Greenfield Development with ASP.NET MVC & S#arp Lite

Proven guidance for developing ASP.NET MVC applications from idea to well-designed implementation.

by Billy McCafferty  
Copyright © 2012 Codai, Inc.   
January 31, 2012

**DRAFT**

Contents

[Introduction 4](#_Toc316561252)

[Who is the Intended Audience? 5](#_Toc316561253)

[Part I – Planning and Design 5](#_Toc316561254)

[Day 1 – Define the Requirements & Actor/System Interactions 5](#_Toc316561255)

[Objective of the Day 5](#_Toc316561256)

[Inputs 6](#_Toc316561257)

[Activities 6](#_Toc316561258)

[Outputs 12](#_Toc316561259)

[Day 2 – Define the Domain Conceptual Model 12](#_Toc316561260)

[Objective of the Day 12](#_Toc316561261)

[Inputs 12](#_Toc316561262)

[Activities 12](#_Toc316561263)

[Outputs 15](#_Toc316561264)

[Day 3 – Define the Domain Design Model 15](#_Toc316561265)

[Objective of the Day 15](#_Toc316561266)

[Inputs 15](#_Toc316561267)

[Activities 16](#_Toc316561268)

[Outputs 31](#_Toc316561269)

[Part II – Implementing the Design 31](#_Toc316561270)

[Day 4 – Setup the Project, Look & Feel, and Security Infrastructure 31](#_Toc316561271)

[Objective of the Day 31](#_Toc316561272)

[Activities 32](#_Toc316561273)

[Day 5 – Develop CRUD and Data Management Capabilities 40](#_Toc316561274)

[Objective of the Day 40](#_Toc316561275)

[Activities 40](#_Toc316561276)

[Day 6 – Provide Reporting Capabilities 45](#_Toc316561277)

[Objective of the Day 45](#_Toc316561278)

[Inputs 45](#_Toc316561279)

[Activities 45](#_Toc316561280)

[Outputs 45](#_Toc316561281)

[Part III – Integrating with External Services 45](#_Toc316561282)

[Day 7 – Integrate with External Services 45](#_Toc316561283)

[Objective of the Day 45](#_Toc316561284)

[Inputs 45](#_Toc316561285)

[Activities 45](#_Toc316561286)

[Outputs 45](#_Toc316561287)

[References 45](#_Toc316561288)

# Introduction

So you’ve landed your first project! Just days ago your client came to you and said “I’ve got a great idea and I want you to build it!” With unwavering confidence, you quickly retorted “And I’ll build it, and it’ll be friggin’ awesome!” You agree to a price, you put all the business paperwork in place, and you’re ready to get coding…you sit down at the computer, raring to go, and think to yourself “Now where the heck do I start?”

To begin with, you’ve already gotten ahead of yourself. Before a single key stroke of code is developed, it’s time to do some planning. And once the planning is done, and *only* after that, it’ll be time to turn those plans into maintainable, tested code.

While there are plenty of coding samples of implemented best practices, it’s sometimes difficult to figure out how the developer got from an empty directory to an implemented piece of architectural and functional beauty. This article will walk you through the basics of the project life cycle, from project inception to project completion, turning ideas into requirements, requirements into plans, and plans into code, with a clear and blatantly unfair bias towards the developer’s perspective.

This isn’t intended to discuss a particular project management methodology or advocate the principles of any specific development methodology. Instead, this article’s focus is on using a number of tricks of the trade of traditional Object Oriented Analysis & Design (OOA&D) and various development techniques to turn ideas into working code. In other words, bits and pieces from UML, design patterns, OOA&D, Domain-Driven Design[[1]](#endnote-1), Test-Driven Development[[2]](#endnote-2), etc…you know, like what you’d do on a real project.

Although we’ll be using S#arp Lite as the project starting point and architectural structure, most of the principles and techniques we’ll be using will be applicable to just about any development implementation you’d prefer.

