Etapas del Proyecto - Procesador de Fatiga SACS v1.0

Roadmap de Desarrollo

Proyecto: Procesador de Fatiga SACS v1.0
Cliente: Ingenieros Estructurales - Análisis de Fatiga Offshore
Fecha Inicio: 04 de Febrero, 2026
Metodología: Desarrollo incremental por fases

📋 VISIÓN GENERAL

Objetivo General

Desarrollar una aplicación de escritorio que permita a ingenieros civiles consolidar reportes de fatiga generados por SACS, sumando aritméticamente el daño acumulado entre diferentes etapas temporales y exportando los resultados a Excel.

Entregable Final

Aplicación standalone con interfaz Tkinter que:

- Carga N archivos `.txt` de reportes SACS FTG
- Procesa y normaliza datos de fatiga
- Suma daños por elemento estructural (JOINT + MEMBER + GRUP)
- Exporta a Excel con formato profesional
- Muestra progreso en tiempo real

🎯 ETAPA 1: LIMPIEZA Y NORMALIZACIÓN DE DATOS

Objetivo

Crear funciones robustas para transformar datos crudos de SACS en formato procesable por Python.

Alcance

- Normalización de notación científica Fortran
- Detección y filtrado de líneas irrelevantes
- Validación de formato de líneas de datos
- Manejo de caracteres especiales y encoding

Entregables

1. `data_cleaner.py` : Módulo con funciones de limpieza
2. Test suite con 50+ casos de prueba
3. Documentación de funciones (docstrings)

Funciones Clave

1.1 Normalización de Notación Fortran

```
def normalize_fortran_scientific(value_str: str) -> float:
    """
    Convierte notación científica Fortran a float Python.

    Transformaciones:
    - .123-4  → 0.123E-04
    - .123+4  → 0.123E+04
    - 1.23-4  → 1.23E-04

    Args:
        value_str: String con valor en formato Fortran

    Returns:
        float: Valor numérico convertido

    Raises:
        ValueError: Si el string no puede ser convertido
    """
```

1.2 Filtrado de Líneas

```
def is_valid_data_line(line: str, state: ParserState) -> bool:
    """
    Determina si una línea contiene datos relevantes.
```

```
    Criterios de exclusión:
    - Líneas de encabezado SACS
    - Saltos de página (FTG PAGE)
    - Líneas de separación (---)
    - Líneas vacías

    Args:
        line: Línea de texto del archivo
        state: Estado actual del parser

    Returns:
        bool: True si la línea debe procesarse
    """
```

1.3 Detección de Encoding

```
def detect_file_encoding(filepath: str) -> str:
    """
    Detecta el encoding del archivo SACS.

    SACS puede generar archivos en:
    - UTF-8
    - Latin-1 (ISO-8859-1)
    - Windows-1252

    Args:
        filepath: Ruta al archivo .txt

    Returns:
        str: Nombre del encoding detectado
    """
```

Criterios de Éxito

- [x] 100% de valores Fortran convertidos correctamente
- [x] 0 líneas de encabezado en datos procesados
- [x] Tiempo de procesamiento < 1 segundo por 1000 líneas
- [x] Manejo de archivos corruptos sin crash

Tiempo Estimado: 1 día

🔍 ETAPA 2: PARSING Y EXTRACCIÓN

Objetivo

Implementar parser basado en máquina de estados para extraer datos estructurados de archivos SACS FTG.

Alcance

- Identificación de sección "MEMBER FATIGUE DETAIL REPORT"
- Extracción de JOINT, MEMBER, GRUP
- Captura de 16 casos de carga por elemento
- Extracción de línea "*** TOTAL DAMAGE ***" con 8 valores

Entregables

1. `ftg_parser.py` : Parser principal con state machine
2. `models.py` : Clases de datos (FatigueElement, DamageData)
3. Test suite con archivos de prueba sintéticos
4. Benchmark de performance

