* ThreadPoolExecutor: Used for executing functions in multiple threads for faster processing when enable\_multithread is set to True.
* tmo.chunk: This utility function is used for chunking the DN list for processing in batches, ensuring large sets are handled efficiently.

### 4. Key Attributes

* script\_data (object): Initialization data required for interacting with the EMS.
* ems (EMSManager): An instance of the EMSManager class used for managing operations in the EMS.
* enable\_multithread (bool): Flag to enable or disable multi-threading. The default is True, enabling multi-threading.
* dn\_chunk\_size (int): Defines the maximum size of DN chunks used for processing (default is 300).

### 5. Key Methods

#### 5.1 \_\_init\_\_(self, script\_data, enable\_multithread=True)

Purpose:  
Initializes an instance of the CmManager class. It sets up the script data, the EMS manager, and the multi-threading flag.

Parameters:

* script\_data (object): Data required for initialization, usually containing information such as region and other configurations.
* enable\_multithread (bool): A flag to enable multi-threading (default is True).

Example Usage:

cm\_manager = CmManager(script\_data)

#### 5.2 get\_attributes(self, dn\_list, parameter\_list, oss\_value=True)

Purpose:  
Retrieves attributes for a list of DNs (Distinguished Names).

Parameters:

* dn\_list (list): A list of DNs to retrieve attributes for.
* parameter\_list (list): A list of parameters (attributes) to fetch for each DN.
* oss\_value (bool): Whether to retrieve the value from OSS (default is True).

Returns:  
A dictionary where the keys are DNs and the values are the corresponding attributes.

Example Usage:

attributes = cm\_manager.get\_attributes(dn\_list, parameter\_list)

#### 5.3 get\_child\_attributes(self, dn\_list, mo\_list, param\_filter=None, value\_filter=None, oss\_value=True, return\_by\_dn=False, dn\_only=False)

Purpose:  
Retrieves child attributes for a list of DNs.

Parameters:

* dn\_list (list): A list of parent DNs to retrieve child attributes for.
* mo\_list (list): A list of Managed Object (MO) types to filter by.
* param\_filter (dict, optional): A filter to specify which attributes to fetch for each MO.
* value\_filter (dict, optional): A filter to specify attribute values to filter by.
* oss\_value (bool): Whether to retrieve the value from OSS (default is True).
* return\_by\_dn (bool): Whether to return the results grouped by DN (default is False).
* dn\_only (bool): If set to True, only child DNs will be returned.

Returns:  
A dictionary of child attributes for the given DNs.

Example Usage:

child\_attributes = cm\_manager.get\_child\_attributes(dn\_list, mo\_list)

#### 5.4 get\_child\_mo\_dns(self, dn\_list, mo\_list)

Purpose:  
Retrieves child DNs for a list of parent DNs.

Parameters:

* dn\_list (list): A list of parent DNs to retrieve child DNs for.
* mo\_list (list): A list of Managed Object types to filter by.

Returns:  
A list of child DNs.

Example Usage:

child\_dns = cm\_manager.get\_child\_mo\_dns(dn\_list, mo\_list)

#### 5.5 \_\_run\_in\_threads(self, func, dn\_list, \*\*kwargs)

Purpose:  
Executes a function in multiple threads to process a list of DNs in parallel.

Parameters:

* func (callable): The function to run in threads.
* dn\_list (list): The list of DNs to process.
* \*\*kwargs: Additional parameters to pass to the function.

Returns:  
A dictionary of results from the function executed in threads.

### 6. Example Usage

# Initialize CmManager

cm\_manager = CmManager(script\_data)

# Get attributes for a list of DNs

attributes = cm\_manager.get\_attributes(dn\_list, parameter\_list)

# Get child attributes for a list of DNs and filter by value

child\_attributes = cm\_manager.get\_child\_attributes(dn\_list, mo\_list, value\_filter={'keyId': 'CXC4012691'})

# Retrieve child DNs for parent DNs

child\_dns = cm\_manager.get\_child\_mo\_dns(dn\_list, mo\_list)

### 7. Multi-threading

By default, the class supports multi-threading to speed up the processing of large DN lists. You can disable multi-threading by setting enable\_multithread = False during initialization.

cm\_manager = CmManager(script\_data, enable\_multithread=False)

### 8. Error Handling

The class includes error handling, especially when calling external functions like get\_child\_attributes. If an exception is raised during processing, it is logged and the method gracefully returns an empty dictionary.

### 9. Performance Optimization

* DN Chunking: The DN list is chunked into smaller batches, typically of 300 DNs (can be adjusted if necessary), which helps in managing the performance when dealing with large sets of DNs.
* Multi-threading: If multi-threading is enabled, the tasks are split into smaller chunks, and each chunk is processed in parallel to improve performance.

