Exercise 4: Merge Conflicts and Resolution

Learning Objectives

By completing this exercise, you will practice:

- Understanding what causes merge conflicts
- Identifying and analyzing merge conflict markers
- Resolving different types of merge conflicts
- Using merge strategies and tools
- Preventing conflicts through good practices

Scenario

You're leading a data science team where multiple developers are working on the same analytics project. Conflicts are inevitable when team members modify the same files. This exercise will teach you to handle these conflicts professionally and efficiently.

Setup

Initialize the analytics project:

```
# Create project directory
mkdir DataScienceProject
cd DataScienceProject

# Initialize repository
git init
git config user.name "Your Name"
git config user.email "your.email@example.com"
git branch -M main
```

Part 1: Project Foundation and Initial Conflicts

Step 1: Create Base Project

Create the foundation files:

`project_config.py`:

```
# Data Science Project Configuration
import os

# Database connection settings
DATABASE_HOST = "localhost"
DATABASE_PORT = 5432
DATABASE_NAME = "analytics_db"
```

```
DATABASE_USER = "analyst"

# Analysis parameters
ANALYSIS_START_DATE = "2024-01-01"
ANALYSIS_END_DATE = "2024-12-31"
CONFIDENCE_LEVEL = 0.95

# Output settings
OUTPUT_FORMAT = "csv"
CHART_STYLE = "seaborn"
```

data_analysis.py:

```
# Main data analysis script
import pandas as pd
import numpy as np
from datetime import datetime

def load_customer_data():
    """Load customer data from database"""
    print("Loading customer data...")
    # Placeholder for database connection
    return pd.DataFrame()

def analyze_customer_trends():
    """Analyze customer behavior trends"""
    data = load_customer_data()
    print("Analyzing customer trends...")
    # Basic analysis placeholder
    return {"total_customers": 1000}

def generate_report():
    """Generate analysis report"""
    results = analyze_customer_trends()
    print(f"Analysis complete. Found {results['total_customers']} customers.")
    return results

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

README.md:

```
# Data Science Analytics Project

## Overview
This project analyzes customer behavior and generates insights for business
decisions.

## Team Members
```

```
- Lead Analyst: [Your Name]

## Current Status
- Project setup: Complete
- Data collection: In Progress
- Analysis: Planned

## Features
- Customer behavior analysis
- Trend identification
- Automated reporting
```

1. Create all files and commit: "Initial project setup with configuration and analysis framework"

Step 2: Simulate Developer A's Work

1. Create a branch for Developer A: `git switch -c feature/database-integration`

Modify `project_config.py` to add database credentials:

```
# Data Science Project Configuration
import os

# Database connection settings
DATABASE_HOST = "prod-analytics-server.company.com"
DATABASE_PORT = 5432
DATABASE_NAME = "analytics_db"
DATABASE_USER = "analyst"
DATABASE_PASSWORD = "secure_password_123"

# Analysis parameters
ANALYSIS_START_DATE = "2024-01-01"
ANALYSIS_END_DATE = "2024-12-31"
CONFIDENCE_LEVEL = 0.95

# Output settings
OUTPUT_FORMAT = "csv"
CHART_STYLE = "seaborn"
```

Modify `data_analysis.py` to add real database functionality:

```
# Main data analysis script
import pandas as pd
import numpy as np
from datetime import datetime
import psycopg2
import project_config as config

def load_customer_data():
```

```
    """Load customer data from database"""
    print("Connecting to production database...")
    conn = psycopg2.connect(
        host=config.DATABASE_HOST,
        port=config.DATABASE_PORT,
        database=config.DATABASE_NAME,
        user=config.DATABASE_USER,
        password=config.DATABASE_PASSWORD
    )

    query = """
    SELECT customer_id, registration_date, total_purchases, customer_segment
    FROM customers
    WHERE registration_date >= %s AND registration_date <= %s
    """

    data = pd.read_sql(query, conn, params=[config.ANALYSIS_START_DATE,
config.ANALYSIS_END_DATE])
    conn.close()
    return data

def analyze_customer_trends():
    """Analyze customer behavior trends"""
    data = load_customer_data()
    print("Analyzing customer trends...")