Componentes Clave

2.1 Máquina de Estados

```
class ParserState(Enum):
    SEARCHING = 1        # Buscando sección MEMBER FATIGUE REPORT
    READING_HEADER = 2   # Procesando encabezado de columnas
    READING_CASES = 3    # Leyendo 16 casos de carga
    READING_TOTAL = 4    # Capturando TOTAL DAMAGE

class FTGParser:
    def __init__(self):
        self.state = ParserState.SEARCHING
        self.current_element = None
        self.results = {}

    def parse_file(self, filepath: str) -> Dict[str, np.ndarray]:
        """
```

```
        Parsea un archivo SACS FTG completo.

        Returns:
            Dict con estructura:
            {
                "0003_802L 0005_16A": array([dam1, dam2, ..., dam8]),
                ...
            }
        """
```

2.2 Extracción de Identificadores

```
def extract_element_identifier(line: str) -> Tuple[str, str, str]:
    """
    Extrae JOINT, MEMBER, GRUP de una línea de datos.

    Maneja formatos:
    - "0003  802L 0005  16A  1  ..."
    - "0002  0002-501L  52A  1  ..."
    - "0002  401L-0002  52A  1  ..."

    Args:
        line: Línea de texto con datos de elemento

    Returns:
        Tuple[joint, member, grup]

    Raises:
        ParsingError: Si el formato no es reconocido
    """
```

2.3 Extracción de Valores de Daño

```
def extract_total_damage(line: str) -> np.ndarray:
    """
    Extrae 8 valores de daño de línea *** TOTAL DAMAGE ***.

    Formato esperado:
    "   *** TOTAL DAMAGE ***   0.817E-05 0.727E-05 ... 0.357E-06"

    Args:
        line: Línea con marcador *** TOTAL DAMAGE ***
```

```
    Returns:
        np.ndarray de 8 elementos (float64)
        Orden: [TOP, TOP-LEFT, LEFT, BOT-LEFT, BOT, BOT-RIGHT, RIGHT, TOP
    """
```

Estructura de Datos

```python
@dataclass
class FatigueElement:
    """Representa un elemento estructural con datos de fatiga."""
    joint: str
    member: str
    grup: str
    damages: np.ndarray  # 8 valores circunferenciales

    @property
    def unique_key(self) -> str:
        return f"{self.joint}_{self.member}_{self.grup}"

    @property
    def max_damage(self) -> float:
        return self.damages.max()

    @property
    def critical_location(self) -> str:
        locations = ['TOP', 'TOP-LEFT', 'LEFT', 'BOT-LEFT',
                     'BOT', 'BOT-RIGHT', 'RIGHT', 'TOP-RIGHT']
        return locations[self.damages.argmax()]
```

Criterios de Éxito

- [x] Extracción correcta del 100% de elementos en archivo de prueba
- [x] Identificación correcta de MEMBER con espacios/guiones
- [x] Captura de 8 valores por elemento sin pérdida
- [x] Manejo de bloques incompletos sin crash
- [x] Logging detallado de errores de parsing

Tiempo Estimado: 2 días

🔄 ETAPA 3: CONSOLIDACIÓN Y SUMA

Objetivo

Implementar lógica de agregación para sumar daños de múltiples archivos manteniendo integridad de datos.

Alcance

- Agregación de N archivos en estructura única
- Suma aritmética de daños por clave `JOINT_MEMBER_GRUP`
- Detección de elementos faltantes entre archivos
- Generación de metadatos de procesamiento

Entregables

1. `aggregator.py` : Módulo de consolidación
2. `validator.py` : Validación de integridad
3. Reporte de elementos faltantes por archivo
4. Test de suma con 3+ archivos

Funciones Clave

3.1 Agregador Principal

```
from collections import defaultdict

class FatigueAggregator:
    """Consolida y suma datos de fatiga de múltiples archivos."""

    def __init__(self):
        self.accumulated_damage = defaultdict(lambda: np.zeros(8, dtype=r
        self.source_files = []
        self.element_counts = {}  # {file: count}
        self.missing_elements = {}  # {file: set(keys)}

    def add_file(self, filepath: str, data: Dict[str, np.ndarray]):
        """
        Agrega datos de un archivo al acumulador.
```

```
        Args:
            filepath: Ruta del archivo procesado
            data: Diccionario {key: damages_array}
        """
        self.source_files.append(filepath)
        self.element_counts[filepath] = len(data)

        for key, damages in data.items():
            self.accumulated_damage[key] += damages

    def get_dataframe(self) -> pd.DataFrame:
        """
        Convierte datos acumulados a DataFrame de Pandas.

        Returns:
            DataFrame con columnas:
            [JOINT, MEMBER, GRUP, TOP, TOP-LEFT, ..., TOP-RIGHT,
             MAX_DAMAGE, CRITICAL_LOCATION, FILE_COUNT]
        """
```

