1. **List of team members:**
   a. Sakshi Tokekar – SXT230143
   b. Asritha Korrapati – ACK200002
   c. Gayatri Rajesh Mangire – GRM210001

2. **Select two coupling and two cohesion metrics for measuring the coupling/cohesion of MangoDB.**

   Following metrics were selected for analysis:

   - **Cohesion Metric:** LCOM (Lack of Cohesion in Methods):
     - LCOM measures how closely methods in a class are related to its instance variables. High cohesion indicates a well-structured class, while low cohesion suggests complexity and potential for subdivision. LCOM is calculated as the ratio of methods not accessing a specific data field, averaged across all fields. A low score means high cohesion; a high score indicates low cohesion.
   - **Cohesion Metric:** WMC (Weighted Method Complexity):
     - WMC measures the complexity of a class by summing the complexities of all its methods. It indicates the effort required to develop and maintain the class. WMC is calculated as the sum of the complexities of all methods defined in a class, where complexity is typically measured using cyclomatic complexity. A lower WMC suggests better abstraction and easier maintenance, while a higher WMC indicates increased complexity, potentially reduced reusability, and greater effort for testing and maintenance
   - **Coupling Metric:** RFC (Response for Class)
     - RFC measures class complexity based on method calls. It sums the number of methods in the class (excluding inherited ones) and the distinct method calls they make (counted once). A low RFC indicates low coupling and better modularity, while a high RFC suggests higher complexity and tighter coupling.
   - **Coupling Metric:** CBO (Coupling Between Objects)
     - CBO measures the interdependence between classes in a system. An object of a class is considered coupled to another if methods of one class use methods or attributes of another class. A low CBO score can enhance modularity, while a high CBO indicates a strongly coupled system, which can create difficulties in maintenance.

3. **Software analysis tool**

   We picked MetrcisReloaded - Automated code metrics plugin for IntelliJ IDEA

4. Computing the metrics for each class:

2	Class	Cyclic	Dcy	Dcy*	Dpt	Dpt*	LCOM
50	com.serotonin.mango.web.dwr.DataSourceEditDwr	0	120	538	0	0	41
51	com.serotonin.mango.vo.report.ReportVO	408	5	509	9	573	33
52	com.serotonin.mango.util.test.DummyServletContext	0	0	0	0	0	23
53	com.serotonin.mango.rt.event.type.EventType	408	17	509	48	573	19
54	com.serotonin.mango.vo.User	408	18	509	61	573	16
55	com.serotonin.mango.vo.dataSource.mbus.MBusDataS	408	12	509	3	573	16
56	br.org.scadabr.vo.dataSource.dnp3.Dnp3PointLocatorVO	408	4	509	4	573	15
57	com.serotonin.mango.view.component.ViewComponent	408	20	509	12	573	14
58	com.serotonin.mango.view.text.BaseTextRenderer	408	11	509	9	573	14
59	com.serotonin.mango.Common	408	10	509	188	573	13
60	com.serotonin.mango.vo.dataSource.openv4j.OpenV4JDataSourceVO	408	11	509	3	573	13
61	com.serotonin.mango.vo.dataSource.ebro.EBI25PointLocatorVO	408	10	509	7	573	12
62	com.serotonin.mango.rt.dataImage.types.MangoValue	408	6	509	96	573	11
63	com.serotonin.mango.rt.event.EventInstance	408	8	509	20	573	11
64	com.serotonin.mango.view.component.PointComponent	408	7	509	14	573	11
65	com.serotonin.mango.rt.dataSource.pop3.Pop3PointLocatorRT	408	3	509	2	573	10
66	com.serotonin.mango.rt.event.detectors.PointEventDetectorRT	408	11	509	18	573	10
67	com.serotonin.mango.vo.DataPointVO	408	23	509	121	573	10
68	com.serotonin.mango.vo.dataSource.DataSourceVO	408	14	509	60	573	10
69	com.serotonin.mango.vo.dataSource.ebro.EBI25DataSourceVO	408	11	509	5	573	10
70	com.serotonin.mango.vo.publish.httpSender.HttpSenderVO	408	11	509	4	573	10
71	com.serotonin.mango.web.dwr.WatchListDwr	0	23	510	1	2	10
72	com.serotonin.mango.web.dwr.beans.EBI25LoggerInfo	0	0	0	1	2	10
73	com.serotonin.mango.rt.dataSource.DataSourceRT	408	9	509	56	573	9

2	Class	CBO	DIT	LCOM	NOC	RFC	WMC
3	com.serotonin.mango.web.dwr.DataSourceEditDwr	120	3	41	0	668	241
4	com.serotonin.mango.rt.RuntimeManager	87	1	4	0	200	146
5	com.serotonin.mango.rt.dataSource.bacnet.BACnetIPDataSourceRT	20	3	9	0	167	108
6	com.serotonin.mango.rt.dataSource.viconics.ViconicsDataSourceRT	36	3	2	0	155	59
7	com.serotonin.mango.web.dwr.MiscDwr	29	2	7	0	154	77
8	com.serotonin.mango.web.dwr.ViewDwr	40	2	7	0	154	100
9	com.serotonin.mango.web.dwr.beans.ImportTask	37	2	2	0	146	93
10	com.serotonin.mango.vo.DataPointVO	141	1	10	0	140	126
11	com.serotonin.mango.db.dao.EventDao	36	3	3	0	127	99
12	com.serotonin.mango.Common	194	1	13	0	122	87
13	com.serotonin.mango.rt.dataSource.modbus.ModbusDataSource	20	3	1	2	122	48
14	com.serotonin.mango.db.dao.DataPointDao	74	3	4	0	118	82
15	com.serotonin.mango.rt.dataSource.persistent.PersistentDataSourceRT.ConnectionHandler	32	1	1	0	117	45
16	com.serotonin.mango.web.dwr.WatchListDwr	24	2	10	0	116	57
17	com.serotonin.mango.db.dao.PointValueDao	40	3	3	0	115	79
18	com.serotonin.mango.vo.event.EventHandlerVO	28	1	2	0	115	168
19	com.serotonin.mango.web.dwr.beans.EBI25InterfaceReader	16	1	5	0	113	28
20	com.serotonin.mango.rt.dataImage.DataPointRT	87	1	4	0	104	91
21	com.serotonin.mango.vo.User	74	1	16	0	103	91
22	com.serotonin.mango.web.dwr.EventHandlersDwr	34	2	5	0	102	28
23	com.serotonin.mango.vo.event.MaintenanceEventVO	19	1	3	0	97	102
24	com.serotonin.mango.vo.event.PointEventDetectorVO	49	2	3	0	96	120
25	com.serotonin.mango.rt.dataSource.sql.SqlDataSourceRT	17	3	1	0	95	45
26	com.serotonin.mango.web.dwr.BaseDwr	49	1	8	20	93	44

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Class	CBO	Cyclic	Dcy	Dcy*	Dpt	Dpt*	PDcy	PDpt				
2	com.serotonin.mango.Common	194	408	10	509	188	573	7	77				
3	com.serotonin.mango.vo.DataPointVO	141	408	23	509	121	573	12	46				
4	com.serotonin.mango.rt.dataImage.PointValueTime	129	408	6	509	123	573	2	47				
5	com.serotonin.mango.web.dwr.DataSourceEditDwr	120	0	120	538	0	0	47	0				
6	com.serotonin.mango.DataTypes	105	408	2	509	104	573	2	57				
7	com.serotonin.mango.rt.dataImage.types.MangoValue	96	408	6	509	96	573	2	38				
8	com.serotonin.mango.web.ContextWrapper	93	408	11	509	86	573	7	37				
9	com.serotonin.mango.rt.event.type.AuditEventType	92	408	14	509	79	573	9	31				
10	com.serotonin.mango.rt.RuntimeManager	87	408	42	509	52	573	24	25				
11	com.serotonin.mango.rt.dataImage.DataPointRT	87	408	28	509	60	573	14	36				
12	com.serotonin.mango.util.ExportCodes	84	0	1	1	83	575	1	43				
13	com.serotonin.mango.db.dao.DataPointDao	74	408	22	509	57	573	11	26				
14	com.serotonin.mango.vo.User	74	408	18	509	61	573	11	22				
15	com.serotonin.mango.vo.dataSource.DataSourceVO	70	408	14	509	60	573	8	44				
16	com.serotonin.mango.rt.dataSource.DataSourceRT	64	408	9	509	56	573	7	48				
17	com.serotonin.mango.rt.dataSource.PointLocatorRT	55	0	0	0	55	574	0	51				
18	com.serotonin.mango.vo.permission.Permissions	55	408	15	509	41	573	8	13				
19	com.serotonin.mango.util.LocalizableJsonException	54	0	0	0	54	574	0	28				
20	com.serotonin.mango.vo.event.EventTypeVO	54	408	11	509	45	573	2	36				
21	com.serotonin.mango.rt.event.type.EventType	53	408	17	509	48	573	7	26				
22	com.serotonin.mango.vo.event.PointEventDetectorVO	49	408	27	509	35	573	9	15				
23	com.serotonin.mango.web.dwr.BaseDwr	49	408	23	509	26	573	14	6				
24	com.serotonin.mango.view.ImplDefinition	48	0	0	0	48	575	0	9				
25	com.serotonin.mango.db.upgrade.DBUpgrade	43	408	5	509	39	573	4	2				
26	com.serotonin.mango.db.dao.PointValueDao	40	408	23	509	21	573	7	10				
27	com.serotonin.mango.db.dao.ReportDao	40	0	33	518	7	15	12	7				
28	com.serotonin.mango.web.dwr.ViewDwr	40	0	39	512	1	2	13	1				
29	com.serotonin.mango.web.dwr.beans.ImportTask	37	408	37	509	2	573	17	2				
30	com.serotonin.mango.db.dao.EventDao	36	408	24	509	17	573	7	8				
31	com.serotonin.mango.rt.dataSource.viconics.ViconicsDataSourceRT	36	408	35	509	2	573	17	2				
32	com.serotonin.mango.db.dao.UserDao	35	408	10	509	25	573	6	13				
33	com.serotonin.mango.vo.dataSource.DataSourceVO.Type	35	408	28	509	35	573	24	31				
34	com.serotonin.mango.rt.EventManager	34	408	19	509	18	573	10	15				
35	com.serotonin.mango.web.dwr.EventHandlersDwr	34	0	34	512	0	0	13	0				
36	com.serotonin.mango.db.dao.BaseDao	33	408	3	509	30	573	3	2				