    # Advanced analysis
    segment_analysis = data.groupby('customer_segment').agg({
        'total_purchases': ['count', 'mean', 'sum'],
        'customer_id': 'count'
    }).round(2)

    return {
        "total_customers": len(data),
        "segments": segment_analysis.to_dict()
    }

def generate_report():
    """Generate analysis report"""
    results = analyze_customer_trends()
    print(f"Analysis complete. Found {results['total_customers']} customers.")
    print("Segment analysis completed.")
    return results

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

2. Commit Developer A's work: "Add production database integration and advanced analytics"

Step 3: Simulate Developer B's Work

1. Switch back to main: `git switch main`

2. Create a branch for Developer B: `git switch -c feature/visualization-upgrade`

Modify `project_config.py` to add visualization settings:

```
# Data Science Project Configuration
import os

# Database connection settings
DATABASE_HOST = "localhost"
DATABASE_PORT = 5432
DATABASE_NAME = "analytics_db"
DATABASE_USER = "analyst"

# Analysis parameters
ANALYSIS_START_DATE = "2024-01-01"
ANALYSIS_END_DATE = "2024-12-31"
CONFIDENCE_LEVEL = 0.95

# Output settings
OUTPUT_FORMAT = "json"  # Changed from csv to json
CHART_STYLE = "plotly"   # Changed from seaborn to plotly
CHART_WIDTH = 1200
CHART_HEIGHT = 800
COLOR_PALETTE = ["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728"]

# New visualization parameters
ENABLE_INTERACTIVE_CHARTS = True
SAVE_CHARTS_TO_FILE = True
CHART_OUTPUT_DIR = "./charts"
```

Modify `data_analysis.py` to add visualization features:

```
# Main data analysis script
import pandas as pd
import numpy as np
from datetime import datetime
import plotly.express as px
import plotly.graph_objects as go
import project_config as config

def load_customer_data():
    """Load customer data from database"""
    print("Loading customer data...")
    # Using sample data for visualization development
    np.random.seed(42)
    sample_data = pd.DataFrame({
        'customer_id': range(1, 1001),
        'registration_date': pd.date_range('2024-01-01', periods=1000, freq='D'),
        'total_purchases': np.random.lognormal(3, 1, 1000),
        'customer_segment': np.random.choice(['Premium', 'Standard', 'Basic'],
```

```
1000)
    })
    return sample_data

def create_visualizations(data):
    """Create interactive visualizations"""
    print("Creating interactive visualizations...")

    # Customer segment distribution
    fig1 = px.pie(data, names='customer_segment',
                  title='Customer Segment Distribution',
                  color_discrete_sequence=config.COLOR_PALETTE,
                  width=config.CHART_WIDTH,
                  height=config.CHART_HEIGHT)

    # Purchase amount distribution by segment
    fig2 = px.box(data, x='customer_segment', y='total_purchases',
                  title='Purchase Distribution by Segment',
                  color='customer_segment',
                  color_discrete_sequence=config.COLOR_PALETTE,
                  width=config.CHART_WIDTH,
                  height=config.CHART_HEIGHT)

    if config.SAVE_CHARTS_TO_FILE:
        os.makedirs(config.CHART_OUTPUT_DIR, exist_ok=True)
        fig1.write_html(f"{config.CHART_OUTPUT_DIR}/segment_distribution.html")
        fig2.write_html(f"{config.CHART_OUTPUT_DIR}/purchase_distribution.html")

    if config.ENABLE_INTERACTIVE_CHARTS:
        fig1.show()
        fig2.show()

    return fig1, fig2

def analyze_customer_trends():
    """Analyze customer behavior trends"""
    data = load_customer_data()
    print("Analyzing customer trends...")

    # Create visualizations
    charts = create_visualizations(data)

    # Basic analysis
    analysis_results = {
        "total_customers": len(data),
        "avg_purchases": data['total_purchases'].mean().round(2),
        "charts_created": len(charts)
    }

    return analysis_results

def generate_report():
    """Generate analysis report"""
    results = analyze_customer_trends()
```

```
    print(f"Analysis complete. Found {results['total_customers']} customers.")
    print(f"Average purchases: ${results['avg_purchases']}")
    print(f"Created {results['charts_created']} interactive charts.")
    return results

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

3. Commit Developer B's work: "Add interactive visualizations and enhanced reporting"

Part 2: Creating and Resolving Basic Conflicts

Step 1: Merge First Feature

1. Switch to main: `git switch main`
2. Merge Developer A's work: `git merge feature/database-integration`
3. This should merge cleanly since main hasn't changed

Step 2: Create Your First Merge Conflict

1. Try to merge Developer B's work: `git merge feature/visualization-upgrade`
2. You should get a merge conflict! Git will show something like:

```
Auto-merging data_analysis.py
CONFLICT (content): Merge conflict in data_analysis.py
Auto-merging project_config.py
CONFLICT (content): Merge conflict in project_config.py
Automatic merge failed; fix conflicts and then commit the result.
```

3. Check the status: `git status`

Step 3: Understand Conflict Markers

Open `project_config.py` in your text editor. You'll see conflict markers:

```
# Data Science Project Configuration
import os

# Database connection settings
DATABASE_HOST = "prod-analytics-server.company.com"
DATABASE_PORT = 5432
DATABASE_NAME = "analytics_db"
DATABASE_USER = "analyst"
<<<<<<< HEAD
DATABASE_PASSWORD = "secure_password_123"
=======
>>>>>>> feature/visualization-upgrade

# Analysis parameters
```

```
ANALYSIS_START_DATE = "2024-01-01"
ANALYSIS_END_DATE = "2024-12-31"
CONFIDENCE_LEVEL = 0.95

# Output settings
<<<<<<< HEAD
OUTPUT_FORMAT = "csv"
CHART_STYLE = "seaborn"
=======
OUTPUT_FORMAT = "json"  # Changed from csv to json
CHART_STYLE = "plotly"   # Changed from seaborn to plotly
CHART_WIDTH = 1200
CHART_HEIGHT = 800
COLOR_PALETTE = ["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728"]

# New visualization parameters
ENABLE_INTERACTIVE_CHARTS = True
SAVE_CHARTS_TO_FILE = True
CHART_OUTPUT_DIR = "./charts"
>>>>>>> feature/visualization-upgrade
```

Understanding the markers:

- `<<<<<<< HEAD` - Start of current branch (main) changes
- `=======` - Separator between conflicting changes
- `>>>>>>> feature/visualization-upgrade` - End of incoming branch changes

Step 4: Resolve Configuration Conflicts

Resolve `project_config.py` by combining the best of both versions:

```
# Data Science Project Configuration
import os

# Database connection settings
DATABASE_HOST = "prod-analytics-server.company.com"
DATABASE_PORT = 5432
DATABASE_NAME = "analytics_db"
DATABASE_USER = "analyst"
DATABASE_PASSWORD = "secure_password_123"

# Analysis parameters
ANALYSIS_START_DATE = "2024-01-01"
ANALYSIS_END_DATE = "2024-12-31"
CONFIDENCE_LEVEL = 0.95

# Output settings
OUTPUT_FORMAT = "json"  # Use JSON for better data structure
CHART_STYLE = "plotly"   # Use plotly for interactive charts
CHART_WIDTH = 1200
CHART_HEIGHT = 800
```

```
COLOR_PALETTE = ["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728"]

# Visualization parameters
ENABLE_INTERACTIVE_CHARTS = True
SAVE_CHARTS_TO_FILE = True
CHART_OUTPUT_DIR = "./charts"
```

Step 5: Resolve Code Conflicts

Now open `data_analysis.py`. This will have more complex conflicts. You need to combine:

- Database functionality from Developer A
- Visualization functionality from Developer B
- Make sure imports work for both features

Create a resolved version that combines both features:

```
# Main data analysis script
import pandas as pd
import numpy as np
from datetime import datetime
import os
import psycopg2
import plotly.express as px
import plotly.graph_objects as go
import project_config as config

def load_customer_data():
    """Load customer data from database"""
    print("Connecting to production database...")
    try:
        conn = psycopg2.connect(
            host=config.DATABASE_HOST,
            port=config.DATABASE_PORT,
            database=config.DATABASE_NAME,
            user=config.DATABASE_USER,
            password=config.DATABASE_PASSWORD
        )

        query = """
        SELECT customer_id, registration_date, total_purchases, customer_segment
        FROM customers
        WHERE registration_date >= %s AND registration_date <= %s
        """

        data = pd.read_sql(query, conn, params=[config.ANALYSIS_START_DATE,
config.ANALYSIS_END_DATE])
        conn.close()
        print(f"Loaded {len(data)} records from database")
        return data
```

```
    except Exception as e:
        print(f"Database connection failed: {e}")
        print("Using sample data for development...")
        # Fallback to sample data
        np.random.seed(42)
        sample_data = pd.DataFrame({
            'customer_id': range(1, 1001),
            'registration_date': pd.date_range('2024-01-01', periods=1000,
freq='D'),
            'total_purchases': np.random.lognormal(3, 1, 1000),
            'customer_segment': np.random.choice(['Premium', 'Standard', 'Basic'],
1000)
        })
        return sample_data

def create_visualizations(data):
    """Create interactive visualizations"""
    print("Creating interactive visualizations...")

    # Customer segment distribution
    fig1 = px.pie(data, names='customer_segment',
                  title='Customer Segment Distribution',
                  color_discrete_sequence=config.COLOR_PALETTE,
                  width=config.CHART_WIDTH,
                  height=config.CHART_HEIGHT)

    # Purchase amount distribution by segment
    fig2 = px.box(data, x='customer_segment', y='total_purchases',
                  title='Purchase Distribution by Segment',
                  color='customer_segment',
                  color_discrete_sequence=config.COLOR_PALETTE,
                  width=config.CHART_WIDTH,
                  height=config.CHART_HEIGHT)

    if config.SAVE_CHARTS_TO_FILE:
        os.makedirs(config.CHART_OUTPUT_DIR, exist_ok=True)
        fig1.write_html(f"{config.CHART_OUTPUT_DIR}/segment_distribution.html")
        fig2.write_html(f"{config.CHART_OUTPUT_DIR}/purchase_distribution.html")
        print("Charts saved to files")

    if config.ENABLE_INTERACTIVE_CHARTS:
        fig1.show()
        fig2.show()

    return fig1, fig2

def analyze_customer_trends():
    """Analyze customer behavior trends"""
    data = load_customer_data()
    print("Analyzing customer trends...")

    # Create visualizations
    charts = create_visualizations(data)
```

```
    # Advanced analysis combining both approaches
    segment_analysis = data.groupby('customer_segment').agg({
        'total_purchases': ['count', 'mean', 'sum'],
        'customer_id': 'count'
    }).round(2)

    analysis_results = {
        "total_customers": len(data),
        "segments": segment_analysis.to_dict(),
        "avg_purchases": data['total_purchases'].mean().round(2),
        "charts_created": len(charts)
    }

    return analysis_results

def generate_report():
    """Generate analysis report"""
    results = analyze_customer_trends()
    print(f"Analysis complete. Found {results['total_customers']} customers.")
    print(f"Average purchases: ${results['avg_purchases']}")
    print(f"Created {results['charts_created']} interactive charts.")
    print("Segment analysis completed.")
    return results

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

Step 6: Complete the Merge

1. Stage the resolved files: `git add project_config.py data_analysis.py`
2. Check that conflicts are resolved: `git status`
3. Complete the merge: `git commit -m "Merge visualization features with database integration"`
4. Verify the merge: `git log --oneline --graph`