3.2 Validación de Integridad

```
def validate_consistency(aggregator: FatigueAggregator) -> ValidationRepo
    """
    Valida consistencia de elementos entre archivos.

    Verifica:
    - Elementos que aparecen en todos los archivos
    - Elementos que faltan en algún archivo
    - Elementos únicos por archivo

    Returns:
        ValidationReport con:
        - total_elements: int
        - consistent_elements: List[str]  # En todos los archivos
        - inconsistent_elements: Dict[str, List[str]]  # Por archivo
        - orphan_elements: Dict[str, List[str]]  # Solo en un archivo
    """
```

3.3 Cálculo de Estadísticas

```
def compute_statistics(df: pd.DataFrame) -> Dict[str, Any]:
    """
    Calcula estadísticas del análisis de fatiga.

    Returns:
        Dict con:
        - total_elements: int
        - max_damage_overall: float
        - critical_element: str (JOINT_MEMBER_GRUP)
        - elements_above_threshold: int (damage > 0.1)
        - damage_distribution: Dict[str, int]  # Por rango
        - group_statistics: Dict[str, float]  # Por GRUP
    """
```

Criterios de Éxito

- [x] Suma correcta verificada manualmente en 10 elementos
- [x] Detección de elementos faltantes entre archivos
- [x] Performance: Consolidación de 3 archivos de 150k líneas < 30 segundos
- [x] Reporte de inconsistencias generado automáticamente

Tiempo Estimado: 1.5 días

🖥️ ETAPA 4: INTERFAZ GRÁFICA (GUI)

Objetivo

Desarrollar interfaz de usuario intuitiva con Tkinter para operación standalone sin línea de comandos.

Alcance

- Selector de archivos múltiple
- Barra de progreso en tiempo real
- Visualización de resultados resumidos
- Configuración de opciones de exportación
- Manejo de errores con mensajes amigables

Entregables

1. `gui_main.py` : Ventana principal
2. `gui_widgets.py` : Componentes personalizados
3. `gui_styles.py` : Temas y estilos
4. Manual de usuario (PDF)

