Jesse Keller
CS 372 Final

Part A I implemented chain of responsibility. I had trouble thinking of an originalish example of this pattern so I made a wizard class. If you were a powerful wizard, the spell you wanted to cast would be casted. If not, something would still happen, most likely a less powerful version of the intended spell would be cast. See code for details.

Part B Here is a list of design patterns and what they do...

Interpreter	Interprets code based on some abstract syntax tree. Useful when constructing new languages.
Composite	If your object can be represented as a tree and certain functions depend upon the values of elements lower in the tree, this may be the design pattern for you. For instance, a file system may use the composite design.
Visitor	Allows you to run a function on various members of a given type without needing to add functions to the members' classes directly.
Template	Its possible that how a particular class behaves may depend upon the type of info that has been passed to it. This is when it is useful to use the template design pattern.
Observer	Think events. Observers allow what one class does to depend upon what happens to another class. You could for instance count the number of times a particular objects has been interacted with, or the number of times a function has been called.
Decorator	Allows functionality to be extended without modifying the base class directly. For instance, the most common example is the window class which may be bordered, or transparent. Rather than modify the already existing window class which may mess up everything.
Command	Move function calls into a class. Useful for creating macros and other types of user defined behavior.
Adapter	Acts as a bridge to change one type of interface into another type (thing cord adapter or something like that).
Facade	A high level, easy to understand interface that handles a lot of low level, difficult to understand stuff. Useful for designing GUIs since buttons presses are essentially commands that do stuff.
State	A state-machine. If your class can be made to look like a network flow diagram, this is probably the design for you.

Part C: Below is the acronym for SOLID.

S: Single responsibility principle. A class should only be responsible for one thing. If you need to redesign your interface, you shouldn't have to rewrite entire portions of your project, just one or two parts of it.

O: Open/Closed Principle. Classes should be open to extension, closed to modification. Unless your class just doesn't do what it claims to do, if you want to add more functionality, create a new modifying class, don't modify the original class.

L: Liskov Substitution Principle. Classes shouldn't be children of other classes unless the parent and children can be replaced with each other in every instance.

I: Interface Segregation Principle. Unrelated functionality should be segregated. For example, the user shouldn't have to know how all of SFML works in order to draw stuff. The sound and networking functions should be separate.

D: Dependency Inversion Principle. Parents should depend on their children.