Part 3: Advanced Conflict Scenarios

Step 1: Documentation Conflicts

1. Switch to a new branch: `git switch -c feature/documentation-update`

Modify `README.md` significantly:

```
# Data Science Analytics Project

## Overview
This project provides comprehensive customer behavior analysis with advanced
visualization capabilities and production database integration.

## Team Members
- Lead Analyst: [Your Name]
```

```
- Database Engineer: Developer A
- Visualization Specialist: Developer B

## Architecture
- **Database Layer**: PostgreSQL with secure connections
- **Analysis Engine**: Pandas + NumPy for data processing
- **Visualization**: Plotly for interactive charts
- **Configuration**: Centralized config management

## Current Status
- Project setup: Complete ☑
- Data collection: Complete ☑
- Database integration: Complete ☑
- Visualization system: Complete ☑
- Analysis pipeline: Complete ☑

## Features
- Customer behavior analysis with segment breakdown
- Interactive data visualizations
- Automated report generation
- Production database connectivity
- Error handling and fallback mechanisms

## Usage
```bash
python data_analysis.py
```

Output

- Analysis results in JSON format
- Interactive HTML charts in ./charts directory
- Console reporting with key metrics

```
2. Commit: "Major documentation update with architecture and usage details"

### Step 2: Create Simultaneous Documentation Changes
1. Switch back to main: `git checkout main`
2. Also modify `README.md` but differently:
```markdown
# Data Science Analytics Project

## Overview
This project analyzes customer behavior and generates insights for business
decisions.

## Team Members
- Lead Analyst: [Your Name]

## Current Status
- Project setup: Complete
```

```
- Data collection: Complete
- Analysis: Complete
- Visualizations: Complete

## Features
- Customer behavior analysis
- Trend identification
- Automated reporting
- Database integration
- Interactive charts

## Quick Start
1. Configure database settings in project_config.py
2. Run: `python data_analysis.py`
3. View charts in the ./charts directory

## Dependencies
- pandas
- numpy
- psycopg2
- plotly

## Recent Updates
- Added production database support
- Implemented interactive visualizations
- Enhanced error handling
```

3. Commit: "Update README with quick start guide and dependencies"

Step 3: Resolve Documentation Conflicts

1. Try to merge: `git merge feature/documentation-update`
2. You'll get conflicts in README.md
3. Resolve by creating a comprehensive version that includes:
    - The detailed architecture from the feature branch
    - The quick start guide from main
    - All team members and features
    - Both usage instructions

Create a resolved README.md:

```
# Data Science Analytics Project

## Overview
This project provides comprehensive customer behavior analysis with advanced
visualization capabilities and production database integration.

## Team Members
- Lead Analyst: [Your Name]
- Database Engineer: Developer A
- Visualization Specialist: Developer B
```

```
## Architecture
- **Database Layer**: PostgreSQL with secure connections
- **Analysis Engine**: Pandas + NumPy for data processing
- **Visualization**: Plotly for interactive charts
- **Configuration**: Centralized config management

## Current Status
- Project setup: Complete ☑
- Data collection: Complete ☑
- Database integration: Complete ☑
- Visualization system: Complete ☑
- Analysis pipeline: Complete ☑

## Features
- Customer behavior analysis with segment breakdown
- Interactive data visualizations
- Automated report generation
- Production database connectivity
- Error handling and fallback mechanisms

## Quick Start
1. Configure database settings in project_config.py
2. Run: `python data_analysis.py`
3. View charts in the ./charts directory

## Usage
```bash
python data_analysis.py
```

Dependencies

- pandas
- numpy
- psycopg2
- plotly

Output

- Analysis results in JSON format
- Interactive HTML charts in ./charts directory
- Console reporting with key metrics

Recent Updates

- Added production database support
- Implemented interactive visualizations
- Enhanced error handling
- Comprehensive documentation