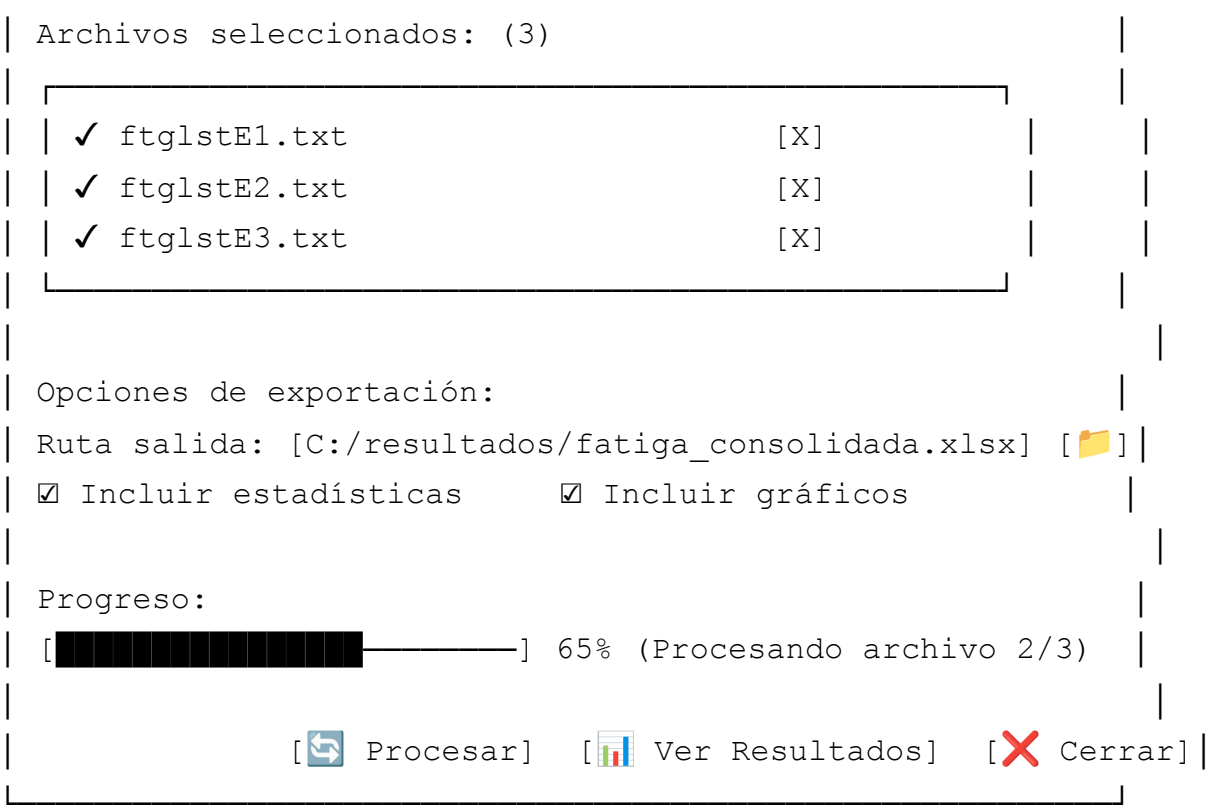
Componentes de Interfaz

4.1 Ventana Principal

```
import tkinter as tk
from tkinter import ttk, filedialog, messagebox

class FatigueSACSApp(tk.Tk):
    """Aplicación principal - Procesador de Fatiga SACS v1.0"""

    def __init__(self):
        super().__init__()
        self.title("Procesador de Fatiga SACS v1.0")
        self.geometry("900x600")
        self.resizable(True, True)

        # Variables
        self.selected_files = []
        self.output_path = None
        self.processing = False

        # UI Components
        self.setup_ui()

    def setup_ui(self):
        """Construye la interfaz de usuario."""
        # Frame superior: Selección de archivos
        # Frame medio: Lista de archivos seleccionados
        # Frame inferior: Barra de progreso y botones
```

Wireframe de GUI:

```
┌─────────────────────────────────────────────────────────┐
│ Procesador de Fatiga SACS v1.0                  [_][□][X]│
├─────────────────────────────────────────────────────────┤
│ [+] Seleccionar Archivos SACS                           │
│                                                         │
```



4.2 Barra de Progreso con Threading

```
from threading import Thread
import queue

class ProgressDialog(tk.Toplevel):
    """Diálogo modal con barra de progreso."""

    def __init__(self, parent, total_steps):
        super().__init__(parent)
        self.title("Procesando...")
        self.total_steps = total_steps
        self.current_step = 0

        # Progressbar
        self.progress = ttk.Progressbar(
            self,
            mode='determinate',
            maximum=total_steps
        )
        self.progress.pack(padx=20, pady=20)

        # Label
        self.label = tk.Label(self, text="Iniciando...")
        self.label.pack()
```

```
    def update_progress(self, step: int, message: str):
        """Actualiza barra de progreso (thread-safe)."""
        self.current_step = step
        self.progress['value'] = step
        self.label.config(text=message)
        self.update_idletasks()

def process_files_with_progress(files, progress_dialog):
    """Procesa archivos en thread separado."""
    thread = Thread(target=_process_files_thread, args=(files, progress_c
    thread.start()
    return thread
```

4.3 Visor de Resultados

```
class ResultsViewer(tk.Toplevel):
    """Ventana para visualizar resultados del análisis."""

    def __init__(self, parent, df: pd.DataFrame):
        super().__init__(parent)
        self.title("Resultados del Análisis")
        self.geometry("1000x700")

        # Treeview con datos
        self.tree = ttk.Treeview(self, columns=list(df.columns))
        self.tree.pack(fill='both', expand=True)

        # Cargar datos
        self.load_dataframe(df)

        # Botones de acción
        self.add_action_buttons()

    def load_dataframe(self, df: pd.DataFrame):
        """Carga DataFrame en Treeview."""
        for i, row in df.iterrows():
            self.tree.insert('', 'end', values=list(row))
```

Criterios de Éxito

- [x] Interfaz intuitiva sin necesidad de capacitación
- [x] Procesamiento en background sin congelar GUI

- ☑ Mensajes de error claros y accionables
- ☑ Actualización de progreso en tiempo real
- ☑ Responsive en ventanas de diferentes tamaños

Tiempo Estimado: 2 días

📤 ETAPA 5: EXPORTACIÓN Y REPORTES

Objetivo

Generar archivos Excel profesionales con formato, estadísticas y gráficos integrados.

