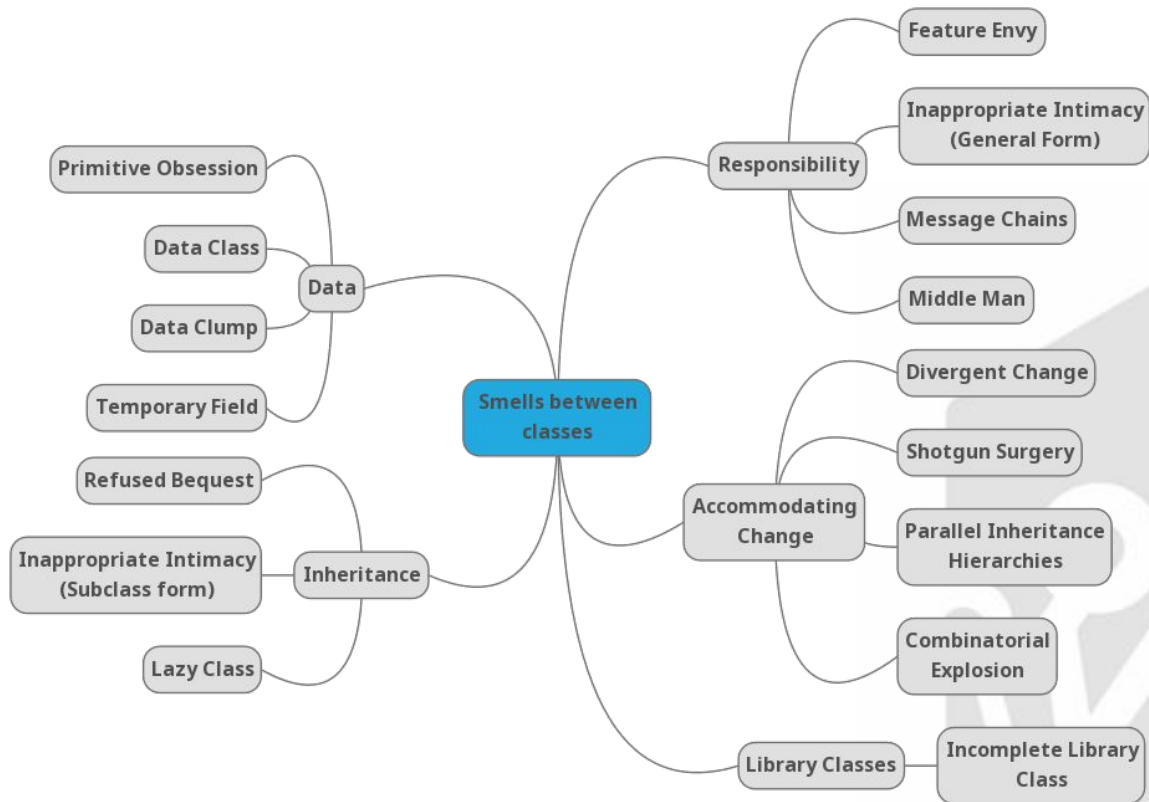


Code Smells between classes (I)



CODESAI

Smells between classes



Smells Between Classes

- **Responsibility Smells**
- Data Smells
- Inheritance Smells
- Accommodating Change Smells
- Library Classes smells