A. LCOM (Lack of Cohesion in Methods)

➢ **Highest Cohesion Classes:**

1. com.serotonin.mango.db.BasePooledAccess:

a. The BasePooledAccess class in the Mango system is all about managing database connections using a connection pool. It sets up the pool, runs SQL scripts, and gives access to the data source. When you look at what it does, it's clear that this class has Functional Cohesion. Functional Cohesion happens when a class or module is focused on doing one specific thing or achieving a single goal. In this case, the class is all about handling database connections and running SQL scripts, which are closely related tasks that work together to achieve one main purpose.
Example:

i. initializeImpl() sets up the connection pool.
ii. runScript() runs SQL scripts.
iii. getDataSource() gives access to the data source.

These methods are all focused on the single goal of managing database connections and operations, which aligns with functional cohesion.

b. Why It Is Not Another Type of Cohesion:

- **Coincidental Cohesion:**
  Reasoning for why not: The methods in BasePooledAccess are not arbitrarily grouped; they are all related to the management of database connections. They serve a unified purpose, making the grouping intentional and meaningful.
  Example: Methods like initializeImpl() and runScript() are both related to database operations, demonstrating a clear and purposeful grouping.

- **Logical Cohesion:**
  Reasoning for why not: The methods in BasePooledAccess are not just logically similar; they are functionally related to the management of database connections. The class is not a collection of loosely related actions but a focused module for connection management.
  Example: initializeImpl() and terminate() are not independent logic-based actions; they contribute to the lifecycle management of the database connection pool.

- **Temporal Cohesion:**
  Reasoning for why not: While initializeImpl() and terminate() are related to the lifecycle of the connection pool, the class also includes methods like runScript() and getDataSource() that can be executed independently of timing constraints. The class is not strictly focused on actions that must occur at specific times.

Example: runScript() can be executed at any time, not just during initialization or shutdown.

- **<u>Procedural Cohesion</u>:**
  Reasoning for why not: The methods in BasePooledAccess do not enforce a strict sequence of execution. They can be called independently and do not rely on a specific order.
  Example: getDataSource() can be called at any time, independent of initializeImpl() or runScript().

- **<u>Communicational Cohesion</u>:**
  Reasoning for why not: While the methods in BasePooledAccess do operate on the same data source, they are functionally related to the management of database connections. This aligns more closely with functional cohesion than communicational cohesion.
  Example: initializeImpl() and getDataSource() both work with the data source, but they are functionally related to connection management.

- **<u>Informational Cohesion</u>:**
  Reasoning for why not: The methods in BasePooledAccess are not just operating on the same data structure; they are focused on a single functional goal of managing database connections.
  Example: runScript() and getDataSource() are not just manipulating the same data; they are contributing to the overall functionality of database connection management.

c. The BasePooledAccess class exhibits high cohesion. It is highly cohesive because it is focused on a single, well-defined task: managing database connections. All methods in the class contribute to this goal, making it a tightly focused and well-organized module.

d. The BasePooledAccess class has an LCOM (Lack of Cohesion in Methods) score of 2, which indicates high cohesion. A low LCOM score suggests that the methods in the class share common fields (e.g., dataSource), which is a sign of good cohesion. The class is focused on managing database connections, and most methods operate on the dataSource field, which is central to the class's functionality.

2. com.serotonin.mango.db.dao.DataSourceDao:

a. The DataSourceDao class in the Mango system is responsible for managing data sources. It handles tasks like retrieving, saving, updating, and deleting data sources, as well as copying data sources and their permissions. When you look at what it does, it's clear that this class has Functional Cohesion. Functional Cohesion happens when a class or module is focused on doing one specific thing or achieving

a single goal. In this case, the class is all about managing data sources and their related operations, which are closely related tasks that work together to achieve one main purpose.

Examples:

i. getDataSources() retrieves a list of all data sources.
ii. getDataSource() fetches a specific data source by its ID.
iii. saveDataSource() saves or updates a data source.
iv. deleteDataSource() deletes a data source and cleans up related data.

These methods are all focused on the single goal of managing data sources, which aligns with functional cohesion.

b. Why It Is Not Another Type of Cohesion:

- **Coincidental Cohesion:**
  Reasoning for why not: The methods in DataSourceDao are not random, they're all about managing data sources. They're grouped together because they serve the same purpose, not because someone just threw them in there.
  Example: Methods like getDataSources() and saveDataSource() are both about data source management, so they belong together.

- **Logical Cohesion:**
  Reasoning for why not: The methods in DataSourceDao are not performing similar operations, they're all working toward the same goal of managing data sources. It's not just a loose grouping of related actions.
  Example: getDataSource() and deleteDataSource() aren't just similar actions; they're part of the same process of managing data sources.

- **Temporal Cohesion:**
  Reasoning for why not: While some methods in DataSourceDao may be executed in sequence (e.g., retrieving a data source before deleting it), the class isn't strictly about timing. Methods like getDataSources() and copyDataSource() can be called at any time.
  Example: getDataSources() can be run whenever needed, not just during setup or shutdown.

- **Procedural Cohesion:**
  Reasoning for why not: The methods in DataSourceDao don't need to be called in a particular order. You can call saveDataSource() anytime, regardless of whether getDataSource() has been called first.
  Example: saveDataSource() and deleteDataSource() are independent and don't rely on a specific sequence.

- **Communicational Cohesion:**

Reasoning for why not: While the methods in DataSourceDao do work with the same data that are the data sources, they're all functionally related to managing data sources. It's not just about sharing data.
Example: getDataSource() and deleteDataSource() both use data sources, but they're both part of the same goal of managing them.

- **<u>Informational Cohesion</u>:**
  Reasoning for why not: The methods in DataSourceDao aren't just manipulating the same data, they're all also working towards the same functional goal of managing data sources.
  Example: saveDataSource() and deleteDataSource() aren't just working on the same data; they're both part of the same data source management process.

c. The DataSourceDao class exhibits high cohesion. It's highly cohesive because it's focused on one clear task: managing data sources. All the methods in the class contribute to this goal, making it a tightly focused and well-organized module

d. The DataSourceDao class has an LCOM (Lack of Cohesion in Methods) score of 2, which is very low. A low LCOM score means the class is highly cohesive because most of its methods share common fields or operate on the same data (e.g., data sources). This shows that the class is well-designed and focused on a single responsibility.

3. com.serotonin.mango.db.dao.EventDao.PendingEventCacheEntry

a. The PendingEventCacheEntry class in the Mango system is a small, inner class within the EventDao class. Its primary purpose is to manage a cache of pending events for a specific user. The class holds a list of pending events and tracks the creation time of the cache entry to determine if it has expired. Given its structure and functionality, the PendingEventCacheEntry class exhibits Informational Cohesion. Informational Cohesion occurs when a module or class is designed to manage and operate on a specific set of data, providing access to or manipulation of that data. In this case, the class is focused on managing a list of pending events and their cache expiration time, which are closely related data elements.
Examples:
   i. The list field holds the list of pending events.
   ii. The createTime field tracks when the cache entry was created.
   iii. The hasExpired() method checks if the cache entry has expired based on the current time and the creation time.

These elements are all focused on managing the cache entry's data, which aligns with informational cohesion.

b. Why It Is Not Another Type of Cohesion:

- **Coincidental Cohesion:**
  Reasoning for why not: The methods and fields in PendingEventCacheEntry are not arbitrarily grouped; they are all related to managing the cache entry for pending events. They serve a unified purpose, making the grouping intentional and meaningful.
  Example: The list field and hasExpired() method are both related to the cache entry's data, demonstrating a clear and purposeful grouping.