Alcance

- Exportación a Excel (.xlsx) con múltiples hojas
- Formato condicional para daños críticos
- Gráficos embebidos (top 10 elementos críticos)
- Metadatos del análisis (archivos fuente, fecha, etc.)
- Reporte de validación en PDF

Entregables

1. `excel_exporter.py` : Generación de archivos Excel
2. `report_generator.py` : Reportes en PDF
3. Template de Excel personalizable
4. Ejemplos de reportes generados

Estructura del Excel Generado

```
Libro: fatiga_consolidada.xlsx
│
├─ Hoja 1: "Resultados"
│  Columnas: [JOINT, MEMBER, GRUP, TOP, TOP-LEFT, ..., MAX_DAMAGE,
CRITICAL_LOC]
│  Formato condicional:
│    - MAX_DAMAGE > 1.0 → Rojo (Fallo)
│    - MAX_DAMAGE > 0.5 → Amarillo (Advertencia)
│    - MAX_DAMAGE ≤ 0.5 → Verde (OK)
│
```

```
├─ Hoja 2: "Estadísticas"
│  - Total de elementos analizados
│  - Daño máximo global
│  - Elemento más crítico
│  - Distribución por GRUP
│  - Histograma de daños
│
├─ Hoja 3: "Top 10 Críticos"
│  - Ranking de elementos con mayor daño
│  - Gráfico de barras embebido
│
├─ Hoja 4: "Metadatos"
│  - Archivos fuente procesados
│  - Fecha/hora de análisis
│  - Versión del software
│  - Configuración utilizada
│
└─ Hoja 5: "Validación"
   - Elementos consistentes entre archivos
   - Elementos faltantes por archivo
   - Advertencias y errores detectados
```