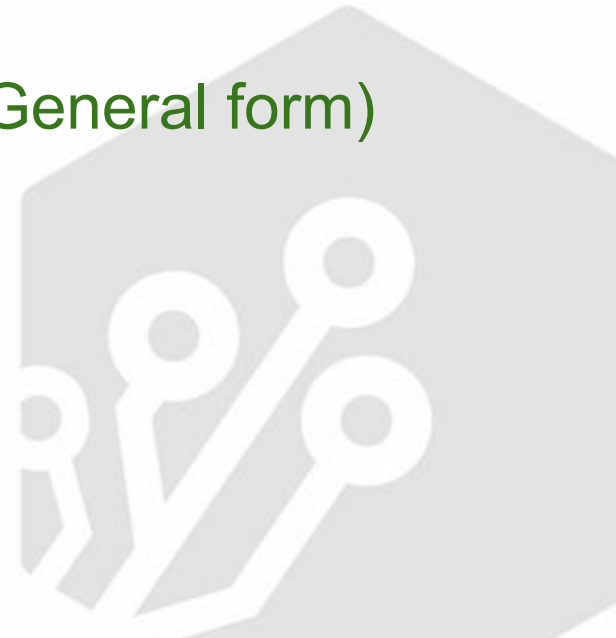
Responsibility Smells

- It's hard to assign the responsibilities to objects well.
- Refactoring allows us to experiment with different ideas in a way that safely lets us change our mind.
- We can use tools like **CRC cards** and **design patterns** to help us decide how our objects should work together.



Responsibility Smells


- Feature Envy
- Inappropriate Intimacy (General form)
- Message Chains
- Middle Man



Feature Envy (Coupler)

- A method seems to be focused on manipulating the data of other classes rather than its own (example). Very common among clients of **Data Classes**, but you can see it between any class and its clients.
- Refactoring tips: Choose the class where the responsibility should be and move it there, (may have to segregate it first).
- **Removal** 💰 :
 - Reduces coupling and duplication. Improves communication. May expose further refactoring opportunities.

Feature Envy (Coupler)

-  Several sophisticated patterns break the “**keeping behavior and data together**” heuristic: strategy, visitor,....
 - Data and its related behavior usually change together, but there are exceptions. In those cases, we move the behavior to keep changes in one place. We apply a more fundamental rule of thumb: “**put things together that change together**”.
 - Strategy and Visitor allow to change behavior easily, because they isolate the small amount of behavior that needs to be overridden, at the cost of further indirection. These patterns are used to combat the **Divergent Change** smell.

Inappropriate Intimacy (general form) (Coupler)

- One class **access internal (“should-be-private”) parts of another class, compromising the other class's encapsulation.**
- There are three possible problems:
 - Two tangled independent classes.
 - The tangled part seems to be a missing concept.
 - Classes that point to each other.



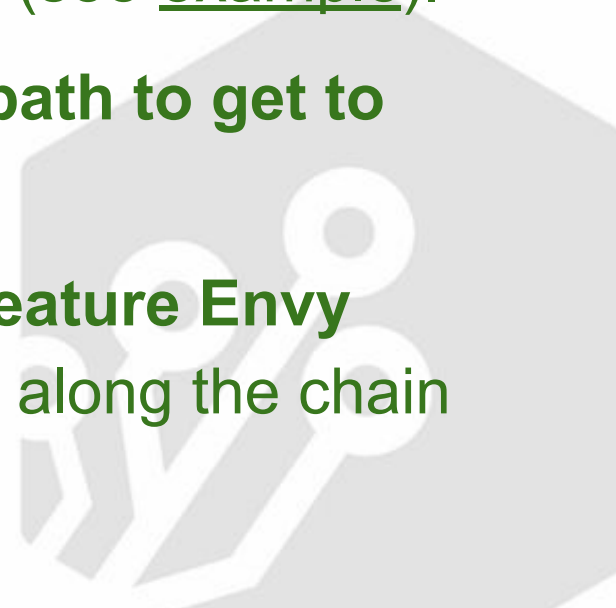
Inappropriate Intimacy (general form) (Coupler)

- Refactoring tip:
 - See the catalog description to choose the appropriate refactoring.
- **Removal** 💰 :
 - Reduces coupling and duplication. Improves communication. May reduce size.



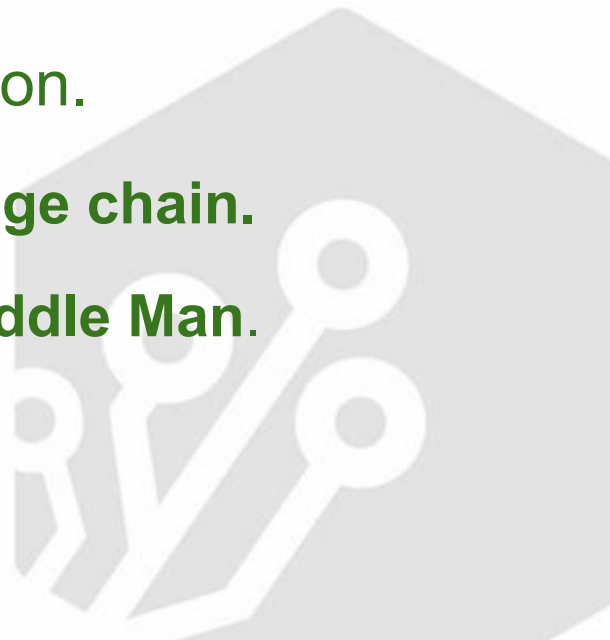
Message Chains (Coupler)