The project activities will be examined over seven “days” with each day focusing on a distinct element of project production. Note that each day, when mapped to a real project, may represent an hour, a day, a week, or perhaps even a month(s) depending on the activities involved and the size of the project. (So don’t take this term too literally.)

The seven days of the project will cover three major themes of project delivery:

* Part I focuses on the planning and design elements of project delivery; i.e., turning project ideas into a workable design. Part I is split into three days:
  + Day 1 – Define the Requirements & Actor/System Interactions
  + Day 2 – Define the Domain Conceptual Model
  + Day 3 – Define the Domain Design Model
* Part II focuses on the implementation phase of project delivery; i.e., turning the design into a working project. Part II is split into four days:
  + Day 4 – Setup the Project & Security Infrastructure
  + Day 5 – Develop CRUD & Data Management Capabilities
  + Day 6 – Provide Reporting Capabilities
* Part III focuses on extending the project to integrate with third-party, external services with an appropriate separation of concerns.
  + Day 7 – Integrate with External Services

While this article walks through the development of the project in a waterfall fashion, it does not intend to promote such a methodology; indeed, an agile/iterative approach to development is recommended but is beyond the scope of this article to describe. The reader is encouraged to adapt some or all of the described techniques into your organization’s preferred methodology.

At the end of the article, you should have a basic understanding of how to turn a project idea into a well-designed, maintainable deliverable using real-world techniques, starting from nothing but the idea.

## Who is the Intended Audience?

While I have attempted to write this article to be understandable to any skill level, the reader is expected to have basic knowledge of UML (specifically class diagrams and sequence diagrams) and to be familiar with ASP.NET MVC and C#. For materials on these subjects, refer to Pro C# 2010 and the .NET Platform [Troelsen 2010], Professional ASP.NET MVC 3 [Galloway 2011], and UML 2.0 Pocket Reference [Pilone 2006]. Other references will be provided along the way to help fill in any missing gaps and secondary subjects.

Developers of experienced-beginner to intermediate levels will gain the most from this article.

# Part I – Planning and Design

For the planning and design phase of project delivery, each “day” will have an objective, inputs, activities, and outputs:

* **Objective**: This is a short description of the goal – or expected end result – of the day.
* **Inputs**: These are the informational inputs, document artifacts, and decisions that should have been made before beginning the day’s activities.
* **Activities**: This is the core element of each day, describing the interactions with the client and the practices used for ultimately transforming ideas into production code. For each day, we’ll discuss the activities to be achieved and put them into practice on our example project.
* **Outputs**: These are the artifacts that will have been generated during the day’s activities, some of which may be used as inputs for subsequent days.

## Day 1 – Define the Requirements & Actor/System Interactions

### Objective of the Day

Work with the client to translate ideas into requirements. By the end of the day, you should be able to concisely describe the project’s vision, actors, and implementable requirements.

### Inputs

* Client project ideas, with knowledge of who will be interacting with the system.

### Activities

#### Project Vision

The first activity of the day is to establish the project vision. The vision describes the high-level goals of the system and the business case for implementation. It’s the two-minute elevator speech describing what the system is intended to achieve.

While defining the project vision, a key activity will also be to clearly define the “actors,” or the people who interact with the system and the goal(s) of those interactions, accordingly.

This may seem like a simple, or even dismissible, step, but clearly defining the vision and actors is one of the most essential elements of the entire project. The vision serves as a good litmus test to use to verify if a client idea is relevant to the project’s vision. Additionally, by clearly defining the actors, it allows you and the client to focus on just those requirements which satisfy the needs of those actors; i.e., if an idea comes up which doesn’t serve the needs of one of the defined actors, it can be summarily discarded or put on an “idea parking lot” (so the client feels like you’re actually going to look at it again someday…which you’re not). In other words, don’t skip this!