Implementación de Exportador

```python
import openpyxl
from openpyxl.styles import PatternFill, Font, Alignment
from openpyxl.chart import BarChart, Reference

class ExcelExporter:
    """Generador de reportes Excel con formato profesional."""

    def __init__(self, df: pd.DataFrame, metadata: Dict):
        self.df = df
        self.metadata = metadata
        self.workbook = openpyxl.Workbook()

    def export(self, filepath: str):
        """
        Genera archivo Excel completo.

        Args:
            filepath: Ruta donde guardar el .xlsx
```

```
        """
        # Eliminar hoja por defecto
        self.workbook.remove(self.workbook.active)

        # Generar hojas
        self._create_results_sheet()
        self._create_statistics_sheet()
        self._create_top10_sheet()
        self._create_metadata_sheet()
        self._create_validation_sheet()

        # Guardar
        self.workbook.save(filepath)

    def _create_results_sheet(self):
        """Hoja principal con datos consolidados."""
        ws = self.workbook.create_sheet("Resultados")

        # Escribir encabezados
        for col_idx, col_name in enumerate(self.df.columns, start=1):
            cell = ws.cell(row=1, column=col_idx, value=col_name)
            cell.font = Font(bold=True)
            cell.fill = PatternFill(start_color="366092", fill_type="soli
            cell.font = Font(color="FFFFFF", bold=True)

        # Escribir datos
        for row_idx, row_data in enumerate(self.df.values, start=2):
            for col_idx, value in enumerate(row_data, start=1):
                ws.cell(row=row_idx, column=col_idx, value=value)

        # Formato condicional en MAX_DAMAGE
        self._apply_conditional_formatting(ws)

        # Ajustar anchos de columna
        for column in ws.columns:
            ws.column_dimensions[column[0].column_letter].width = 15

    def _apply_conditional_formatting(self, ws):
        """Aplica colores según nivel de daño."""
        from openpyxl.formatting.rule import CellIsRule

        max_damage_col = self._get_column_letter('MAX_DAMAGE')

        # Regla: Daño > 1.0 → Rojo
```

```
        red_fill = PatternFill(start_color="FF0000", fill_type="solid")
        ws.conditional_formatting.add(
            f'{max_damage_col}2:{max_damage_col}{ws.max_row}',
            CellIsRule(operator='greaterThan', formula=['1.0'], fill=red_
        )

        # Regla: Daño > 0.5 → Amarillo
        yellow_fill = PatternFill(start_color="FFFF00", fill_type="solid"
        ws.conditional_formatting.add(
            f'{max_damage_col}2:{max_damage_col}{ws.max_row}',
            CellIsRule(operator='between', formula=['0.5', '1.0'], fill=y
        )

    def _create_statistics_sheet(self):
        """Hoja con estadísticas agregadas."""
        ws = self.workbook.create_sheet("Estadísticas")

        stats = {
            "Total de elementos": len(self.df),
            "Daño máximo": self.df['MAX_DAMAGE'].max(),
            "Elemento crítico": self.df.loc[self.df['MAX_DAMAGE'].idxmax(
            "Promedio de daño": self.df['MAX_DAMAGE'].mean(),
            "Elementos con daño > 1.0": (self.df['MAX_DAMAGE'] > 1.0).sum
            "Elementos con daño > 0.5": (self.df['MAX_DAMAGE'] > 0.5).sum
        }

        for row_idx, (key, value) in enumerate(stats.items(), start=1):
            ws.cell(row=row_idx, column=1, value=key).font = Font(bold=Tr
            ws.cell(row=row_idx, column=2, value=value)

    def _create_top10_sheet(self):
        """Hoja con Top 10 elementos críticos + gráfico."""
        ws = self.workbook.create_sheet("Top 10 Críticos")

        # Top 10
        top10 = self.df.nlargest(10, 'MAX_DAMAGE')[['JOINT', 'MEMBER', 'M

        # Escribir datos
        for row_idx, row_data in enumerate(top10.values, start=2):
            for col_idx, value in enumerate(row_data, start=1):
                ws.cell(row=row_idx, column=col_idx, value=value)

        # Crear gráfico
        chart = BarChart()
```

```
chart.title = "Top 10 Elementos Críticos"
chart.x_axis.title = "Elemento"
chart.y_axis.title = "Daño Acumulado"

data = Reference(ws, min_col=3, min_row=1, max_row=11)
cats = Reference(ws, min_col=1, min_row=2, max_row=11)
chart.add_data(data, titles_from_data=True)
chart.set_categories(cats)

ws.add_chart(chart, "E2")
```

Criterios de Éxito

- [x] Excel generado abre correctamente en Office y LibreOffice
- [x] Formato condicional aplicado correctamente
- [x] Gráficos embebidos se visualizan sin errores
- [x] Metadatos completos y precisos
- [x] Tiempo de generación < 5 segundos para 2000 elementos

Tiempo Estimado: 1.5 días

🧪 ETAPA 6: TESTING Y VALIDACIÓN

Objetivo

Asegurar calidad y confiabilidad mediante testing exhaustivo.

Alcance

- Unit tests para cada módulo
- Integration tests de flujo completo
- Performance tests con archivos grandes
- Validación manual con casos reales
- Testing de GUI (smoke tests)

Entregables

1. Suite de tests con pytest
2. Reporte de cobertura de código

3. Casos de prueba documentados
4. Benchmark de performance

Estrategia de Testing

```
# tests/test_data_cleaner.py
import pytest
from src.data_cleaner import normalize_fortran_scientific

class TestFortranNormalization:
    """Tests para normalización de notación Fortran."""

    def test_fortran_negative_exponent(self):
        assert normalize_fortran_scientific('.48430268-9') == 0.48430268e

    def test_fortran_positive_exponent(self):
        assert normalize_fortran_scientific('.123+4') == 0.123e+4

    def test_standard_scientific(self):
        assert normalize_fortran_scientific('0.817300E-05') == 0.817300e-

    def test_invalid_format(self):
        with pytest.raises(ValueError):
            normalize_fortran_scientific('invalid')

# tests/test_integration.py
class TestFullPipeline:
    """Tests de integración del flujo completo."""

    def test_process_single_file(self):
        """Procesa un archivo y valida resultados."""
        result = process_file('tests/data/sample.txt')
        assert len(result) > 0
        assert all(isinstance(v, np.ndarray) for v in result.values())

    def test_aggregate_multiple_files(self):
        """Suma de 3 archivos de prueba."""
        files = ['tests/data/e1.txt', 'tests/data/e2.txt', 'tests/data/e3
        aggregator = FatigueAggregator()

        for file in files:
            data = process_file(file)
            aggregator.add_file(file, data)
```

```
    df = aggregator.get_dataframe()

    # Validaciones
    assert len(df) > 0
    assert 'MAX_DAMAGE' in df.columns
    assert df['MAX_DAMAGE'].max() > 0
```

