Here's a concise cheat sheet on anti-patterns and code smells for your exam, focusing on the Chain of Responsibility design pattern with included UML diagrams where necessary.

Anti-Patterns Overview

Anti-patterns are common solutions to problems that generate negative consequences. They are divided into:

- **Software Development Anti-Patterns**
- **Software Architecture Anti-Patterns**
- **Software Project Management Anti-Patterns**

The Blob (Software Development Anti-Pattern)

- **Problem**: A small number of classes monopolize processing, while the rest only provide data.
- **Solution (Refactoring)**: Move methods closer to the data they use, identify and reduce far-coupling, and restructure associations to balance responsibilities across classes.

Functional Decomposition (Software Development Anti-Pattern)

- **Problem**: Direct translation of procedural design into an OO language without adapting to OO principles.
- **Solution (Refactoring)**: Create a UML design model to better represent the system, combine related functionalities into cohesive classes, and ensure proper use of OO principles.

Stovepipe System (Software Architecture Anti-Pattern)

- **Problem**: Systems integrated in a point-to-point manner without scalability.
- **Solution (Refactoring)**: Introduce common interfaces for services, increase abstraction, and clarify roles to reduce the number of direct interfaces.

Swiss Army Knife (Software Architecture Anti-Pattern)

- **Problem**: Overly complex interfaces or standards trying to cater to all possible situations.
- **Solution (Refactoring)**: Use the Facade pattern to simplify access and break down functionalities into separate classes as needed.

Code Smells

Code smells are indicators of potential problems in code. Key smells include:

- **Bloaters**: Large classes or methods that are hard to work with.
- **Object-Orientation Abusers**: Incorrect use of OO principles.
- **Change Preventers**: Code that requires changes in many places to add new features.
- **Dispensables**: Unnecessary code that can be removed.
- **Couplers**: Excessive coupling between classes.

Chain of Responsibility Design Pattern

This pattern allows passing requests along a chain of handlers, where each handler decides whether to process the request or pass it along.

UML Diagram for Chain of Responsibility

The UML diagram illustrates how different handlers are linked in a chain, with each capable of processing a request or passing it to the next handler.

Java Implementation Snippet:

```java
abstract class Handler {
    protected Handler nextHandler;
    public abstract void handleRequest(String request);
    public void setNextHandler(Handler nextHandler) {
        this.nextHandler = nextHandler;
    }
}
```

Concrete handlers extend `Handler` and implement the `handleRequest` method to perform specific actions.

Refactoring Techniques:

- **Compose Methods**: Change how methods are composed to improve readability.
- **Move Features between Objects**: Restructure how functionality is divided for better clarity.
- **Organize Data**: Decouple classes for better portability.
- **Simplify Conditional Expressions**: For clearer program flow.
- **Method Calls Simplification**: Make method signatures easier to understand.

In Summary:
Anti-patterns and code smells highlight areas where code can be improved for better maintainability and understandability. Refactoring is a crucial process for addressing these issues, supported by various techniques and practices that enhance code quality over time.