Code Analysis Report

Analyzed Code:

```
import pickle
print("1.Create a Class\n2.Mark Attendance\n3.Add Name and Detail of a Student\n4.Remove Name and Details of a student\n5. Check Attendance of a Particular Student")
w=int(input("What you want to do? 1, 2, 3, 4 or 5:"))
if w==1:
def create_class():
cn=input("Enter name of the class:")
nc=cn+".dat"
f=open(nc,'wb')
n=int(input("Enter no. of students in the class:"))
std={}
for i in range (1,n+1):
admnno=input("Enter Admn no:")
name=input("Enter name of the student:")
att={}
data=[name,att]
std[admnno]=data
pickle.dump(std,f)
f.close()

create_class()

elif w==2:
def mark_attendance():
cn=input("Enter name of the class:")
nc=cn+".dat"
f1=open(nc,'rb')
stud=pickle.load(f1)
ky=stud.keys()
f1.close()
f2=open(nc,'wb')
nstd={}
d=input("Enter date in the format DD-MM-YYYY:")
```

```
for j in ky:
lt=stud[j]
atd=lt[1]
print(lt[0],"Present or Absent")
a=input("Enter:")
atd[d]=a
lt[1]=atd
nstd[j]=lt
pickle.dump(nstd,f2)
f2.close()

mark_attendance()

elif w==3:
def add_a_student():
cn=input("Enter name of the class:")
nc=cn+".dat"
f3=open(nc,'rb')
stu=pickle.load(f3)
ks=stu.keys()
f3.close()
f4=open(nc,'wb')
adno=input("Enter Admn no of new student:")
nm=input("Enter name of the student:")
at={}
dat=[nm,at]
stu[adno]=dat
pickle.dump(stu,f4)
f4.close()

add_a_student()

elif w==4:
def remove_a_student():
cn=input("Enter name of the class:")
nc=cn+".dat"
f5=open(nc,'rb')
sdt=pickle.load(f5)
f5.close()
f6=open(nc,'wb')
```

```
ano=input("Enter admn no of the student to be removed:")
l=sdt[ano]
n=l[0]
print("Details of student to be removed:",ano,"&",n)
k=input("If wrong Enter 'x' to stop:")
if k=='x' or k=='X':
print("Start again")
exit

else:
sdt.pop(ano)
pickle.dump(sdt,f6)
f6.close

remove_a_student()

elif w==5:
def attendance_of_particular_student():
cn=input("Enter name of the class:")
nc=cn+".dat"
f7=open(nc,'rb')
stdt=pickle.load(f7)
f7.close()
ano=input("Enter admn no of the student to be searched:")
L=stdt[ano]
name=L[0]
at=L[1]
k=at.keys()
print("{:<10} {:<10}".format('DATE','ATTENDANCE'))
for x in k:
d=x
a=at[x]
print("{:<10} {:<10}".format(d,a))

attendance_of_particular_student()

def read():
cn=input("Enter name of the class:")
nc=cn+".dat"
f8=open(nc,'rb')
st = pickle.load(f8)
```

```
print("Content of the File ",cn,'\n')
k = st.keys()
for a in k:
l = st[a]
nm = l[0]
atd = l[1]
print("Admn No: ",a)
print("Name: ",nm)
print("Attendance \n",atd,'\n')
read()
```

Static Code Metrics:

Metric	Value
Lines of Code (LOC)	112
Comment Density	0.0
Cyclomatic Complexity	12
Maintainability Index	37.33386481348589
Halstead Metrics	
Unique Operators	3
Unique Operands	25
Total Operators	15
Total Operands	30
Volume	216.33097149259217
Effort	389.3957486866659
Depth of Inheritance Tree (DIT)	0
Coupling Between Object classes (CBO)	0
Lack of Cohesion of Methods (LCOM)	0
Fan-in	36
Fan-out	36
Number of Methods (NOM)	0

McCall's Intermediate Quality Metrics:

Metric	Value
Modifiability	0.42
Testability	0.57
Reliability	0.6
Understandability	0.27
Self-Descriptiveness	0.06
Reusability	1.0
Portability	0.85
Efficiency	0.73

GPT Analysis Report:

Code Quality Analysis and Suggestions for Improvement

1. Code Structure and Organization

- **Modularization**: The current implementation has all functionality put together in a linear fashion. Consider breaking each operation (create class, mark attendance, etc.) into distinct functions or even separate class methods.

