I am not a dot net programmer but an ms-access programmer and am looking for design patterns that help me make the most of what ms-access does well. I feel like I am about half way to a decent pattern. Ms-access may not be able to support a dot net design pattern and still be easy to develop so I am happy to take what I can where I can.   
  
This is what I have worked out so far:

* Try to keep generic functions in re-usable modules that can be used in any ms-project, just by cut and paste. These modules must not have any hardwired internal references to named objects such as forms, reports and especially fields and tables.
* Each module generally has from about 3 to 15 procedures/functions. Any more and you should probably break the module up. The reason is that when you paste the module into a new project, you need to be able to quickly scan the module to make sure all the code is compatible with the new project.
* Modules that have hardwired references to named database objects should also be kept small as possible. For example the module that references application settings should be as small as possible. This facilitates copying the module to another project and changing the object references.
* I am finding that it is useful to have modules that map directly to tables, for example I have tblSettings and modTblSettings. This is conceptually very simple. I can type something like Browser = modTblSettings.DefaultBrowser. The module takes care of checking that the settings table only has exactly one data row and that a default browser has been entered into this row, and that it exists on the current system. Obviously checking that the browser.exe file exists is handled by a call to another module to avoid code duplication for common tasks. Note that typing in the module name triggers intellisense which lists the modules procedures and functions.
* The module per table method seems to result in some code duplication compared with having generic procedures that can work on any table, however the code in these modules is simple and I do not have to spend much time ensuring that I do not upset other existing algorithms that use generic functions. This is important for me as unit testing databases that are open systems is a nightmare. My current project is for organising photos, so involves the filing system, internal photo metadata and database data. People can delete photos or edit photo metadata with other programs, hence I am working with an open system. Only 15% of my code is easily isolated from the external environment and can be unit tested easily. The rest of the code shifts files around or changes metadata and would create more work than I can do to test properly.
* I do have a version of polymorphism. In the function definition, I just add a parameter to say which algorithm to run within the procedure. This parameter is usually an enum as enums provide an intellisense list of acceptable values for the parameter. For example, I might have an enum for GPS coordinate format. GPS coordinates can be in degrees, minutes and seconds or decimalised degrees. I could pass the X and Y coordinates to a function that generates a one kilometre square box around the point and an enum that tells the function what format the input coordinates are in.
* Call stacks are provided by a the pushstack/popstack code sample from FMS inc.'s website. It works but it annoys me. My error handler writes error info to tblErrLog including parameters and the call stack. The error handling procedure records the error variables to static internal variables so that they can be compared to the next incoming error. If the next incoming error is identical, then that usually indicates that the error is in a loop and the application only records the first two errors, apologises to the user then quits. I don't think you can recover from unhandled errors in loops.
* Integration is a major issue as many different workflows are possible with my applications. In dot net, developers often lock down user choices by using lots of simple forms. In some domains, this can create a painful user experience, similar to the fabled blind men trying to describe an elephant. My forms often include data from several related tables and can provide views that slice down through relational hierarchies. Real world example, what are the environmental issues associated with a particular tower on a particular transmission line on a particular property with particular landholder(s). My code includes a lot of checking the status of its world. Someone may enter all the towers before they enter the properties. This means I have to elegantly detect that the stack of data is incomplete, tell the user and cancel the operation if showing incomplete data could result in consequences. Currently, I rely on the contract by design pattern, where the contract is confirmed explicitly whenever code runs. Unlike unit testing that tells the developer if there is an issue, with this pattern the user finds the issue and gets a good error message that tells him what went wrong. Obviously you are assuming the user will contact you if the error is a design error. The general form of the code looks like this:

**CODE**

Set P = new photocollection  
if not P.ViewPhotos(folder) then   
msgbox P.ErrMess  
End if

Inside the view photos module is a heavy validation section that generates error messages if anything is wrong (folder does not exist, don't have permissions, folder is empty). The view photos method is public function that returns a boolean success flag. The idea of contract by design also works if you are using procedural code, you just need to return the function result as a user defined type that includes field for a success flag, a return value and an error message. All messages are displayed via the calling form and are not directly raised in business modules. This idea comes from the ms-Access Bible, as the business layer may eventually migrate to another machine and msgboxes on this machine would be invisible to the user and would just freeze the users screen.

* Forms handle all user interface logic. They call procedures in modules for anything that fetches or manipulates data. Usually a button click or combobox selection will call a private procedure in the forms module that reads the state of various controls then creates SQL statements and info that is sent to the business layer modules for processing. This is sort of a halfway house between the n-tier design pattern and the thick-client server pattern that is what ms-access really is. Later, I could create modules that shadow forms, eg. modFrmSettings and this would facilitate unit testing. The general principle is that anything that I prefer to capture events with the forms code module. Processing of the events can occur elsewhere. If you compromise speed of development, then you might as well use some other development environment. Ms-access is great for trying to find a way through the jungle of really complex real world issues. Once I have found a good solid data model, then I will attempt to jump to C#, WPF etc (so far unsuccessfully). So my approach is to try to pave the way for a design pattern by partially implementing it from the start. Many ms-access applications are in-house applications where I am the main or only user or are disposable prototypes so I can afford to compromise on design patterns.
* Even within single procedures, I separate business logic from presentation logic. Ms-access is not an MVC or MVVM environment but I can still use some of the principles. I write a lot of php like code to generate web pages. Each block of code either does business logic or presentation logic. For example, finding a thumbnail image is business logic. Creating a line of html to display a thumbnail with a link to the original image is presentation logic. The first code block will fetch a thumbnail path into a variable, the second will create the line of html with only the variable name in the position of the img src. Even if within a single procedure separation of business and presentation logic improves code readability. Obviously some of the code blocks can be extracted as private functions within the same module to further reduce separate business and presentation logic.

As someone said, using a borrowed design pattern is likely to confuse potential maintainers who are not professional developers. I have worked on a ms-access application that was based on unbound forms and it was terrible to work on. It felt like a foreign paradigm being forced on ms-access. The first thing I did was go back to bound forms. Unbound forms take a long time to perfect as you have to have code for everything the user can possible do.   
  
The question is are there a bunch of simple ideas such as the ones above that do not slow the ms-access development process and which can be taught in seconds to maintainers who are not professional developers.   
  
The core issue seems to be finding ways to take big lumps of code that are usually found in forms and breaking them down to well structured modules or class modules that are easy and safe to maintain and debug. Stepping though code that jumps through vast numbers of basic well tested user defined functions is not fun. Finally check out VBA rubberduck as this tool has some of the answers.