Bloaters	Bloaters are code, methods and classes that have increased to huge proportions that makes it hard to work with. These bloaters accumulate over time.			
	Description	**Reason**	**Treatment**	**Payoff**
1. Long method	A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.	Something is always being added to a method, but nothing is ever taken out. Since it's easier to write code than to read it, this "smell" remains unnoticed until the method turns into an ugly, oversized beast. This is known as spaghetti code.	• Group similar code in a new method and replace the old code with a call to this new method. • Transform the long method into a separate class so that the original class can use this class as an object of the original method.	• Easier to maintain a shorter method as long method is tough to understand. • Short methods avoid duplicated code.
2. Large class	A class contains many fields/methods/lines of code.	Classes usually start small. But over time, they get bloated as the program grows. Something is always added into the same class, violates single responsibility, as the class is wearing too many (functional) hats.	• Create a new class and place the relevant fields and methods associated to its responsibility in it • Use interface / abstract class to utilize polymorphism if the code allows inheritance	• Avoid remember lots of attributes in a class (short and simple class) • Splitting classes and create methods avoid code duplication
3. Long parameter list	More than three or four parameters for a method.	A long list of parameters might happen after several types of algorithms are merged in a single method. It's hard to use as they grow longer. Instead of a long list of parameters, a method can use the data of its own object.	• Reduce the number of parameters and perform query call in the method • Parse a general/polymorphic object like superclass rather than the subclass to allow all the necessary data stored in the method itself.	• More readable, shorter code. • Refactoring may reveal previously unnoticed duplicate code.
4. Primitive obsession	Use of constant primitives, like USER_ADMIN_ROLE = 1 for referring to users with administrator rights. Storing everything in primitives (int, string, char)	Primitives usage makes validation difficult because the developer thinks in "short-term" by storing things without thinking about maintainability and extendibility.	• Replace data value with objects by creating a new class that stores the associated fields • Use objects as the parameters in a method rather than primitives.	• Code becomes more flexible due to objects rather than primitives. • Better code organization. • Easier to find duplicate code.
5. Data clumps	Different parts of the code contain identical group of variables (like different variables that represent the same information).	Often these data groups are due to poor program structure or "copy paste programming".	• Move the relevant fields into a new class to fulfill single responsibility. • Use objects as the parameters in a method rather than actual values.	• Improves understanding. • Better code organization. • Reduces code size.

Change Preventers	These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result.			
	Description	**Reason**	**Treatment**	**Payoff**
1. Divergent change	You find yourself having to change many unrelated methods when you make changes to a class. For example, when adding a new product type, you have to change the methods for finding, displaying, and ordering products.	Often these divergent modifications are due to poor program structure or "copy paste programming”.	• Create a new class and place the relevant fields and methods associated to its responsibility in it • Use interface / abstract class to utilize polymorphism if the code allows inheritance	• Improves code organization. • Reduces code duplication. • Simplifies support.
2. Shotgun surgery	Making any modifications requires that you make many small changes to many different classes.	A single responsibility has been split up among a large number of classes.	• Move relevant methods and fields into a new class to fulfill single responsibility • Use interface / abstract class to utilize polymorphism if the code allows inheritance so that code	• Better organization. • Less code duplication. • Easier maintenance.

<u>Couplers</u>	All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation. Coupling means there is a tight effect between classes.			
	<u>Description</u>	**<u>Reason</u>**	**<u>Treatment</u>**	**<u>Payoff</u>**
1. Feature envy	A method accesses the data of another object more than its own data.	This smell may occur after fields are moved to a data class. If this is the case, you may want to move the operations on data to this class as well.	• Move the method to a place where it is supposed to call	• Less code duplication (if the data handling code is put in a central place). • Better code organization (methods for handling data are next to the actual data).
2. Message chains	A series of calls like getX().getY().getZ().	A message chain occurs when a client requests another object, that object requests yet another one, and so on. The client is dependent on navigation along the class structure. Any changes in these relationships require modifying the client.	• Extract the method of the functionality and move the method into the beginning of the chain	• Reduces dependencies between classes of a chain. • Reduces the amount of bloated code.
3. Middle man	If a class performs only one action, delegating work to another class, why does it exist at all?	This smell can be the result of overzealous elimination of *Message Chains*. The class remains as an empty shell that doesn't do anything other than delegate	• Delete the middle man and let the client to call the end methods directly	• Less bulky code.

Dispensable	A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.			
	Description	**Reason**	**Treatment**	**Payoff**
1. Duplicate code	Two code fragments look almost identical.	Multiple programmers ae working on the same program at the same time. Unaware that there may be a subtle duplication as to meet the deadlines by using "copy paste programming".	• Group similar code in a new method and replace the old code with a call to this new method. • Use interface / abstract class to utilize polymorphism if the code allows inheritance (superclass that maintains the general functionality) • If one class does the work of two, split them by creating a new class and place the relevant fields and methods in it, adhering single responsibility.	• Merging duplicate code simplifies the structure of your code and makes it shorter. • Makes code simple and easier to support in the future.
2. Lazy class	A class that doesn't interact much in the program.	Maybe the class was designed to function after refactoring or to support future development but never got done before.	• If a subclass does the same as the superclass, merge both classes into one class to reduce complexity and navigability. • Delete unwanted class.	• Reduced code size. • Easier maintenance.
3. Data class	A class that contains only fields and crude methods for accessing them (getters and setters). It doesn't have any implementation logic.	Might be trying to create data quickly without thinking further about utilizing the existing classes to store those relevant data.	• Move the relevant fields into existing classes (if possible)	• Improves understanding • Improves code organization because operations are now gathered in single place. • Easily detect duplicate code.
4. Speculative generality	There's an unused class, method, field or parameter.	Code is created "just in case" to support for every anticipated case in the future. As a result, it becomes hard to understand and maintain.	• Remove unused fields • Remove unused parameters • Remove unused methods. • If there is unwanted superclass that is only utilized by one subclass, merge the superclass with the subclass	• Slimmer code. • Easier support.