- **Single Responsibility Principle**: Each function should have a single purpose. For example, the `mark_attendance` function does both reading from and writing to a file, which can be separated for better clarity and maintenance.

2. File Handling

- **Use of Context Managers**: Instead of manually opening and closing files with `open()` and `close()`, utilize context managers (`with` statement). This ensures files are properly closed even if an error occurs.

- **Existence Checking for Files**: Before loading a class file, check if the file exists to avoid errors. This can be done using `os.path.exists()`.

3. Error Handling

- Improve error handling, particularly for file operations. Use try-except blocks to catch exceptions related to file operations (like FileNotFoundError) and handle them gracefully.

- Provide feedback to users on invalid entries (e.g., if they input a non-existent student ID).

4. User Input Handling

- **Input Validation**: Always validate user inputs. The current code does not handle invalid data types or unexpected input, which can lead to runtime errors.

- **Better User Interactions**: Instead of using plain print statements, consider using formatted strings for clarity.

5. Code Duplication

- Notice that reading from the same file format happens multiple times across different functions. This logic can be abstracted into a single function that returns the student data, hence reducing code duplication and improving maintainability.

6. Documentation and Comments

- There's a lack of comments and documentation. In-line comments explaining non-obvious logic and a module or function-level docstring can greatly enhance understandability.

- Include a README file or appropriate documentation to describe how to use the program effectively.

7. Data Structures

- Consider using a more structured approach, such as creating a `Student` class to encapsulate student properties (name, attendance). This would also promote better data management practices.

8. Performance Optimization

- Using dictionaries is generally efficient, but for large class sizes, consider using a database for persistent storage and retrieval. This would also allow for more complex queries and better scalability.

9. Metrics Analysis

- **Maintainability Index**: The low maintainability index indicates that the code is hard to maintain. Refactoring to reduce complexity, improve modularity, and improve comments/documentation can help increase this score.

- **Cyclomatic Complexity**: Consider simplifying conditional logic. Instead of using many `elif` statements, a dictionary mapping user choices to functions can lead to cleaner code.

- **Comment Density**: With a comment density of 0, the code does not explain itself. Increasing this is critical for improving understandability.

Refactored Code Example

Below is an example refactor, particularly focusing on the `create_class` function:

```
python
import pickle
import os

class ClassManager:
def __init__(self, class_name):
self.class_name = class_name
self.file_name = f"{class_name}.dat"
self.students = self.load_students()

def load_students(self):
if os.path.exists(self.file_name):
with open(self.file_name, 'rb') as f:
return pickle.load(f)
return {}

def save_students(self):
with open(self.file_name, 'wb') as f:
pickle.dump(self.students, f)

def add_student(self, admission_no, name):
if admission_no not in self.students:
self.students[admission_no] = [name, {}]
self.save_students()
else:
print("Student already exists.")

def remove_student(self, admission_no):
if admission_no in self.students:
del self.students[admission_no]
self.save_students()
else:
print("No such student.")

def mark_attendance(self):
date = input("Enter date (DD-MM-YYYY): ")
for admission_no, (name, attendance) in self.students.items():
status = input(f"{name}, Present or Absent? ")
attendance[date] = status
self.save_students()

def display_students(self):
print("Class:", self.class_name)
for admission_no, (name, attendance) in self.students.items():
print(f"Adm No: {admission_no}, Name: {name}, Attendance: {attendance}")

def main():
print("1. Create a Class\n2. Mark Attendance\n3. Add Student\n4. Remove Student\n5. Display Students")
choice = int(input("Choose an option: "))
class_name = input("Enter class name: ")
manager = ClassManager(class_name)

if choice == 1:
num_students = int(input("Enter number of students: "))
for _ in range(num_students):
```

```
adm_no = input("Enter Admission No: ")
name = input("Enter Student Name: ")
manager.add_student(adm_no, name)
elif choice == 2:
manager.mark_attendance()
elif choice == 3:
adm_no = input("Enter Admission No: ")
name = input("Enter Student Name: ")
manager.add_student(adm_no, name)
elif choice == 4:
adm_no = input("Enter Admission No to remove: ")
manager.remove_student(adm_no)
elif choice == 5:
manager.display_students()
else:
print("Invalid choice.")

if __name__ == "__main__":
main()
```

Summary

Refactoring the code will significantly improve its readability, maintainability, and robustness. Implementing the suggested changes helps to address issues related to error handling, file I/O, code duplication, and adherence to best practices in software development. As a result, the program will be more user-friendly, and less prone to errors while also being easier to extend and maintain.