- **Logical Cohesion:**
  Reasoning for why not: The methods and fields in PendingEventCacheEntry are not just logically similar; they are functionally related to managing the cache entry. The class is not a collection of loosely related actions but a focused module for cache management.
  Example: The list field and createTime field are not just similar data; they are both part of the same cache entry management process.

- **Temporal Cohesion:**
  Reasoning for why not: While the createTime field is related to the time of cache creation, the class is not strictly focused on actions that must occur at specific times. The hasExpired() method can be called at any time, independent of when the cache entry was created.
  Example: The hasExpired() method can be called at any time, not just during a specific phase of the application lifecycle.

- **Procedural Cohesion:**
  Reasoning for why not: The methods in PendingEventCacheEntry do not enforce a strict sequence of execution. They can be called independently and do not rely on a specific order.
  Example: The getList() method can be called independently of hasExpired(), and there is no requirement for one to be executed before the other.

- **Communicational Cohesion:**
  Reasoning for why not: While the methods in PendingEventCacheEntry do operate on the same data (the cache entry), they are functionally related to managing the cache entry. This aligns more closely with informational cohesion than communicational cohesion.
  Example: The list field and hasExpired() method both work with the cache entry's data, but they are functionally related to cache management.

- **Functional Cohesion:**
  Reasoning for why not: The PendingEventCacheEntry class is not performing a single task; rather, it is managing a set of related data (the cache entry). While

it is cohesive, it is more focused on data management than on performing a specific function.

Example: The class includes methods for accessing the cache entry's data and checking its expiration, which are related but not a singular function.

c. The PendingEventCacheEntry class exhibits high cohesion. It is highly cohesive because it is focused on a single, well-defined task: managing the cache entry for pending events. All methods and fields in the class contribute to this goal, making it a tightly focused and well-organized module.

d. The PendingEventCacheEntry class has an LCOM (Lack of Cohesion in Methods) score of 2, which indicates high cohesion. A low LCOM score suggests that the methods in the class share common fields (e.g., list and createTime), which is a sign of good cohesion. The class is focused on managing the cache entry, and most methods operate on the same data, which is central to the class's functionality.

➢ **Lowest Cohesion Classes:**

1. com.serotonin.mango.web.dwr.DataSourceEditDwr:

a. The DataSourceEditDwr class is a large and complex class that handles the editing and management of various types of data sources in the Mango system. It contains a wide range of methods, each responsible for different functionalities related to data source configuration, validation, and testing. The DataSourceEditDwr class exhibits Logical Cohesion. The class contains many methods that are related to the general concept of editing and managing data sources, but these methods are not tightly related to each other in terms of functionality. For example, the class contains methods for handling different types of data sources (e.g., Modbus, SNMP, OPC, etc.), but each method operates independently and does not share much common logic or data. The class groups methods that are logically related to data source editing, but these methods do not share a common purpose or operate on the same data. Instead, they are grouped because they all fall under the broad category of "data source editing."

b. Why It Is Not Another Type of Cohesion:

- **Coincidental Cohesion:**

  Reasoning for why not: The methods in DataSourceEditDwr are not arbitrarily grouped. They are all related to the editing and management of data sources, so their grouping is meaningful, even if the methods are not tightly related.

  Example: public DwrResponseI18n saveModbusSerialDataSource and public DwrResponseI18n saveSnmpDataSource. These methods are grouped under data source editing but are not arbitrarily placed together. They serve similar purposes but work on different data sources.

- **Temporal Cohesion:**
  Reasoning for why not: The class does not represent a set of operations that happen at the same time. It is a simple data holder with no temporal behaviour.
  Example: editInit() initializes data editing but does not operate in a strict temporal sequence with other methods like saveSqlDataSource() or saveModbusSerialDataSource().

- **Procedural Cohesion:**
  Reasoning for why not: The methods in DataSourceEditDwr do not follow a sequence of steps. Each method is independent and performs a specific task related to data source editing.
  Example: saveModbusPointLocator() and saveSnmpPointLocator() methods can be executed in any order, as they are not dependent on each other.

- **Communicational Cohesion:**
  Reasoning for why not: The methods in DataSourceEditDwr do not operate on the same data. Each method operates on different data related to different types of data sources.
  Example: getPoints() retrieves data points related to a data source but doesn't share a common structure with methods like saveModbusSerialDataSource()

- **Informational Cohesion:**
  Reasoning for why not: The class does not encapsulate a set of related data. Instead, it provides a wide range of methods for editing and managing different types of data sources.
  Example: toggleEditDataSource() modifies a data source but does not use the same structure as methods handling different data source types.

- **Functional Cohesion:**
  Reasoning for why not: The class does not perform a single, well-defined task. Instead, it performs a wide range of tasks related to data source editing, each of which is independent of the others.
  Example: Method saveSqlDataSource() deals only with SQL data sources, whereas saveModbusSerialDataSource() handles Modbus serial data sources. The class lacks a single, well-defined purpose.

c. The DataSourceEditDwr class has low cohesion because it contains many methods that are only loosely related to each other. The methods are grouped under the broad category of "data source editing," but they do not share a common purpose or operate on the same data. This results in a class that is difficult to maintain and understand, as changes to one part of the class may have unintended consequences on other parts.

d. The DataSourceEditDwr class has LCOM = 41, which indicates low cohesion. This is because the class contains many methods that operate on different data and do not share a common purpose. The high LCOM value reflects the class's lack of cohesion and suggests that it could benefit from refactoring to improve its structure and maintainability. In this case, the only field is LOG, and most methods do not interact with it directly. Only a few methods (e.g., those that log errors or debug information) access the LOG field. For example: editInit(), tryDataSourceSave(), getPoints(). Most methods do not share any fields. The class is performing a wide range of tasks, but these tasks are not tightly coupled, leading to a lack of cohesion.

2. com.serotonin.mango.vo.report.ReportVO:

a. The ReportVO class in the Mango system is primarily a data holder that manages report-related settings such as time range, scheduling, recipients, and report content inclusion. Given its purpose and structure, the ReportVO class exhibits Informational Cohesion. Informational Cohesion occurs when a module performs multiple actions, each having its own entry point, but all operations are performed on the same shared data structure. In ReportVO, various getter and setter methods manipulate the same set of attributes related to a report.

Example:

- setFromYear(int fromYear) modifies the fromYear field.
- setToYear(int toYear) modifies the toYear field.
- setSchedule(boolean schedule) modifies the schedule field.

These methods operate on different attributes of a report but belong to the same entity, making the cohesion informational rather than another type.

b. Why It Is Not Another Type of Cohesion:

- **<u>Coincidental Cohesion</u>:**
  Reasoning for why not: The methods in ReportVO are all related to report configuration and storage. They are not arbitrarily grouped but serve a unified purpose of handling report properties.
  Example: Methods like setEmail() and setRecipients() are related to email report distribution, showing intentional grouping.

- **<u>Logical Cohesion</u>:**
  Reasoning for why not: Methods in ReportVO directly operate on attributes of the report object rather than a loosely related set of actions. The class is not just a collection of different report-related actions; instead, it represents a structured data entity.
  Example: setDateRangeType() and setRelativeDateType() are not independent logic-based actions; they contribute to configuring the report.

- **<u>Temporal Cohesion</u>:**

Reasoning for why not: ReportVO does not enforce a strict sequence of execution. The methods are not designed to be executed together based on time.
Example: The constructor initializes date fields, but methods like setSchedulePeriod() or setZipData() can be executed at any time independently.

- **Procedural Cohesion:**
  Reasoning for why not: ReportVO does not enforce a step-by-step execution pattern. The setters and getters operate independently.
  Example: setFromYear(int fromYear) and setToYear(int toYear) are independent and do not follow a strict execution sequence.

- **Communicational Cohesion:**
  Reasoning for why not: ReportVO encapsulates and modifies its own data, but all methods directly pertain to modifying the report. This aligns more with Informational Cohesion.
  Example: setRecipients() and setIncludeEvents() modify different aspects of a report rather than working on the same dataset in a procedural manner

- **Functional Cohesion:**
  Reasoning for why not: ReportVO does not perform a single task; rather, it provides an interface for configuring multiple aspects of a report.
  Example: The class includes methods for setting date ranges, scheduling, recipient lists, and more, which are related but not a singular function.

c. The ReportVO class has moderate cohesion. It is not highly cohesive because it deals with various aspects of a report, from scheduling to recipient management, rather than focusing on a single well-defined function. However, it maintains informational cohesion because all methods manipulate the attributes of a report entity.

d. The ReportVO class has LCOM = 33, indicating low cohesion. A high LCOM suggests that many methods do not share common fields, which can indicate that the class might benefit from refactoring. The class contains many unrelated attributes such as time periods, recipient lists, and scheduling details, making it a broad data holder rather than a tightly focused module. For instance setFromYear(int fromYear) modifies date properties, while setEmail(boolean email) modifies email settings. These functionalities do not necessarily belong in the same class.

e. We can try splitting the class into multiple smaller classes, such as ReportScheduleVO and ReportRecipientsVO, to improve cohesion.

