1. Identified Code Smells:

a. Magic Strings:

The string "player1" and "player2" are used in the wonPoint method without any clear explanation. Magic strings are better replaced with constants or enums for better maintainability.

```
// Replace magic strings with constants or enums
private static final String PLAYER_1 = "player1";
private static final String PLAYER_2 = "player2";

public void wonPoint(String playerName) {
  if (PLAYER_1.equals(playerName))
    m_score1 += 1;
  else
    m_score2 += 1;
}
```

b. Code Duplication:

The code within the loop in the getScore method repeats similar logic. Extracting this logic into a separate method can improve code maintainability.

```
// Extract repeated logic into a separate method
private String getScoreDescription(int tempScore) {
  switch (tempScore) {
    case 0:
      return "Love";
    case 1:
      return "Fifteen";
    case 2:
      return "Thirty";
    case 3:
      return "Forty";
    default:
      return "";
  }
}

public String getScore() {
    // ...
    else {
        for (int i = 1; i < 3; i++) {
            if (i == 1) tempScore = m_score1;
            else {
                score += "-";
                tempScore = m_score2;
            }
            score += getScoreDescription(tempScore);
        }
    }
    // ...
}
```

2. Qualities These Smells Impact:

- **Readability and Maintainability:** Magic strings and duplicated code make the code harder to read and maintain.
- **Scalability:** The current implementation may not easily scale if additional players are introduced. Using player identifiers directly in code limits flexibility.

3. Identified Design Smells:

a. Lack of Abstraction:

The code directly operates with player identifiers and scores, lacking proper abstraction. Introducing a Player class and a Score class could improve design.

```
// Introduce a Player class
public class Player {
    private String name;

    public Player(String name) {
        this.name = name;
    }

public String getName() {
    return name;
    }
}

// Introduce a Score class
public class Score {
    private int scoreValue;

    public Score(int scoreValue) {
        this.scoreValue = scoreValue;
    }

    public int getScoreValue() {
        return scoreValue;
    }
}

// Refactor TennisGame1 class to use Player and Score classes
public class TennisGame1 implements TennisGame {
    private Player player1;
    private Player player2;
    private Score score1;
    private Score score2;

    public TennisGame1(Player player1, Player player2) {
        this.player1 = player1;
        this.player2 = player2;
        this.score1 = new Score(0);
        this.score2 = new Score(0);
    }

    public void wonPoint(Player player) {
        // Update the score based on the player
    }

    // Other methods remain similar, adapting to the new structure
}
```

4. Suggested Refactoring:

- **Introduce Constants:** Replace magic strings with constants or enums.
- **Extract Methods:** Extract repeated logic into separate methods for better readability.
- **Introduce Abstraction:** Create classes to represent players and scores, improving the overall design.

5. Justification for Refactored Version:

The refactored version enhances readability, maintainability, and scalability. Introducing constants and extracting methods make the code more readable. The abstraction of players and scores allows for easier extension to handle more players in the future, providing a more flexible and maintainable design. The separation of concerns and the use of classes enhance code modularity and encapsulation.

6. Final Code:

```
public class TennisGame1 implements TennisGame {
        private static final String PLAYER_1 = "player1";
        private static final String PLAYER_2 = "player2";
        private Player player1;
        private Player player2;
        private Score score1;
        private Score score2;

        public TennisGame1(Player player1, Player player2) {
                this.player1 = player1;
                this.player2 = player2;
                this.score1 = new Score(0);
                this.score2 = new Score(0);
        }
        public void wonPoint(Player player) {
                if (player.equals(player1))
                        score1.incrementScore();
                else
                        score2.incrementScore();
        }

        public String getScore() {
                String score = "";
                if (score1.equals(score2)) {
                        score = handleEqualScores();
                } else if (score1.isAdvantage(score2)) {
```

```
                score = handleAdvantage();
            } else {
                score = handleNonEqualScores();
            }
            return score;
        }

        private String handleEqualScores() {
            switch (score1.getScoreValue()) {
                case 0:
                    return "Love-All";
                case 1:
                    return "Fifteen-All";
                case 2:
                    return "Thirty-All";
                default:
                    return "Deuce";
            }

        }

        private String handleAdvantage() {
            int minusResult = score1.getScoreValue() - score2.getScoreValue();
            if (minusResult == 1)
                return "Advantage " + player1.getName();
            else if (minusResult == -1)
                return "Advantage " + player2.getName();
            else if (minusResult >= 2)
            return "Win for " + player1.getName();
        else
            return "Win for " + player2.getName();
    }

    private String handleNonEqualScores() {
        String tempScoreDescription = "";
        for (int i = 1; i < 3; i++) {
            int tempScore = (i == 1) ? score1.getScoreValue() :
            score2.getScoreValue();
            tempScoreDescription += (i == 1) ? "" : "-";
            tempScoreDescription += getScoreDescription(tempScore);
        }
        return tempScoreDescription;
    }

    private String getScoreDescription(int tempScore) {
        switch (tempScore) {
            case 0:
                return "Love";
            case 1:
                return "Fifteen";
            case 2:
                return "Thirty";
            case 3:
                return "Forty";
            default:
                return "";
                }
            }
        }
```

```
public class Player {
	private String name;
	public Player(String name) {
		this.name = name;
	}

	public String getName() {
		return name;
	}
}

public class Score {
	private int scoreValue;
	public Score(int scoreValue) {
		this.scoreValue = scoreValue;
	}

	public int getScoreValue() {
		return scoreValue;
	}

	public void incrementScore() {
		scoreValue++;
	}

	public boolean isAdvantage(Score opponentScore) {
		return Math.abs(scoreValue - opponentScore.getScoreValue()) == 1;
	}
}
```