##### Putting it Into Practice

Our aspiring client’s company is Acme Telecom. Acme Telecom would like to create a support call tracking system to log and monitor all calls, which will be tracked as “support tickets,” coming into its support department. Currently, Acme Telecom’s paper-based system is making it difficult to assess how well its support staff is responding and closing calls. Acme Telecom doesn’t want to try to do too much at once, so the client wants to keep things very simple and small for this first release. Accordingly, the vision that we and the client were able to agree on was as follows:

*Acme Telecom’s paper-based approach to support call tracking is limiting the company’s ability to improve support capabilities. Accordingly, Acme Telecom needs an easy-to-use, secure, web-based Call Tracking System (called “CaTS”) for its Support Staff to track and resolve support calls from Customers as “support tickets.” In addition to facilitating support ticket resolution, the system must provide high-level reporting for Management to filter support ticket logs and to assess key performance metrics, such as rate of resolution and resolution bottlenecks.*

Not only does this vision define the general scope of the project, but did you notice that the actors were also clearly stated? (We can even infer hints of the domain model…but that’ll be for a later discussion.)

Taking the vision a bit further, the actors, and goals of each actor, are as follows:

|  |  |
| --- | --- |
| **Actor** | **Goals** |
| Support Staff | Log support ticket.  Update and/or resolve calls opened by self. |
| Customer | Have support staff resolve calls. |
| Management | Run filterable reports on call logs.  Run “canned reports” to assess resolution performance. |

With that, we have a good general scope definition of the project, we know who the primary stakeholders are, and what each would like to achieve.

At this stage, it doesn’t look like the customer would actually be interacting with the system since they’ll be calling the Support Staff directly…should we include them as an actor? This could be argued either way, but I would say yes due to the fact that they represent the key stakeholder to be satisfied and should be considered during requirements elaboration by asking “Is this idea going to help the Customer get their calls resolved?”

#### User Stories or Use Cases

The User Stories or Use Cases summarize the various features of the system. Each should be small enough to be estimable, but big enough that the client would be willing to pay for its implementation and measure progress.

It is important to decide very early on if requirements will be defined using User Stories or Use Cases (or another means altogether).

Users Stories, an element of XP, are very short requirement descriptions which are intended to be elaborated during development iterations; they are truly markers for further discussion. A User Story is a feature stated from the perspective of the actor(s) who will benefit from that feature. Each User Story typically includes the following details:

* Description: The feature from the actor’s perspective.
* Priority: Higher priority features should be addressed earlier in the project life-cycle while lower priority features may be dropped in favor of higher priority change requests, or dropped altogether to assist with meeting schedule deadlines.
* Risk: Higher risk User Stories may be A) split into smaller risk User Stories, B) researched to mitigate risk, C) reflected as a higher uncertainty in the estimate, or D) dropped to reduce uncertainty in scope and schedule.
* Estimate: Typically described in “points” rather than in hours to better facilitate relative-estimating techniques.

To learn more about User Stories, I encourage you to read (LINK TO FURTHER READING).

Use Cases, on the other hand, include much more detail for each requirement. For example, while a User Story may be described in a sentence or two, a Use Case describes prerequisites for the Use Case to be activated, a “happy path” of execution, exceptive cases to execution, and post-conditions, describing the state of the system after completion. To learn more about Use Cases, I encourage you to read (LINK TO FURTHER READING).

There are pros and cons to each; google “User Story vs. Use Case” to get a swath of opinions. Personally, I prefer User Stories if I feel the client is very unsure of what the final system will be, or if the client is supportive of flexible scope; I prefer Use Cases if the end result is very definable or if the client demands fixed bid (which always makes me shudder). There’s more to it than that, but for the purposes at hand, we’re going to go with User Stories for Acme Telecom’s Call Tracking System (CaTS).