3. **com.serotonin.mango.rt.event.type.EventType:**

   a. The EventType class in the Mango system exhibits Logical Cohesion. Logical cohesion occurs when methods are grouped because they perform similar operations but are invoked based on a control flag or condition. The EventType class provides a framework for different event types, but the actual operations it performs are diverse, ranging from event identification to JSON serialization.
   Examples:
   ⇒ getDataSourceId(), getDataPointId(), and getScheduleId() return different IDs depending on the type of event, but they all follow the same pattern.
   ⇒ getDuplicateHandling() determines how duplicate events should be processed, providing multiple options (DO_NOT_ALLOW, IGNORE, etc.).
   ⇒ JSON serialization methods (jsonSerialize() and jsonDeserialize()) allow different event types to be stored and retrieved, but each event type has different data to serialize.

   The grouping of these methods is based on the general idea of handling events, but each method deals with different aspects (data sources, scheduled events, compound detectors, etc.), which aligns with logical cohesion rather than a more focused type.

   b. Why it is not another type of Cohesion:
   - **Coincidental Cohesion:**
     Reasoning for why not: The methods in EventType are all related to event handling, even though they deal with different event sources. They are not arbitrarily grouped.
     Example: getDataSourceId() and getPublisherId() retrieve IDs relevant to specific event sources rather than being unrelated actions.

   - **Temporal Cohesion:**
     Reasoning for why not: EventType does not enforce a specific execution sequence tied to time. Methods can be invoked at any time as required.
     Example: jsonSerialize() and jsonDeserialize() operate independently without requiring sequential execution.

   - **Procedural Cohesion:**
     Reasoning for why not: The class does not enforce an ordered execution of its methods; each can be used separately.
     Example: getDataPointId() and getEventSourceId() are related to identifying events but are not part of a single ordered process.

   - **Communicational Cohesion:**
     Reasoning for why not: Methods in EventType do not always share the same data; they operate on various unrelated attributes of an event.

Example: getDuplicateHandling() works on duplicate event policies, while getScheduledEventId() retrieves a scheduled event.

- **Informational Cohesion:**
  Reasoning for why not: While EventType contains methods that retrieve information, it does not primarily act as a structured data holder (like a VO class would).
  Example: Unlike a ReportVO class that encapsulates report attributes, EventType provides various unrelated event-handling utilities.

- **Functional Cohesion:**
  Reasoning for why not: EventType is responsible for various event-related functionalities (retrieving IDs, handling serialization, checking event duplication, etc.), which means it does not focus on a single function.
  Example: jsonSerialize() and getReferenceId1() serve different purposes, preventing the class from being functionally cohesive.

c. The EventType class has low to moderate cohesion. While all methods relate to event processing, they serve different purposes and do not operate on a unified data structure. The class acts more as a general framework for handling various event types rather than focusing on a single responsibility.

d. With an LCOM of 19, EventType has low cohesion. A high LCOM means that many methods do not share common fields, suggesting that the class handles multiple unrelated responsibilities.
   Example:
   ⇒ getDataSourceId(), getDataPointId(), and getScheduleId() retrieve IDs but are unrelated to event duplication policies managed by getDuplicateHandling().
   ⇒ jsonSerialize() works with JSON operations but is independent of event-type-specific methods like getReferenceId1().

B. WMC (Weighted Method Complexity)

➢ **Lowest Cohesion Classes:**

1. com.serotonin.mango.web.dwr.DataSourceEditDwr

- **Metrics Overview:**
  - WMC (Weighted Methods per Class): 241
  The class has 241 methods , which is extremely high. This indicates a very high complexity in terms of the number of methods, suggesting that the class is doing too much and is likely violating the Single Responsibility Principle (SRP).
- **Cohesion Analysis:**
  Low Cohesion Characteristics:
  The class exhibits low cohesion because:

- It handles a wide variety of responsibilities, including managing different types of data sources (e.g., Modbus, SNMP, SQL, HTTP, etc.)
- The methods are not logically grouped or related to a single purpose. Instead, they are scattered across multiple functionalities.
- There is no clear separation of concerns, as the class deals with configuration, validation, testing, and data retrieval for numerous data source types.

Internal Method Dependencies:

The methods in the class are loosely related and depend on different external systems (e.g., databases, network protocols, etc.). There is no strong internal dependency between methods, as they are designed to handle different tasks independently.

- **Type of Cohesion:** Logical Cohesion

The class demonstrates logical cohesion because:

- The methods are grouped together because they logically perform similar operations (e.g., saving, validating, and testing data sources), but they are not functionally related to a single task.
- The class is organized around the concept of "data source editing," but the methods within it are not tightly coupled to a single purpose.

- **Why Not Other Types of Cohesion?**

1. Temporal Cohesion: This occurs when elements are grouped because they are executed at the same time, but not necessarily for the same purpose.

In this class, the methods are not executed together; they are invoked based on user interactions or system events.

2. Procedural Cohesion: This occurs when elements are grouped because they must be executed in a specific sequence, but the operations themselves may not be directly related to a single task.

- In this class, the methods are not executed in a specific sequence; they are independent and invoked as needed.

3. Communicational Cohesion: This occurs when elements are grouped because they operate on the same data, but not necessarily for the same purpose.

- In this class, the methods do not operate on the same data; they handle different data sources and configurations.

4. Informational Cohesion: This occurs when elements are grouped because they operate on the same data structure or object, but not necessarily for the same purpose.

- In this class, the methods do not operate on the same data structure; they handle different types of data sources.

5. Functional Cohesion: - This occurs when elements are grouped because they contribute to a single, well-defined task.

- In this class, the methods do not contribute to a single task; they handle multiple unrelated tasks

- **Conclusion:**

The `DataSourceEditDwr` class is a low-cohesion class with a high WMC value of 241 , indicating that it is doing tooo much and is not focused on a single responsibility. The class exhibits logical cohesion because its method are grouped based on their logical similarity (data source editing), but they are not functionally related to a single task. This makes the class difficult to maintain, understand, and reuse. To improve cohesion, the class should be refactored into smaller, more focused classes, each handling a specific type of data source or functionality.

2. com.serotonin.mango.vo.event.EventHandlerVO

- **Metrics Overview:**
  - WMC (Weighted Methods per Class): 168
  The class has 168 methods , which is extremely high. This indicates a very high complexity in terms of the number of methods, suggesting that the class is doing too much and is likely violating the Single Responsibility Principle (SRP) .
- **Cohesion Analysis**:
  Low Cohesion Characteristics:
  The class exhibits low cohesion because:
  - It handles multiple unrelated responsibilities, including:
  - Managing different event handler types (set-point, email, process)
  - Serialization/deserialization logic (`writeObject`, `readObject`).
  - JSON serialization/deserialization (`jsonSerialize`, `jsonDeserialize`).
  - Audit logging for property changes (`createRecipientMessage`, maybeAddPropertyChangeMessage`).
  - Methods are scattered across different functionalities (e.g., email recipient management, process command execution, audit logging).
  - No clear separation of concerns between event handling logic, data persistence, and external system interactions.
  Internal Method Dependencies:
  - Methods like `createRecipientMessage` are used by `jsonSerialize` and `maybeAddPropertyChangeMessage`, but there is no cohesive flow between methods.
  - Serialization/deserialization methods (`writeObject`, `readObject`) are isolated from the core event-handling logic.
  - JSON methods (`jsonSerialize`, `jsonDeserialize`) operate independently of the audit logging logic.
- **Type of Cohesion**: Logical Cohesion
  The class demonstrates logical cohesion because:
  - Methods are grouped together because they logically belong to the concept of "event handling," but they are not functionally related to a single task.
  - For example:
  - Methods for email recipients (`createRecipientMessage`, `jsonSerialize`) are grouped with methods for process commands (`activeProcessCommand`, `inactiveProcessCommand`)
  - Serialization logic is mixed with audit logging and JSON handling.
- **Why Not Other Types of Cohesion?**
  1. Functional Cohesion:
  - This occurs when all methods contribute to a single, well-defined task.
  - Not applicable here : The class handles multiple tasks (email, process commands, audit logging, serialization).
  2. Procedural Cohesion
  - This occurs when methods are executed in a specific sequence for a task.
  - Not applicable here : Methods like `jsonSerialize` and `readObject` are not part of a procedural flow.
  3. Communicational Cohesion:
  - This occurs when methods operate on the same data.
  - Not applicable here : Email recipients, process commands, and audit logs are distinct data types.
  4. Informational Cohesion:
  - This occurs when methods operate on the same data structure

- Not applicable here : No shared data structure binds the methods.

5. Temporal Cohesion:

- This occurs when methods are executed at the same time.
- Not applicable here : Methods are invoked based on different triggers (e.g., JSON parsing vs. audit logging).