### 10. Use Cases

* CM Data Retrieval: This class can be used in environments where Configuration Management data needs to be fetched for various network elements (DUs, CUs, cells, etc.) and their child attributes.
* Efficiency Needs: The multi-threading and DN chunking features make it suitable for large-scale networks where hundreds or thousands of DNs need to be processed efficiently.

### 11. Conclusion

The CmManager class is an efficient, multi-threaded tool designed for managing and retrieving Configuration Management data using the SON SDK APIs. By supporting both single-threaded and multi-threaded operations, as well as optimizing data processing through DN chunking, it ensures performance is maximized when working with large datasets.

***Overview of the Custom Class 'CellDetails'***

**Overview of the Custom Class 'CellDetails'**

In this project, wherever string manipulation is performed, we need to replace it with the name of our custom class. For example, in the code, you will find various string manipulations like **cell\_name[1:8]** for site\_name, **cell\_name[1:9]** for site\_name, **cell\_name[1:-2]** for site\_name, **cell\_name[1:10]** for sector\_name, etc.

Our task is to identify these instances and replace them with data from our custom function. This approach will have several benefits, such as eliminating repetitive string manipulation and enabling our function to retrieve data from the cache, improving efficiency.

Currently, the class I've created is not fully accurate. It may require modifications based on new challenges or ideas that arise.

**Class Name: 'CellDetails'**

The 'CellDetails' class provides essential information, such as **site\_name, sector\_name**, and **band\_name**, from site data (target\_data) and cell data. Additionally, it caches the data to avoid repeated retrieval. If the required data is not found, the class will retrieve the data from enetsdk and return it.

### **Overview**

The CellDetails class is designed to manage and cache key details related to cellular nodes, such as the node name, site name, sector, and band. This class utilizes the enetsdk library to interact with a database of cell details and provides methods to retrieve and store information related to cells in a cache, improving efficiency and reducing redundant calculations.

### **Class Breakdown**

1. **\_\_init\_\_(self, script\_data)**:
   * This is the initializer method where the class is instantiated.
   * **Parameters**:
     + script\_data: An object containing the necessary methods for retrieving data. It's used to interact with other systems (e.g., enetsdk and get\_cm\_reader).
   * **Attributes**:
     + cache: A dictionary to store cached cell details, avoiding repeated data retrieval.
     + script\_data: The script data object passed to interact with external data sources.
     + cdb: The enetsdk's GetCellDB() method, which helps retrieve cell-related data.
2. **run\_cache(self, targets: list[object] = None, cells: list[str] = None)**:
   * This method caches cell details for a list of target objects or cell names.
   * **Parameters**:
     + targets: An optional list of target objects representing cells.
     + cells: An optional list of cell names (strings) to resolve and process.
   * If cells are provided, it retrieves the corresponding target objects and extends the list of targets, which is then passed to the get\_cell\_details() method to build a cache of cell data.
3. **\_validate\_and\_get\_cell\_name(self, target: object, cell: str) -> str**:
   * This helper function ensures that either the target or cell is provided, but not both or neither. It returns the cell name.
   * **Parameters**:
     + target: The target object representing a cell.
     + cell: The cell name (string).
   * **Returns**: The cell name as a string.
4. **\_get\_cached\_value(self, cell\_name: str, key: str)**:
   * This method retrieves a specific value from the cache based on the cell\_name and key (like node name, site name, etc.).
   * If the data is not found in the cache, it fetches the required details from enetsdk and updates the cache.
   * **Parameters**:
     + cell\_name: The name of the cell.
     + key: The specific key (e.g., nodeName, siteName, etc.) to retrieve.
   * **Returns**: The value associated with the key, or None if not found.
5. **get\_node\_name(self, target: object = None, cell: str = None) -> str**:
   * Retrieves the node name for a given cell.
   * **Parameters**:
     + target: The target object representing the cell.
     + cell: The cell name.
   * **Returns**: The node name for the cell, or None if not found.
6. **get\_site\_name(self, target: object = None, cell: str = None) -> str**:
   * Retrieves the site name for a given cell.
   * **Parameters**:
     + target: The target object representing the cell.
     + cell: The cell name.
   * **Returns**: The site name for the cell, or None if not found.
7. **get\_sector(self, target: object = None, cell: str = None) -> str**:
   * Retrieves the sector for a given cell.
   * **Parameters**:
     + target: The target object representing the cell.
     + cell: The cell name.
   * **Returns**: The sector for the cell, or None if not found.
8. **get\_band(self, target: object = None, cell: str = None) -> str**:
   * Retrieves the band for a given cell.
   * **Parameters**:
     + target: The target object representing the cell.
     + cell: The cell name.
   * **Returns**: The band for the cell, or None if not found.
9. **get\_cell\_details(self, targets: list[object]) -> dict**:
   * This method processes a list of target objects and returns a dictionary containing detailed information about the cells, such as nodeName, siteName, sector, and band.
   * **Parameters**:
     + targets: A list of target objects (cells) to process.
   * **Returns**: A dictionary with cell names as keys and cell details (node name, site name, sector, band) as values.
   * **How It Works**:
     + First, it retrieves all site names for the given targets using get\_cm\_reader().read\_cells\_by\_uids().
     + Then, it processes each target:
       - Retrieves the cell name.
       - Determines the node name based on the technology (NR, LTE, 4G-NBIoT).
       - Retrieves the site name using the UID.
       - Derives the sector and band.
     + Finally, it stores all the retrieved data in a dictionary and returns it.