##### Putting it Into Practice

Now that the vision, actors, and actor-goals have been defined, we’re ready to define the requirements of the solution. We’ll do so by working with the client to define the requirements as User Stories. To keep things simple, we’ll ignore Priority, Risk and Estimate, focusing simply on the description of each. Our User Stories for the Call Tracking System are as follows:

* Support Staff may open a new support ticket and provide details including: Customer, Issue Details (such as description and “dynamically managed” type), Status, and Resolution Details (if resolved immediately). Status of the call may be New, In Progress, or Resolved. (“Dynamically managed” means that if the issue type isn’t available, the Support Staff may enter a new one when entering the support ticket.)
* Support Staff may manage Customer details, adding and updating Customer information when necessary.
* Support Staff may view listing of all tickets opened by self, defaulting to those which are not resolved and filter the listing, accordingly.
* Support Staff may search for an existing support ticket, opened by self, view details of ticket, and update its status and resolution details.
* Management may view listing of all tickets, defaulting to those which are not resolved and filter the listing, accordingly.
* Management may view report showing information regarding how quickly calls are being resolved, called the “Ticket Resolution Report by Support Staff.” The report may be run for all Support Staff or for a specific Support Staff employee. The manager creating the report may only include Support Staff of whom they are manager.
* Admin may view the “Ticket Resolution Report by Support Staff” for any and all Support Staff.
* Management may view chart showing a breakdown of call by issue type and average resolution time for each issue type, called the “Ticket Resolution Report by Issue Type.” The manager creating the report will only see stats for Support Staff of whom they are manager.
* Admin may view the “Ticket Resolution Report by Issue Type” to include any and all Support Staff.
* Admin may manage all details of Support Staff and Management, along with which Support Staff are assigned to which Manager. Each Support Staff may only have, and must have, one Manager.

#### Data Dictionary

One of the most frequently skipped steps of project delivery is also one of the simplest and helpful: the creation of a data dictionary. A data dictionary is a simple glossary of domain-related terms which serves to reduce requirements confusion between the client and project delivery team.

As project requirements become more complicated, a data dictionary becomes increasingly important to ensure everyone is speaking the same language. (Not) Surprisingly, it’s frequently difficult for even client associates to agree upon term definitions; a data dictionary helps to get people talking and to (hopefully) agree upon the meaning of business terms. Forming this agreement of definitions forms the basis of Ubiquitous Language, a prominent component of Domain-Driven Design.

A few years back I developed a project management tool for a construction management project. One of the key features of this tool was a Master Budget Report which combined information from various sources to ultimately provide an Estimate at Completion for each project. The report’s data columns included Expenditures, Change Orders, Pending Obligations, Remaining Obligations, Total Obligations, and others. It took over three months of workshops involving client stakeholders from the financial group, project controls group, project managers, and upper management to agree upon what constituted a Pending Obligation vs. a Remaining Obligation and to sort out other such definitions associated with the report. Imagine the frustrations that would have ensued if we had tried to develop this “standard” report when no one agreed to what was actually “standard.”

The lesson of the story is…don’t underestimate the usefulness of a well-discussed data dictionary. And knowing is half the battle.

##### Putting it Into Practice

Acme Telecom’s CaTS is starting to take shape; we’ve agreed to the vision and even the requirements. But during requirement elaboration discussions, there was some confusion on a few key terms. The following data dictionary helped to alleviate the confusion:

|  |  |
| --- | --- |
| **Term** | **Definition** |
| Customer | A person who has an Acme Telecom account number. (Anyone else is just an annoying person wasting Support Staff’s time.) |
| Resolution Time | The number of business days (whole number) to resolve a call is calculated by taking the difference between the day a ticket is opened, and the day when it is resolved, excluding weekends. If a ticket is resolved in the same day it was opened, its resolution time is 0 days. |
| Avg. Resolution Time | This is the number of mean business days (decimal number) it takes for an arbitrary group of support tickets to be resolved; i.e., the sum of business days to resolve each ticket divided by the number of tickets. |

It was easy to assume that everyone understood what “resolution time” meant when discussing requirements. But as we attempted to define this term in the data dictionary, it was recognized to be a non-trivial idea. Having this cleared up will help to ensure that anyone using the system has the same understanding of the term...assuming they’ve taken five minutes to read the data dictionary, which is quite an assumption in and of itself!