- **Conclusion:**
  The class is a low-cohesion class with a high WMC value of 168 , indicating it is doing too much and lacks a focused responsibility. It exhibits logical cohesion because its methods are grouped by their association with "event handling" but are not functionally unified. For example:
  - Email recipient management and process command execution are unrelated functionalities forced into the same class.
  - Serialization/deserialization logic is tangled with JSON handling and audit logging.

3. com.serotonin.mango.rt.RuntimeManager

- **Metrics Overview:**
  - WMC (Weighted Methods per Class): 146
  The class has 146 methods , indicating very high complexity. This suggests the class is handling multiple responsibilities and is likely violating the Single Responsibility Principle (SRP)
- **Cohesion Analysis**:
  Low Cohesion Characteristics
  The class exhibits low cohesion because:
  - It manages a wide variety of runtime components, including data sources, data points, publishers, point links, scheduled events, maintenance events, and compound event detector
  - Methods are scattered across unrelated functionalities (e.g., `startDataPoint`, stopPublisher`, `saveMaintenanceEvent`).
  - No clear separation of concerns between lifecycle management (start/stop), configuration (save/delete), and runtime operations.
  Internal Method Dependencies:
  - Methods like `initialize`, `terminate`, and `joinTermination` depend on shared data tructures (`runningDataSources`, `dataPoints`, `runningPublishers`).
  - Component-specific methods (e.g., `startDataSource`, `stopPointLink`) operate independently but share no cohesive flow or data.
- **Type of Cohesion:** Logical Cohesion
  The class demonstrates logical cohesion because:
  - Methods are grouped under the broad theme of "runtime management," but they are not functionally unified.
  - Example: Methods for managing data sources (`saveDataSource`, `stopDataSource`) are grouped with methods for publishers (`startPublisher`, `stopPublisher`) and point links (`startPointLink`, `stopPointLink`), despite these being distinct concerns.
- **Why Not Other Types of Cohesion?**
  1. Functional Cohesion:
     - This occurs when all methods contribute to a single, well-defined task.
     - Not applicable here : The class handles multiple tasks (data source management, event scheduling, publisher control).
  2. Procedural Cohesion:
     - This occurs when methods are executed in a specific sequence for a task.

- Not applicable here : Methods like `startDataPoint` and `stopPublisher` are not part of a procedural workflow.

3. Communicational Cohesion:
   - This occurs when methods operate on the same data.
   - Not applicable here : Data sources, publishers, and point links use separate data structures (`runningDataSources`, `runningPublishers`, `pointLinks`).
4. Informational Cohesion:
   - This occurs when methods operate on the same data structure.
   - Not applicable here : No shared data structure unifies the class.
5. Temporal Cohesion:
   - This occurs when methods are executed at the same time.
   - Not applicable here : Methods are invoked based on component-specific triggers (e.g., user actions, system events).

- **Conclusion:**

The `RuntimeManager` class is a low-cohesion class with a high WMC value of 146 , indicating it is overloaded with responsibilities. It exhibits logical cohesion because its methods are grouped under the theme of "runtime management" but lack functional unity. For example:
- Managing data sources and publishers in the same class introduces unrelated responsibilities.
- Lifecycle methods (start/stop) are mixed with configuration methods (save/delete).

C. RFC (Response for Class)

➢ **Lowest Coupling Classes:**

It was noticed that majority of classes whose RFC value were 0, 1, 2 were generally trivial. Hence following analysis is for classes with RFC value greater than equal to 3

1. com.serotonin.mango.rt.dataImage.IdPointValueTime

a. The IdPointValueTime class in the Mango system is a simple data structure that holds information about a data point's value, its ID, and the timestamp when the value was recorded. It is designed to store and provide access to this information in a structured way. Given its purpose and structure, the IdPointValueTime class exhibits Data Coupling. Data Coupling occurs when only required data is passed from one module to another. In this case, the class primarily acts as a data holder, and its methods are focused on accessing or modifying its fields, which are passed as parameters or returned as values.

   Examples:
   - The class has fields like id, value, and time, which store the data point's ID, value, and timestamp.
   - Methods like getId(), getValue(), and getTime() provide access to these fields.

   The class does not directly interact with other classes or modules beyond passing or returning these data values. These interactions are minimal and involve only data exchange, which aligns with data coupling.

b. Why it is not another type of Coupling:

- **Content coupling:** Observed when a module directly references the content of another module module and modifies a statement of module Q, or if module P refers to local data of module Q, or if module P branches to a local label of module Q
  Reasoning for why not: The IdPointValueTime class does not modify or rely on the internal workings of other classes. It only stores and provides access to its own data fields.
  Example: The class does not directly manipulate or depend on the internal state of other classes; it only interacts through its own fields.

- **Common coupling:**
  Reasoning for why not: The IdPointValueTime class does not rely on global data or shared state. It encapsulates its own data and does not interact with global variables or shared resources.
  Example: The class does not use or modify any global variables; all data is contained within the class itself.

- **Control coupling:**
  Reasoning for why not: The IdPointValueTime class does not pass control information to other classes. It only passes or returns data values, without influencing the behaviour of other modules.
  Example: The class does not use flags or commands to control the behaviour of other classes; it only provides data.

- **Stamp coupling:**
  Reasoning for why not: The IdPointValueTime class does not share complex data structures with other classes. It only passes or returns simple data values (e.g., id, value, time).
  Example: The class does not pass or receive complex objects; it only deals with simple fields.

c. The IdPointValueTime class exhibits low coupling. It is loosely coupled because it does not depend on the internal workings of other classes and only interacts with them through simple data exchange. This makes the class easy to understand, maintain, and reuse in different contexts.

d. The IdPointValueTime class has an RFC (Response For a Class) score of 3, which indicates low coupling. A low RFC score suggests that the class has a small number of methods and interacts minimally with other classes. This is a sign of good design, as the class is focused on its single responsibility of storing and providing access to data point information.

2. com.serotonin.mango.rt.dataSource.meta.DataPointStateException

a. The DataPointStateException class in the Mango system is a custom exception that is used to indicate an error related to the state of a data point. It extends LocalizableException and includes a dataPointId field to identify the specific data point that caused the exception. Given its purpose and structure, the DataPointStateException class exhibits Data Coupling. Data Coupling occurs when modules or classes interact by passing only the necessary data, without sharing internal data structures or control information. In this case, the class primarily acts as a container for exception-related data, and its interactions with other classes are limited to passing or returning this data.

Examples:

- The class has a dataPointId field to store the ID of the data point that caused the exception.
- The getDataPointId() method provides access to this field.
- The class does not directly interact with other classes beyond passing or returning this data.

These interactions are minimal and involve only data exchange, which aligns with data coupling.

b. Why it is not another type of Coupling:

- **Content coupling:**

  Reasoning for why not: The DataPointStateException class does not modify or rely on the internal workings of other classes. It only stores and provides access to its own data fields.

  Example: The class does not directly manipulate or depend on the internal state of other classes; it only interacts through its own fields.

- **Common coupling:**

  Reasoning for why not: The DataPointStateException class does not rely on global data or shared state. It encapsulates its own data and does not interact with global variables or shared resources.

  Example: The class does not use or modify any global variables; all data is contained within the class itself.

- **Control coupling:**

  Reasoning for why not: The DataPointStateException class does not pass control information to other classes. It only passes or returns data values, without influencing the behaviour of other modules.

  Example: The class does not use flags or commands to control the behaviour of other classes; it only provides data.

- **Stamp coupling:**

Reasoning for why not: The DataPointStateException class does not share complex data structures with other classes. It only passes or returns simple data values eg: dataPointId.
Example: The class does not pass or receive complex objects; it only deals with simple fields.

c. The DataPointStateException class exhibits low coupling. It is loosely coupled because it does not depend on the internal workings of other classes and only interacts with them through simple data exchange. This makes the class easy to understand, maintain, and reuse in different contexts.

d. The DataPointStateException class has an RFC (Response For a Class) score of 3, which indicates low coupling. A low RFC score suggests that the class has a small number of methods and interacts minimally with other classes. This is a sign of good design, as the class is focused on its single responsibility of handling exceptions related to data point states.

3. com.serotonin.mango.rt.dataSource.snmp.Version2c

a. The Version2c class in the Mango system is a subclass of Version1 and is used to represent SNMP version 2c. It inherits the basic functionality from its parent class and overrides the getVersionId() method to return the specific version identifier for SNMP version 2c. Given its purpose and structure, the Version2c class exhibits Data Coupling. Data Coupling occurs when modules or classes interact by passing only the necessary data, without sharing internal data structures or control information. In this case, the class primarily interacts with its parent class and the SnmpConstants class by passing or returning simple data values, such as the version identifier.
Examples:
- The class inherits from Version1 and overrides the getVersionId() method.
- The getVersionId() method returns the version identifier for SNMP version 2c, which is a simple integer value.
- The class does not directly interact with other classes beyond passing or returning this data.

