Deliverable Code Smells

A code smell was described in the article as an example of an attribute in a code base that could be indicative of a deeper problem. Wikipedia gave several examples for this:

- Duplicated code: identical or very similar code exists in more than one location.
- Long method: a method, function, or procedure that has grown too large.
- Large class: a class that has grown too large. See God object.
- Too many parameters: a long list of parameters is hard to read, and makes calling and testing the function complicated. It may indicate that the purpose of the function is ill-conceived and that the code should be refactored so responsibility is assigned in a more clean-cut way.
- Feature envy: a class that uses methods of another class excessively.
- Inappropriate intimacy: a class that has dependencies on implementation details of another class.
- Refused bequest: a class that overrides a method of a base class in such a way that the contract of the base class is not honored by the derived class. See Liskov substitution principle.
- Lazy class / Freeloader: a class that does too little.
- Contrived complexity: forced usage of overly complicated design patterns where simpler design would suffice.
- Excessively long identifiers: in particular, the use of naming conventions to provide disambiguation that should be implicit in the software architecture.
- Excessively short identifiers: the name of a variable should reflect its function unless the function is obvious.
- Excessive use of literals: these should be coded as named constants, to improve readability and to avoid programming errors. Additionally, literals can and should be externalized into resource files/scripts where possible, to facilitate localization of software if it is intended to be deployed in different regions.
- Cyclomatic complexity: too many branches or loops; this may indicate a function needs to be broken up into smaller functions, or that it has potential for simplification.
- Downcasting: a type cast which breaks the abstraction model; the abstraction may have to be refactored or eliminated.

In looking for code smells in my own code, I especially focused on having long methods which try to do too much, and simplify too little. The only real functional method (that does not read console output) of multiple lines is my load_data method which needs to load files. It is as follows:

```
def load_data(filename) #Reads all rating data
open(filename).each_line do |line| #iterates over all lines in a file
splitLine = line.split("\t") #splits the line
user_id, item_id, rating = splitLine.shift.to_i, splitLine.shift.to_i, splitLine.shift.to_f
@person_movies_rating[user_id] ||= Hash.new() #sets a variable if nill
@person_movies_rating[user_id][item_id] = rating #sets the persons rating of the movie
@movie_rating[item_id] ||= [0,0] #if movie rating hasn't been intalized, make a new sum / # of ra
@movie_rating[item_id] = [@movie_rating[item_id][0] + rating, @movie_rating[item_id][1]+1]
#increment its value
end
end
```

The biggest code smells in this one is the lack of tendency to use an Object Oriented approach in reading in the file input. Rather than have discrete objects for each movie. Good object oriented design for this project would most likely involve reading each movie in the hash into a "movie" class, which in turn has behavior (such as rating, total amount of ratings, etc.). Instead, a movie rating and total amount of ratings is stored as a pair - a major anti-pattern to the OOP goal of Ruby. There is a reason for this - performance. Storing this behavior as a list of two elements rather than a whole class is much more effective and simple than making a whole class whose behavior is to store two numbers. However, to a programmer trying to decipher this, it might at first seem a bit cryptic... So that's code smell #1.

The second major code smell might be the excessive use of Ruby shorthands, most of which are aimed at reducing LOC count - to an excessive degree. For example, the shorthand `||=` means "if it is Nill, make it equal to this". To an experienced Ruby developer this might be obvious, but in detecting code smell one must consider the environment in which the code is written. Ultimately - programming languages are meant to be read by people. If the code is not easily understood, then it would smell much worse than code that is a tad more verbose but whose behavior is much more obvious. In my example, the `||=` could be replaced by an if else statement which would accomplish the same thing. In four times the LOC, I could do something that is easier to understand for people less familiar with Ruby shorthands.