### **Key Features and Benefits**

* **Caching**: The class stores the data in a cache to avoid redundant operations.
* **Efficiency**: Data retrieval is optimized by directly accessing cached values, reducing the need for repetitive calculations.
* **Flexibility**: It can handle both target objects and cell names, automatically resolving and processing them.
* **Error Handling**: Exceptions are caught when retrieving site names or determining the band, ensuring the class handles errors gracefully.

### **Conclusion**

The CellDetails class simplifies and optimizes the process of extracting and caching cell details like node name, site name, sector, and band, while avoiding the need for repetitive string manipulations. By using caching and error handling, it enhances performance and reliability in retrieving and managing cell data.

# ***CmManager Class***

### **1. Overview**

The CmManager class is designed to handle data retrieval for Configuration Management (CM) using SON SDK APIs. This class provides methods to retrieve various types of attributes (including child attributes) related to DNs (Distinguished Names), using both single-threaded and multi-threaded operations. It also supports chunking DN lists to optimize performance when dealing with large sets of DNs.

### **2. Purpose**

The CmManager class facilitates interaction with the EMS (Element Management System) for retrieving attributes and child attributes of Configuration Management data. It provides an efficient way to process large DN lists by utilizing threading, ensuring the operation is optimized for both small and large datasets.

### **3. Dependencies**

* **enet\_ems.ems. EMSManager:** This class is used to interact with the EMS to fetch attributes and child attributes.
* **ThreadPoolExecutor:** Used for executing functions in multiple threads for faster processing when enable\_multithread is set to True.
* **tmo.chunk:** This utility function is used for chunking the DN list for processing in batches, ensuring large sets are handled efficiently.

### **4. Key Attributes**

* **script\_data (object):** Initialization data required for interacting with the EMS.
* **ems (EMSManager):** An instance of the EMSManager class used for managing operations in the EMS.
* **enable\_multithread (bool):** Flag to enable or disable multi-threading. The default is True, enabling multi-threading.
* **dn\_chunk\_size (int):** Defines the maximum size of DN chunks used for processing (default is 300).

### **5. Key Methods**

#### **5.1 \_\_init\_\_(self, script\_data, enable\_multithread=True)**

**Purpose:**Initializes an instance of the CmManager class. It sets up the script data, the EMS manager, and the multi-threading flag.

**Parameters:**

* script\_data (object): Data required for initialization, usually containing information such as region and other configurations.
* enable\_multithread (bool): A flag to enable multi-threading (default is True).

**Example Usage:**

cm\_manager = CmManager(script\_data)

#### **5.2 get\_attributes(self, dn\_list, parameter\_list, oss\_value=True)**

**Purpose:**Retrieves attributes for a list of DNs (Distinguished Names).

**Parameters:**

* dn\_list (list): A list of DNs to retrieve attributes for.
* parameter\_list (list): A list of parameters (attributes) to fetch for each DN.
* oss\_value (bool): Whether to retrieve the value from OSS (default is True).

**Returns:**A dictionary where the keys are DNs and the values are the corresponding attributes.

**Example Usage:**

attributes = cm\_manager.get\_attributes(dn\_list, parameter\_list)

#### **5.3 get\_child\_attributes(self, dn\_list, mo\_list, param\_filter=None, value\_filter=None, oss\_value=True, return\_by\_dn=False, dn\_only=False)**

**Purpose:**Retrieves child attributes for a list of DNs.

**Parameters:**

* dn\_list (list): A list of parent DNs to retrieve child attributes for.
* mo\_list (list): A list of Managed Object (MO) types to filter by.
* param\_filter (dict, optional): A filter to specify which attributes to fetch for each MO.
* value\_filter (dict, optional): A filter to specify attribute values to filter by.
* oss\_value (bool): Whether to retrieve the value from OSS (default is True).
* return\_by\_dn (bool): Whether to return the results grouped by DN (default is False).
* dn\_only (bool): If set to True, only child DNs will be returned.