These interactions are minimal and involve only data exchange, which aligns with data coupling.

b. Why it is not another type of Coupling:
- **Content coupling:**
Reasoning for why not: The Version2c class does not modify or rely on the internal workings of other classes. It only interacts with its parent class and the SnmpConstants class by passing or returning simple data values.
Example: The class does not directly manipulate or depend on the internal state of other classes; it only interacts through its own methods and inherited fields.

- **Common coupling:**
  Reasoning for why not: The Version2c class does not rely on global data or shared state. It encapsulates its own data and does not interact with global variables or shared resources.
  Example: The class does not use or modify any global variables; all data is contained within the class itself or passed through method parameters.

- **Control coupling:**
  Reasoning for why not: The Version2c class does not pass control information to other classes. It only passes or returns data values, without influencing the behavior of other modules.
  Example: The class does not use flags or commands to control the behavior of other classes; it only provides data.

- **Stamp coupling:**
  Reasoning for why not: The Version2c class does not share complex data structures with other classes. It only passes or returns simple data values (e.g., the version identifier).
  Example: The class does not pass or receive complex objects; it only deals with simple fields.

c. The Version2c class exhibits low coupling. It is loosely coupled because it does not depend on the internal workings of other classes and only interacts with them through simple data exchange. This makes the class easy to understand, maintain, and reuse in different contexts.

d. The Version2c class has an RFC (Response For a Class) score of 3, which indicates low coupling. A low RFC score suggests that the class has a small number of methods and interacts minimally with other classes. This is a sign of good design, as the class is focused on its single responsibility of representing SNMP version 2c.

➢ **Highest Coupling Classes:**

1. **com.serotonin.mango.web.dwr.DataSourceEditDwr**
   Metrics Overview

   RFC (Response For a Class) : 668

   Coupling Analysis

   Type of Coupling: Common Coupling
   - Global Dependencies : Heavy reliance on the `Common` class (e.g., `Common.getUser()`, `Common.ctx.getRuntimeManager()`). This class acts as a global access point, shared across multiple modules, leading to common coupling .
   - Static Data Access : Frequent use of static methods/fields (e.g., `Common.NEW_ID`), which are shared globally.
   Coupling Characteristics

1. Data Coupling :
   - Methods like `saveVirtualDataSource` pass primitive/string parameters (e.g., `name`, `xid`) to configure `DataSourceVO` objects.
   - Example: `ds.setXid(xid); ds.setName(name).
2. Stamp Coupling :
   - Complex objects (e.g., `DwrResponseI18n`, `DataPointVO`) are passed between methods, but only specific fields are used.
   - Example: `validatePoint` returns a `DwrResponseI18n` object containing validation results.
3. Control Coupling :
   - Parameters like `quantize` in `saveModbusSerialDataSource` indirectly control internal logic of called methods (e.g., data quantization behavior).

- **Internal Method Dependencies**
  - High Dependence on External Classes :
  - DAOs (`DataPointDao`, `EventDao`), VOs (`DataSourceVO`, `DataPointVO`), and utilities (`StringUtils`, `StreamUtils`).
  - Direct calls to runtime managers Common.ctx.getRuntimeManager().saveDataSource(ds)`).
  - Tight Integration with Protocols/Devices :
  - Modbus, SNMP, SQL, HTTP, and other protocols involve calls to external libraries (e.g., `ModbusFactory`, `JMXConnector`).
- **Why Not Other Types of Coupling?**
  - Content Coupling : Absent, as the class does not directly modify internal data of other modules (e.g., uses public setters like `ds.setXid()`).
  - Lower Priority Couplings: Data/Stamp coupling exists but is less impactful than the systemic common coupling via `Common`.
- **Conclusion**
  - Primary Coupling : Common Coupling due to global state access via `Common`.
  - Secondary Issues : High RFC (668) exacerbates complexity, and low cohesion magnifies maintainability challenges.

2. com.serotonin.mango.rt.RuntimeManager

- **Metrics Overview**
  - RFC (Response For a Class) : 200 (extremely high, indicating extensive interactions with external classes and complex internal logic).
- **Type of Coupling**: Stamp Coupling (Primary) with Elements of Common Coupling
  1. Stamp Coupling :
     - Complex Object Passing : Methods like `saveDataSource(DataSourceVO<?> vo)` and `startDataPoint(DataPointVO vo)` pass entire configuration objects (e.g., `DataSourceVO`, `DataPointVO`). These objects contain more data than necessary for the operations, leading to stamp coupling .
     - Example: `saveDataPoint` uses `DataPointVO` but only requires specific fields (e.g., `point.getId()`, `point.isEnabled()`).
  2. Common Coupling :
     - Global Dependencies : Reliance on `Common.ctx.getEventManager()` and `Common.ctx.getRuntimeManager()` introduces common coupling via shared global state.
  3. Data Coupling :

- Primitive/Simple Data Exchange : Methods like `setDataPointValue(int dataPointId, MangoValue value)` pass minimal data (IDs, values), but this is secondary to stamp coupling.

4. Control Coupling :

- Boolean Flags : The `initialize(boolean safe)` method uses a `safe` flag to alter behavior (e.g., disabling data sources during safe mode), introducing control coupling.

- **Internal Method Dependencies**

- High Dependency on External Classes :

- DAOs (`DataSourceDao`, `DataPointDao`, `PublisherDao`), VOs (`DataSourceVO`, `DataPointVO`), and runtime components (`DataSourceRT`, `PublisherRT`).

- Direct calls to external libraries (e.g., `ConcurrentHashMap`, `CopyOnWriteArrayList`).

- Lifecycle Management :

- Methods like `startDataSourcePolling()` and `stopDataPoint()` orchestrate dependencies across data sources, points, and event detectors.

- **Why Not Other Types of Coupling?**

- Content Coupling : Absent. The class does not directly modify internal fields of other classes (e.g., uses setters like `vo.setEnabled(false)`).

- Lower Impact Couplings :

- Data Coupling exists but is overshadowed by stamp coupling.

- Control Coupling is minimal compared to the systemic stamp/common coupling.

- **Conclusion**

Primary Coupling : Stamp Coupling (due to passing complex objects) and Common Coupling (via global `Common` class).

-

3. **com.serotonin.mango.rt.dataSource.bacnet.BACnetIPDataSourceRT**

- **Metrics Overview**

- RFC (Response For a Class) : 167 (very high, indicating extensive interactions with external classes and complex BACnet protocol handling).

- **Type of Coupling**: Stamp Coupling (Primary) with Elements of Common Coupling

1. Stamp Coupling :

- Complex Object Passing : Methods like `pollDevice(RemoteDevice, List<DataPointRT>, long)` and `sendCovSubscriptionImpl(...)` pass BACnet-specific objects (`RemoteDevice`, `ObjectIdentifier`, `PropertyReferences`). These objects contain more data than needed for individual operations.

- Example: `covNotificationReceived` processes `SequenceOf<PropertyValue>`, extracting specific properties for data points.

2. Common Coupling

- Global Context Access : Uses `Common.ctx.getBackgroundProcessing()` and `Common.timer` for background tasks, introducing dependency on global state.

3. Data Coupling :

- Primitive Data Exchange : Methods like `forcePointRead(DataPointRT)` pass minimal data (e.g., `dataPointId`), but this is secondary to stamp coupling.

4. Control Coupling :

- Boolean Flags : The `unsubscribe` parameter in `sendCovSubscription(...)` alters method behavior (e.g., subscription vs. unsubscription).

Internal Method Dependencies

- High Dependency on BACnet4j Library :

- Direct interactions with `LocalDevice`, `RemoteDevice`, `ObjectIdentifier`, and BACnet message types (e.g., `ReadPropertyRequest`, `SubscribeCOVRequest`).
- Integration with Mango Framework :
- Relies on `DataPointRT`, `MangoValue`, and `Common` for runtime management and data processing.
- Concurrency & Timing :
- Uses `TimerTask`, `WorkItem`, and synchronized blocks for polling and COV resubscriptions.

Why Not Other Types of Coupling?

- Content Coupling : Absent. The class does not directly modify internal fields of external objects (e.g., uses BACnet4j API methods like `localDevice.send()`).
- Lower Impact Couplings :
  - Data Coupling occurs but is overshadowed by stamp/common coupling.
  - Control Coupling is limited to specific methods (e.g., `unsubscribe` flag).

- **Conclusion**

  - Primary Coupling : Stamp Coupling (due to BACnet object passing) and Common Coupling (via global `Common` context).
  - Secondary Issues :
  - High RFC (167) reflects complex interactions with BACnet4j and Mango framework.
  - Moderate cohesion is maintained but could be improved by separating protocol logic from data point management.

D. CBO (Coupling Between Objects)

➢ **Highest Coupling Classes:**

1. **com.serotonin.mango.Common**
   a. The com.serotonin.mango.Common class is a utility class in the Mango framework that helps improve code reuse by providing shared methods and constants used throughout the application. It takes care of important tasks like managing user sessions, configurations, and different utility functions. This class has a high Coupling Between Objects (CBO) with a value of 194, which indicates that it relies on many other classes. This class serves as a shared utility that multiple other classes depend on.