```

```
4. Complete the merge: `git add README.md` and `git commit -m "Merge comprehensive
documentation update"`

Part 4: Preventing and Managing Conflicts

Step 1: Create a Requirements File Conflict
1. Create branch: `git checkout -b feature/dependency-management`
2. Create `requirements.txt`:
```

pandas==1.5.3 numpy==1.24.3 psycopg2-binary==2.9.6 plotly==5.14.1 jupyter==1.0.0 matplotlib==3.7.1 seaborn==0.12.2

```
3. Commit: "Add requirements.txt with specific versions"
4. Switch to main: `git checkout main`
5. Create a different `requirements.txt`:
```

pandas>=1.5.0 numpy>=1.20.0 psycopg2-binary>=2.9.0 plotly>=5.0.0 scikit-learn==1.2.2 scipy==1.10.1

```
6. Commit: "Add requirements with flexible versions and ML libraries"

### Step 2: Resolve Requirements Conflicts
1. Merge: `git merge feature/dependency-management`
2. Resolve the conflict by creating a comprehensive requirements file:
```

Data processing

pandas>=1.5.0,<2.0.0 numpy>=1.20.0,<2.0.0

Database

psycopg2-binary>=2.9.0,❤️.0.0

Visualization

plotly>=5.0.0,<6.0.0 matplotlib>=3.7.0,<4.0.0 seaborn>=0.12.0,<1.0.0

Development and analysis

jupyter>=1.0.0,<2.0.0 scikit-learn>=1.2.0,<2.0.0 scipy>=1.10.0,<2.0.0

```
3. Complete the merge with good practices

### Step 3: Practice Conflict Prevention
Create a `.gitignore` file to prevent future conflicts:
```

Python

pycache/ *.py[cod] *$py.class *.so .Python env/ venv/ .venv/

Charts and outputs

charts/ *.html *.json *.csv

IDE

.vscode/ .idea/ *.swp *.swo

OS

.DS_Store Thumbs.db

Project specific

config/local_config.py *.log

```
Commit: "Add comprehensive gitignore to prevent conflicts"

## Part 5: Conflict Resolution Strategies

### Challenge 1: Binary File Conflicts
1. Add a sample image file `logo.png` (create an empty file for simulation)
2. In one branch, replace it with a different "image"
3. In another branch, also replace it
4. Learn how Git handles binary conflicts (it doesn't merge them automatically)

### Challenge 2: Multiple File Conflicts
Create a scenario where several files conflict simultaneously:
1. Create branch with changes to config, analysis, and README
2. Create another branch with different changes to the same files
```

```
3. Practice resolving multiple conflicts systematically

### Challenge 3: Partial Merge Conflicts
Create conflicts where only part of a function conflicts:
1. Add different error handling to the same function in two branches
2. Resolve by keeping the best parts of both approaches

## Verification and Review

### Test Your Understanding
1. What do the conflict markers `<<<<<<<`, `=======`, and `>>>>>>>` represent?
2. How do you check which files have conflicts during a merge?
3. What's the difference between `git merge --abort` and resolving conflicts?
4. How can you preview what conflicts might occur before merging?

### Review Commands Used
- `git merge` - Combine branches
- `git status` - Check merge status and conflicts
- `git add` - Stage resolved files
- `git commit` - Complete the merge
- `git merge --abort` - Cancel a problematic merge
- `git diff` - See differences between branches

## Expected Outcomes
After completing this exercise, you should:
- Recognize merge conflict indicators and markers
- Systematically resolve different types of conflicts
- Understand when to keep changes from different branches
- Know how to combine features from multiple developers
- Be able to complete merge commits successfully
- Understand strategies for preventing conflicts

## Best Practices Learned
- Communicate with team members about overlapping work
- Pull latest changes before starting new features
- Keep feature branches focused and short-lived
- Test merged code thoroughly before pushing
- Use meaningful commit messages especially for merge commits
- Consider using merge tools for complex conflicts

**Note:** In professional environments, conflicts are often resolved through code
review processes and pair programming rather than individual resolution.
```