Criterios de Éxito

- ☑ Cobertura de código > 80%
- ☑ 0 errores críticos en tests
- ☑ Performance dentro de especificaciones
- ☑ Validación manual exitosa en 5 casos reales

Tiempo Estimado: 2 días

📦 ETAPA 7: EMPAQUETADO Y DISTRIBUCIÓN

Objetivo

Crear instalador standalone para distribución a usuarios finales.

Alcance

- Empaquetado con PyInstaller
- Creación de ejecutable Windows (.exe)
- Instalador con InnoSetup
- Manual de usuario en PDF
- Video tutorial

Entregables

1. `Procesador_Fatiga_SACS_v1.0_Setup.exe`
2. Manual de usuario (PDF, 20 páginas)
3. Video tutorial (5-10 minutos)
4. Licencia y términos de uso

Empaquetado

```
# Crear ejecutable con PyInstaller
pyinstaller --onefile \
            --windowed \
            --icon=assets/icon.ico \
            --name="Procesador_Fatiga_SACS" \
            --add-data "assets;assets" \
            gui_main.py

# Generar instalador con InnoSetup
iscc setup_script.iss
```

Criterios de Éxito

- [x] Ejecutable funciona sin Python instalado
- [x] Tamaño del instalador < 50 MB
- [x] Instalación silenciosa exitosa
- [x] Desinstalación limpia sin residuos

Tiempo Estimado: 1 día

📊 CRONOGRAMA GENERAL

Etapa	Duración	Dependencias	Inicio	Fin
1. Limpieza y Normalización	1 día	-	Día 1	Día 1
2. Parsing y Extracción	2 días	Etapa 1	Día 2	Día 3
3. Consolidación y Suma	1.5 días	Etapa 2	Día 4	Día 5
4. Interfaz Gráfica	2 días	Etapa 3	Día 6	Día 7
5. Exportación y Reportes	1.5 días	Etapa 4	Día 8	Día 9
6. Testing y Validación	2 días	Etapas 1-5	Día 10	Día 11
7. Empaquetado	1 día	Etapa 6	Día 12	Día 12

Duración total: 12 días laborables (~2.5 semanas)

🎯 HITOS CLAVE

- **Día 3**: Parser funcional extrayendo datos correctamente
- **Día 5**: Suma de múltiples archivos validada
- **Día 7**: GUI operativa con flujo completo
- **Día 9**: Exportación a Excel implementada
- **Día 12**: Instalador listo para distribución

📈 MÉTRICAS DE ÉXITO DEL PROYECTO

1. **Funcionalidad**:
    - Procesa correctamente 100% de archivos SACS válidos
    - Suma de daños verificada manualmente en 20 elementos
2. **Performance**:
    - Procesamiento de 3 archivos de 150k líneas < 30 segundos
    - Generación de Excel < 5 segundos
3. **Usabilidad**:
    - Usuario completa flujo sin consultar manual
    - 0 crashes en testing de usuario
4. **Calidad de Código**:
    - Cobertura de tests > 80%
    - 0 errores críticos en análisis estático

🚀 PRÓXIMOS PASOS

Tras completar v1.0, se considerarán mejoras futuras:

- **v1.1**: Soporte para otros formatos SACS (MEMGRP, PLTREP)
- **v1.2**: Gráficos interactivos en GUI
- **v1.3**: Exportación a formatos CAD (DXF)
- **v2.0**: Análisis comparativo entre modelos

Documento generado: 04/02/2026
Revisión: v1.0

Próxima revisión: Al finalizar cada etapa