   The coupling being measured here is common coupling, which occurs when many classes use the same global variables or resources. The com.serotonin.mango.Common class interacts with many global resources like the session attributes and user data. This is a bad coupling because this setup can result in accidental interactions between classes.

   For example, Some methods, such as getUser(), rely on this shared data, which allows several classes to access these shared properties directly. As a result, any changes to shared resources can affect multiple classes at once, which also affects the maintenance of the system.

b. Why it is not another type of Coupling:

Content coupling:

Reasoning for why not: The Common class makes no direct changes to another module's internal data or logic. It provides shared resources and utility functions without interfering with the internal operations of the other modules. Therefore, this class does not show content coupling.

Example: When getting system time periods, it simply references predefined constants instead of altering another class's internal state.

Control coupling:

Reasoning for why not: The Common class does not pass control switch as an argument. Although it provides shared functionality such as methods and resources, it doesn't directly affect the logic of the other classes. Therefore, this class does not show control coupling.

Example: When storing a user's custom view, it saves it in session storage instead of passing a control switch to influence logic in other modules.

Stamp coupling:

Reasoning for why not: The common class doesn't pass unnecessary data to other modules. This class usually provides shared functionality, rather than passing excessive data that is not needed. For example, the getUser() method only returns the user data without adding any unnecessary data. Therefore, this class does not show stamp coupling.

Example: When creating an HTTP client, it only sets essential parameters like timeouts, avoiding any unnecessary data transfer.

Data coupling:

Reasoning for why not: While Common may use simple data types occasionally, it primarily serves as a source for shared global variables or resources rather than passing clean specific data arguments between the modules. Therefore, this class does not show stamp coupling.

Example: The method for getting user sessions gets data from a global session object rather than passing clean parameters between modules.

c. The Common class exhibits high coupling. It is tightly coupled because many classes depend on it for shared functionality, making modifications more difficult. Changes in one part of the system can create disruptions in other dependent modules.

d. The Common class has an RFC (Response for a Class) score of 194, indicating extremely high coupling. A high RFC score means that the class is deeply interconnected with other classes, making it harder to maintain and modify without affecting multiple components

2. **com.serotonin.mango.vo.DataPointVO**
    a. The DataPointVO class is an important part of the Mango framework that handles data points along with their metadata, logging settings, and configurations. With a high CBO value of 141, it shows that this class depends on a lot of other classes, indicating a tightly connected architecture with high coupling. Since the DataPointVO class represents the different properties and behaviors of data points, it naturally interacts with many other parts of the application, making it an important model within the system. The coupling being measured here is stamp coupling, which occurs when one module passes more data than necessary to another module. The DataPointVO class frequently passes entire objects, such as PointLocatorVO, TextRenderer, and ChartRenderer, rather than just the specific attributes needed. While this isn't the ideal way to structure data, it's still considered a reasonable form of coupling.

    b. Why it is not another type of Coupling:
    Content coupling:

    Reasoning for why not: The DataPointVO class makes no direct changes to another module's internal data or logic. Instead, it relies on object references rather than directly changing another module's data or logic. Therefore, this class does not show content coupling.
    Example: In the getDataTypeMessage() method, DataPointVO calls a method from PointLocatorVO rather than changing its internal state. This means DataPointVO depends on PointLocatorVO, but it does not interfere with its internal structure.

    Common coupling:

    Reasoning for why not: The DataPointVO class avoids using global variables or shared data blocks. Instead, it directly passes objects as parameters to methods, rather than accessing a common global state. Therefore, this class does not show common coupling.
    Example: In the setLoggingType(int loggingType) method, DataPointVO stores the logging type within its own instance instead of using a globally shared variable. This makes sure that changes are only permitted to particular objects instead of impacting multiple modules simultaneously.

    Control coupling:

    Reasoning for why not: The DataPointVO class does not pass control switch as an argument to guide other modules. Instead, it focuses on transferring data structures without directly affecting the logic implemented in the other modules. Therefore, this class does not show control coupling.

Example: During the validation process in the validate(DwrResponseI18n response) method, DataPointVO internally verifies logging configurations but does not pass control-related parameters to influence the behavior of another class. Each module manages its own logic independently.

Data coupling:

Reasoning for why not: Instead of passing only the required data, the DataPointVO class often transfers entire objects, which sometims inclue unused fields. Therefore, this class does not show data coupling.

Example: When setting a point's rendering propertiesin the in the setTextRenderer(TextRenderer textRenderer) method, DataPointVO passes the entire TextRenderer object, even if only specific text formatting details are required. This shows stamp coupling rather than data coupling, as it involves the transfer of extra information.

c. The DataPointVO class exhibits high coupling. It is tightly coupled because it depends on multiple other classes and passes unnecessary data, making modifications and debugging more challenging.

d. The DataPointVO class has an RFC score of 141, indicating high coupling. A high RFC score suggests that the class interacts with many methods across different classes, increasing complexity and interdependencies.

3. com.serotonin.mango.rt.dataImage.PointValueTime

a. The PointValueTime class is an important part of the Mango framework since it is responsible for storing point values along with their associated timestamps. It has a high Coupling Between Objects (CBO) value of 129, which indicates that it interacts with many other classes. PointValueTime works with different types of values, like BinaryValue, MangoValue, and NumericValue, which means it relies on several other classes and has strong connections throughout the system.

The coupling being measured is stamp coupling, which occurs when one module sends more data to another than is necessary. The PointValueTime class frequently passes entire objects, particularly MangoValue and its subclasses, rather than simply the necessary data. As mentioned before, while this isn't the ideal way to structure data, it's still considered a reasonable form of coupling.

b. Why it is not another type of Coupling:

Content coupling:

Reasoning for why not: The PointValueTime class does not change the internal state of MangoValue or any other class. It only interacts with these

objects through their public methods, which also maintains encapsulation. Therefore, this class does not show content coupling.

Common coupling:

Reasoning for why not: The PointValueTime class does not rely on global variables or shared memory between multiple modules. Instead, all interactions happen through method parameters and object references. Therefore, this class does not show common coupling.

Control coupling:

Reasoning for why not: The PointValueTime class doesn't influence the execution flow of other modules by passing control switches. Its role is limited to storing and retrieving values, ensuring that it does not interfere with the logic of other classes. Therefore, this class does not show common coupling.

Data coupling:

Reasoning for why not: Rather than passing only the essential values such as doubles, booleans, or ints, the PointValueTime class sends entire MangoValue objects, even when only a portion of their data is required. This makes the class stamp coupling instead of pure data coupling.

c. The PointValueTime class exhibits high coupling. It is tightly coupled because it depends on multiple value types and frequently transfers unnecessary data, making the system less modular.

d. The PointValueTime class has an RFC score of 129, indicating high coupling. A high RFC score suggests that the class is deeply connected to other classes, making maintenance more difficult.

- **Lowest Coupling Classes:**

1. **com.serotonin.mango.db.DBConvert**
   a. The DBConvert class is a low-coupling class, with a Coupling Between Objects (CBO) value of 1, which indicates that it interacts with very few other classes. This class is built to handle database conversions by moving data between different database systems. It mainly works with DatabaseAccess objects for the source and target databases. Since DBConvert only focuses on database migration tasks and doesn't rely on a lot of external classes, it is not highly coupled, which makes it easier to modify. The coupling being measured here is

data coupling, which occurs when only the required data is passed between modules using simple and efficient parameters. The DBConvert class mainly interacts with database connections and runs queries without sending extra data. It interacts with DatabaseAccess objects through clear method calls, reducing its dependence on their internal logic or architecture. This makes the class very well structured.

b. Why it is not another type of Coupling:
Content coupling:

Reasoning for why not: The class doesn't change the internal structure of DatabaseAccess or any other classes. It simply uses their methods to get and insert data, ensuring that the operations remain separate.

Common coupling:

Reasoning for why not: The class doesn't depend on global variables or shared states from different modules. It functions independently by using database connections that are given directly.

Control coupling:

Reasoning for why not: Why DBConvert is Not Control Coupling: The class does not pass control switch arguments to affect the execution flow of another module. It just transfers data between databases without impacting any external logic.

Stamp coupling:

Reasoning for why not: Why DBConvert is Not Stamp Coupling: Rather than passing complete objects with extra data, the class uses specific parameters such as Connection objects and table names, which ensures minimal data transfer.

c. The DBConvert class shows low coupling. It is loosely coupled because it interacts with other modules through simple data exchange, making it easy to modify and maintain.

d. The DBConvert class has an RFC score of 1, indicating very low coupling. A low RFC score suggests that the class has minimal dependencies and follows good design principles.