- Calls of the form `a.b() . c() . d()`, they may happen directly or through intermediate results (see example).
- It **couple**s both the objects and the path to get to them. Demeter's Law violation.
- Refactoring tips: If there is a case of **Feature Envy** remove it first, then use *Hide Delegate* along the chain to remove the knowledge of the path.



Message Chains (Coupler)

- Removal 💰 :
 - Reduces coupling.
 - May reduce or expose duplication.
- ⚠️ **A fluent interface is not a message chain.**
- ⚠️ **Trade-off: Message Chains vs Middle Man.**

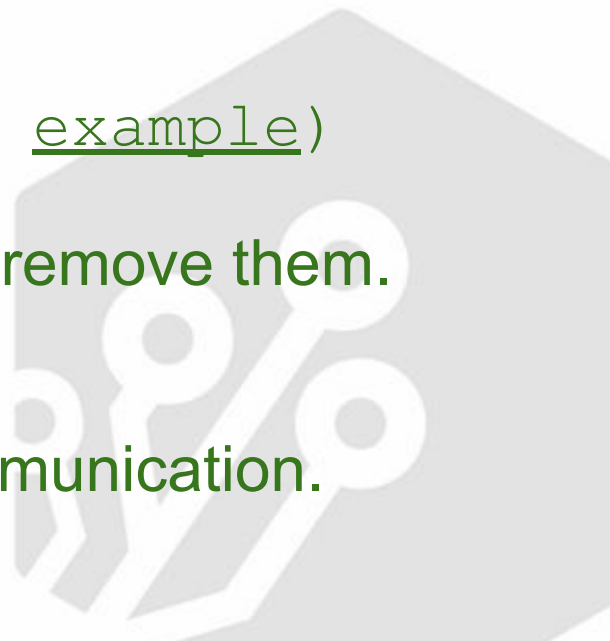


Middle Man (Coupler)




- Most methods of a class call the same or a similar method on another object:

```
f() {delegate.f();} (see example)
```

- Look for **shallow abstractions** and remove them.
- **Removal** 💰 :
 - Reduces size. May improve communication.



Middle Man (Coupler)

-  Some patterns (Proxy, Decorator, Façade, etc.) intentionally create delegates. Don't remove a **Middle Man** that's there for a reason.
-  Trade-off: **Message Chains** vs **Middle Man**.
-  **Middle Man** may provide a sort of **façade**, letting callers remain unaware of implementation details.

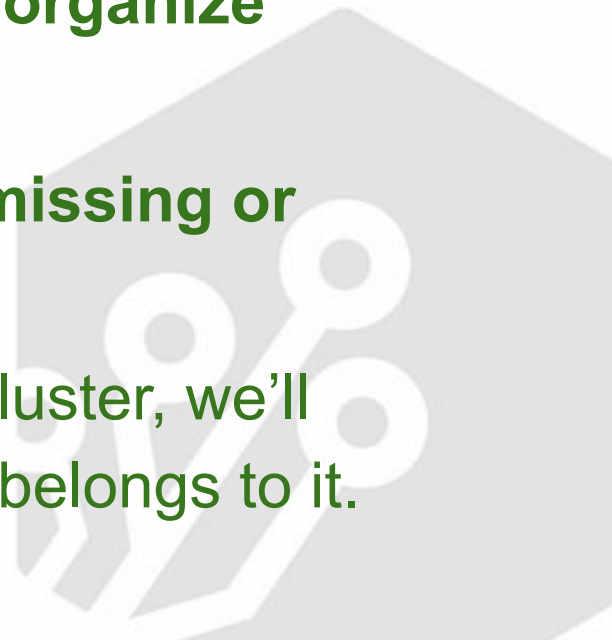
Smells Between Classes

- Responsibility Smells
- **Data Smells**
- Inheritance Smells
- Accommodating Change Smells
- Library Classes smells