**Returns:**A dictionary of child attributes for the given DNs.

**Example Usage:**

child\_attributes = cm\_manager.get\_child\_attributes(dn\_list, mo\_list)

#### **5.4 get\_child\_mo\_dns(self, dn\_list, mo\_list)**

**Purpose:**Retrieves child DNs for a list of parent DNs.

**Parameters:**

* dn\_list (list): A list of parent DNs to retrieve child DNs for.
* mo\_list (list): A list of Managed Object types to filter by.

**Returns:**A list of child DNs.

**Example Usage:**

child\_dns = cm\_manager.get\_child\_mo\_dns(dn\_list, mo\_list)

#### **5.5 \_\_run\_in\_threads(self, func, dn\_list, \*\*kwargs)**

**Purpose:**Executes a function in multiple threads to process a list of DNs in parallel.

**Parameters:**

* func (callable): The function to run in threads.
* dn\_list (list): The list of DNs to process.
* \*\*kwargs: Additional parameters to pass to the function.

**Returns:**A dictionary of results from the function executed in threads.

### **6. Example Usage**

# Initialize CmManager

cm\_manager = CmManager(script\_data)

# Get attributes for a list of DNs

attributes = cm\_manager.get\_attributes(dn\_list, parameter\_list)

# Get child attributes for a list of DNs and filter by value

child\_attributes = cm\_manager.get\_child\_attributes(dn\_list, mo\_list, value\_filter={'keyId': 'CXC4012691'})

# Retrieve child DNs for parent DNs

child\_dns = cm\_manager.get\_child\_mo\_dns(dn\_list, mo\_list)

### **7. Multi-threading**

By default, the class supports multi-threading to speed up the processing of large DN lists. You can disable multi-threading by setting enable\_multithread = False during initialization.

cm\_manager = CmManager(script\_data, enable\_multithread=False)

### **8. Error Handling**

The class includes error handling, especially when calling external functions like get\_child\_attributes. If an exception is raised during processing, it is logged and the method gracefully returns an empty dictionary.

### **9. Performance Optimization**

* **DN Chunking:** The DN list is chunked into smaller batches, typically of 300 DNs (can be adjusted if necessary), which helps in managing the performance when dealing with large sets of DNs.
* **Multi-threading:** If multi-threading is enabled, the tasks are split into smaller chunks, and each chunk is processed in parallel to improve performance.

### **10. Use Cases**

* **CM Data Retrieval:** This class can be used in environments where Configuration Management data needs to be fetched for various network elements (DUs, CUs, cells, etc.) and their child attributes.
* **Efficiency Needs:** The multi-threading and DN chunking features make it suitable for large-scale networks where hundreds or thousands of DNs need to be processed efficiently.

### **11. Conclusion**

The CmManager class is an efficient, multi-threaded tool designed for managing and retrieving Configuration Management data using the SON SDK APIs. By supporting both single-threaded and multi-threaded operations, as well as optimizing data processing through DN chunking, it ensures performance is maximized when working with large datasets.

### 

### ***This is what i understand***

### **Refactoring and Optimization Overview for CmManager**

**This document outlines the recent changes and enhancements made to the CmManager module, focusing on replacing the old structure, improving performance, and introducing a new class for efficient data retrieval.**

#### **1. Transition to a New Structure**

* **The earlier structure primarily relied on attributes such as site\_name and sector\_name.**
* **The objective of this update was to replace the old structure across all existing flows with a new, more reliable format.**
* **All occurrences of the old flow have been thoroughly reviewed and replaced with the new structure, ensuring consistency across the codebase.**

#### **2. Enhancing Efficiency and Reliability**

* **Challenge Identified:**
  + **The previous process required fetching data directly from the SDK API for every request. This approach was not only slow but also less efficient.**
* **Solution Implemented:**
  + **To address this, a caching mechanism has been introduced to minimize repetitive API calls.**
  + **This enhancement ensures faster response times and reduces the dependency on continuous SDK API requests.**

#### **3. Introduction of the CellDetails Class**

* **To further optimize data retrieval, a new class named CellDetails has been implemented.**
* **Purpose of the Class:**
  + **The CellDetails class acts as a bridge between the new structure and efficient data caching.**
  + **It retrieves data in the new format while storing frequently accessed data in a cache for faster subsequent access.**
* **Key Benefits:**
  + **Eliminates repeated API calls for identical data.**
  + **Accelerates the data retrieval process by leveraging in-memory caching.**
  + **Ensures a seamless transition to the new structure while maintaining performance.**

#### **Conclusion**

**With these updates, the CmManager module now supports a more robust, efficient, and reliable workflow. The introduction of the CellDetails class addresses long-standing issues related to repetitive API calls and slow data retrieval, making the system significantly faster and more adaptable to future requirements.**

**These changes not only improve the overall performance but also align the module with modern development practices, ensuring scalability and maintainability.**