2. com.serotonin.mango.db.MSSQLAccess

a. The MSSQLAccess class exemplifies low coupling, with a Coupling Between Objects (CBO) value of 2, which indicates that it interacts with only a small number of other classes. This class manages database connections and operations for Microsoft SQL Server within the Mango framework. It extends BasePooledAccess to inherit core database functionalities and mainly depends on ExtendedJdbcTemplate for executing SQL queries. Since this class is designed specifically for MSSQL databases with minimal dependencies, MSSQLAccess is not strongly coupled, making it easy to modify. The coupling being measured here is data coupling, which occurs when only the required data is passed between modules using simple and efficient parameters. The MSSQLAccess class interacts with the database by passing SQL commands and managing query results through method calls, which minimizes dependency on external classes.

b. Why it is not another type of Coupling:

Content coupling:

Reasoning for why not: The class doesn't change the internal structure of ExtendedJdbcTemplate or any other classes. It simply uses their methods to interact with the database, ensuring that the operations remain separate.

Common coupling:

Reasoning for why not: The class doesn't depend on global variables or shared states from different modules. It functions independently because all of the operations are contained within the class and are handled through method parameters.

Control coupling:

Reasoning for why not: The class does not pass control switch arguments to affect the execution flow of another module. It just processes SQL queries and manages exceptions without affecting external logic.

Stamp coupling:

Reasoning for why not: The class avoids passing unnecessary objects with extra data. Instead, it uses straightforward method parameters such as SQL queries and numeric values, ensuring that data exchange remains efficient.

c. The MSSQLAccess class shows low coupling. It is loosely coupled because it interacts with other modules only through clean, well-defined data exchanges.

d. The MSSQLAccess class has an RFC score of 2, indicating low coupling. A low RFC score suggests that the class has minimal dependencies, making it easier to maintain.

3. com.serotonin.mango.rt.dataSource.snmp.SnmpTrapRouter

a. The SnmpTrapRouter class has a low coupling, with a Coupling Between Objects (CBO) value of 2, which indicates that it interacts with very few other classes. Its primary role is to manage SNMP trap messages and route them to the appropriate SnmpDataSourceRT instances. The class primarily interacts with SnmpDataSourceRT and the org.snmp4j library, which helps to keep its dependencies to a minimum. This class is also not strongly coupled, which makes it easier to maintain. The coupling being measured here is data coupling, which occurs when only the required data is passed between modules using simple and efficient parameters. The SnmpTrapRouter class connects with SnmpDataSourceRT by sending SNMP messages and directing them according to their source addresses. This class only shares the necessary data, and also makes sure that it doesn't modify the internal structures of the objects it works with.

b. Why it is not another type of Coupling:

Content coupling:

Reasoning for why not: The class doesn't change the private data of SnmpDataSourceRT or PDU objects. Instead, it only uses their public methods to route messages.

Common coupling:

Reasoning for why not: The class doesn't depend on global variables or shared states from different modules. It only handles SNMP messages within its local scope and passes them to the correct data source instance.

Control coupling:

Reasoning for why not: The class does not pass control switch arguments to affect the execution flow of another module. It just routes SNMP messages without affecting the logic of SnmpDataSourceRT.

Stamp coupling:

Reasoning for why not: The class avoids passing unnecessary objects with extra data. Instead, it retrieves only the necessary data, such as sender addresses

and SNMP message content, which makes sure that data is transferred efficiently.

c. The SnmpTrapRouter class exhibits low coupling because it only exchanges necessary data, making it easy to maintain.
d. The SnmpTrapRouter class has an RFC score of 2, indicating low coupling and good modularity.

5. Analysing the difference:

1. Metrics: LCOM (Lack of Cohesion in Methods)

Aspect	**Highest Cohesion**	**Lowest Cohesion**
Class	BasePooledAccess	DataSourceEditDwr
Cohesion Type	Functional Cohesion: All methods are focused on a single task (managing database connections).	Logical Cohesion: Methods are grouped under a broad category (data source editing) but are not tightly related.
Value	2 (Low LCOM = High Cohesion): Methods share common fields and operate on the same data.	41 (High LCOM = Low Cohesion): Methods do not share common fields or operate on the same data.
Purpose	Single, well-defined purpose: managing database connections.	Broad, general purpose: editing and managing various types of data sources.
Design Quality	Well-organized and easy to maintain. Adheres to the Single Responsibility Principle.	Difficult to maintain and error-prone. Could benefit from refactoring to improve cohesion.
Field Usage	Most methods interact with the dataSource field, which is central to the class's functionality.	Only a few methods interact with the LOG field. Most methods do not share any fields.
Refactoring Need	No immediate need for refactoring. The class is already highly cohesive.	Strong need for refactoring to split the class into smaller, more cohesive components.

2. Metrics: WMC (Weighted Method Complexity)

Aspect	**Highest Cohesion**	**Lowest Cohesion**
Class	WatchListRowMaper	DataSourceEditDwr
Cohesion Type	Functional Cohesion	Logical Cohesion

Value	1 (Low LCOM)	241(High LCOM)
Purpose	Single, well-defined purpose	Broad, general purpose: editing and managing various types of data sources.
Design Quality	Well-organized and easy to maintain.	Difficult to maintain and error-prone. Could benefit from refactoring to improve cohesion.
Field Usage	Most methods interact with the dataSource field, which is central to the class's functionality.	The fields are not logically grouped or related to a single purpose
Refactoring Need	No immediate need for refactoring.	Strong need for refactoring to split the class into smaller, more cohesive components.

2. Metrics: RFC (Response for Class)

Aspect	**Highest Coupling**	**Lowest Coupling**
Class	DataSourceEditDwr	IdPointValueTime
Cohesion Type	Common Coupling	Logical Cohesion: Methods are grouped under a broad category (data source editing) but are not tightly related.
Value	RFC (Response For a Class) : 668	41 (High LCOM = Low Cohesion): Methods do not share common fields or operate on the same data.
Purpose	This class acts as a global pointaccess point	Broad, general purpose: editing and managing various types of data sources.
Design Quality	Difficult to maintain and error-prone	Difficult to maintain and error-prone. Could benefit from refactoring to improve cohesion.
Field Usage	Most methods do not share any fields.	Only a few methods interact with the LOG field. Most methods do not share any fields.
Refactoring Need	Heavy refactoring needed	Strong need for refactoring to split the class into smaller, more cohesive components.

3. Metrics: CBO (Coupling Between Objects)

Aspect	**Highest Coupling**	**Lowest Coupling**
Class	Common	DBConvert
Cohesion Type	Common Coupling: The class interacts with many global variables and shared resources, leading to strong dependencies across the system.	Data Coupling: The class exchanges only necessary data with other modules using clean, well-defined method parameters.
Value	194 (Very High COB = Strong Dependencies)	1 (Very Low COB = Minimal Dependencies)
Purpose	Provides shared functionality for various modules, such as session management, system settings, and utility functions.	Helps with database conversions, transferring data between different database systems without unnecessary dependencies.
Design Quality	Very interdependent and difficult to modify. Changes to shared resources could impact multiple modules, reducing maintainability.	Well-structured and easy to maintain. The class follows best practices by interacting only through method parameters, making it loosely coupled.
Field Usage	Many methods depend on global variables, system settings, and external utilities, making the class heavily interconnected.	Methods only use database connections and table names as parameters, avoiding reliance on external components.
Refactoring Need	Strong need for refactoring to reduce dependency on global state and improve modularity. Converting global variables into instance variables or dependency injections could help reduce coupling.	No immediate need for refactoring. The class is already well-designed with minimal dependencies.

	LCOM	**WMC**	**RFC**	**CBO**
Type	**Cohesion**	**Cohesion**	**Coupling**	**Coupling**
Highest Class	BasePooledAccess	WatchListRowMapper	DataSourceEditDwr	Common
Lowest Class	DataSourceEditDwr	DataSourceEditDwr	IdPointValueTime	DBConvert
Value	**2**	**41**	**3**	**194(highest), 1(lowest)**

Design Quality	Well-organized and easy to maintain. Adheres to the Single Responsibility Principle.	Difficult to maintain and error-prone. Could benefit from refactoring to improve cohesion.	Heavy reliance on the common class. Has maintainability challenges	Highly coupled and difficult to modify (Common), but DBConvert is well-structured with minimal dependencies

6. Github path:

https://github.com/gayatri48765/MangoTeam02/tree/main/Assignment2

7. Task performed by each team member:

a. Sakshi Tokekar: Software tool analysis (MetricsReloaded), Computing the metrics for each class, analysis of lowest and highest cohesion classes using cohesion metrics - LCOM, analysis of lowest coupling classes using coupling metrics – RFC, analyzing the differences

b. Asritha Korrapati - Computing the metrics for each class, analysis of lowest and highest coupling classes using coupling metric - CBO, analyzing the differences

c. Gayatri Rajesh Mangire- Software tool analysis - MetricsReloaded, Computing the metrics for each class, analysis of lowest and highest cohesion classes using cohesion metrics - WMC, analysis of highest coupling classes using– RFC

8. Contribution Table:

Gayatri Rajesh Mangire GRM210001	33%
Sakshi Tokekar SXT230143	34%
Asritha Korrapati ACK200002	33%