Data Smells

- Objects are about **data and behavior together**.
- Your code will be more robust if you **organize objects by behavior**.
- These smells are often **a sign of a missing or inadequately formed class**.
- If the new class represents a good cluster, we'll usually be able to find behavior that belongs to it.



Data Smells

- Data Class
- Primitive Obsession
- Data Clump
- Temporary Field



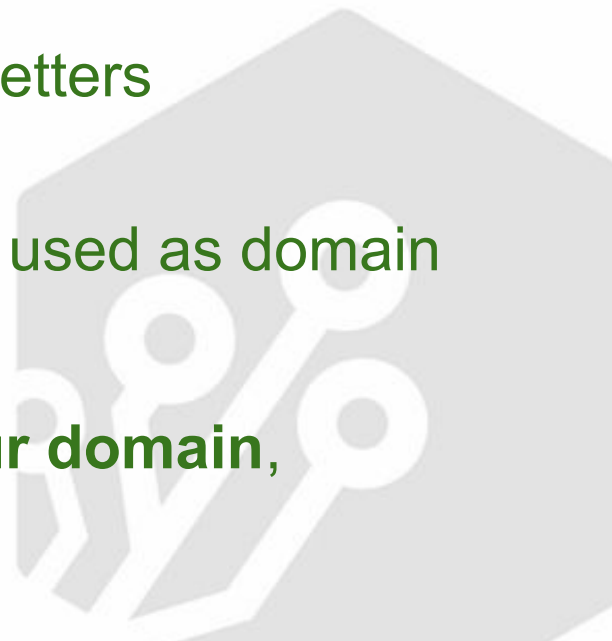
Data Class (Dispensable)

- Class consists only of public data members (or simple getters and setters).
- Clients **depend on its mutability** and **representation**
- To be **real objects** they need to **have behavior**.
- **Anemic domain model antipattern.**
- Refactoring tips: *Encapsulate data*, but even better look for cases of **Feature Envy** and fix them.



Data Class (Dispensable)

- **Removal** 💰: Improves communication. May expose duplication (in clients manipulation of data).
- ⚠️ Some infrastructure rely on getters & setters (persistence,...).
- ⚠️ **DTOs** are fine, as long as they are not used as domain objects.
- ⚠️ **Don't let infrastructure leak into your domain,** use wrappers or ACLs to avoid it.



Primitive Obsession (Bloater)

- Using primitive or near-primitive types to represent missing concepts.
- Refactoring tips:
 - **For missing objects:** introduce the missing abstraction. See also **Data Clump** (a kind of primitive obsession).
 - **Simulated inheritance:** use polymorphism



Primitive Obsession (Bloater)

- **Removal** 💰 :
 - Improves communication and flexibility.
 - May expose duplication.
 - Often exposes the need for other refactorings: the new concepts will attract behavior, look for cases of **Feature envy**.



Data Clump (Bloater)

- Groups of fields, variables or parameters that **only make sense when they are together** (example).
- A particular case of *Primitive Obsession*.
- It's also a signal of a **missing concept**.
- The new concepts will **attract behavior**
 - Look for **Feature Envy** cases.



Data Clump (Bloater)

- Refactoring tips:
 - Introduce the new concept, then look for cases of **Feature Envy** and fix them.
 - Take advantage of the new concept to alleviate cases of **Long Parameters List**.
- **Removal** 💰 :
 - Improves communication. May expose duplication. Usually reduces size.

Temporary Field

- A field is set only at certain times, and it's null, unused or initialized to something awkward at other times.
- Sometimes a signal of a **missing concept**.
- **Removal** 💰 : Improves communication and clarity. May reduce duplication, especially if other places can use the new class.
- ⚠️ **Not worthy if the new object is not a useful abstraction.**