It should also be noted that the data dictionary was used to not only clarify project-related terms and calculations, but also to clarify the definition of actors.

While it’s not necessary to document every project term, any term which involves tricky calculations, subjectivity in defining, or has been identified as causing disagreement, should certainly be included. Furthermore, the data dictionary may be separated into separate lists such as Actors, Domain, and Calculations, but start simple and expand when necessary.

#### Actor/System Interaction Diagrams

I’ve been a strong proponent of agile development techniques for years; but a challenge I’ve always had is keeping track of the “big picture” as requirements are broken down into more granular requirements as user stories. An associate of mine, Jim Tucker, introduced me to a quick and easy technique that I’ve not seen surpassed for expressing the big picture in a clear and concise manner.

Before delving into technical design, Jim creates Actor/System Interaction Diagrams (as UML sequence diagrams) to document how the actors interact with each other and with the underlying system, treating the system as a black box. It’s possible to use this technique not only for the overall system, but down to the level of a single user story. This facilitates “zooming in and out” of the details as appropriate.

This is similar in nature to the back and forth between an actor’s actions and system’s responses described within a Use Case (an alternative approach to documenting feature requirements), and to Craig Larman’s System Sequence Diagrams in Applying UML and Design Patterns (INCLUDE BOOK LINK HERE). It differs from Use Cases in the fact that it’s visually oriented, and differs from both in that it may be easily expanded to include many participants and need not be constrained to a single feature. On a recent project, we used this technique to illustrate the high-level interactions among eight actors and multiple third party systems.

To use this approach, it’s first necessary to understand UML sequence diagrams, which is beyond the scope of this article. (Googling “UML sequence diagrams” will get you going in the right direction.) Actor/System Interaction Diagrams differ from sequence diagrams in the following ways:

* Actors are used in place of classes.
* The system itself is treated as a single participant in the sequence diagram.
* The “messages” among participants, instead of being API calls, are requests between actors and/or the system itself.

But, as usual, an example speaks a thousand words…

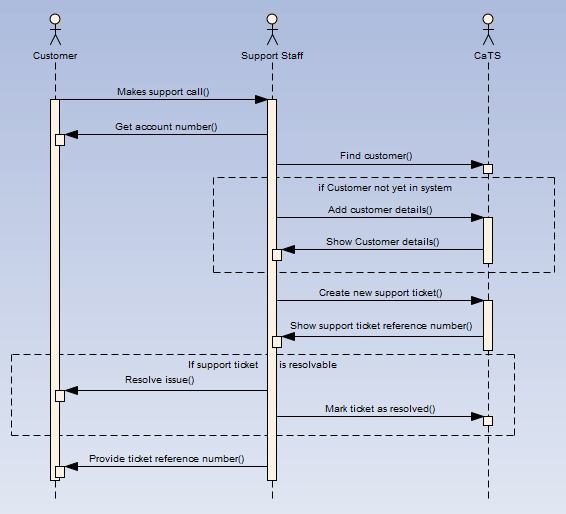
##### Putting it Into Practice

The trickiest User Story in the CaTS application is as follows:

*Support Staff may open a new support ticket and provide details including: Customer, Issue Details (such as description and type), Status, and Resolution Details (if resolved immediately). Status of the call may be New, In Progress, or Resolved.*

This one caused the most discussion and had the most unknowns. Accordingly, it was decided that an Actor/System Interaction Diagram would be created to clarify the sequence of interactions.

The actors that will be in the sequence diagram include Customer, Support Staff, and the CaTS application. The sequence of events to handle a new support call is as follows:

1. Customer calls Support Staff and describes the issue.
2. Support Staff opens a new support ticket in CaTS which requires the following sub-steps:
   1. Support Staff gets Customer account number.
      1. Support Staff uses account number to find customer.
      2. If not found, Support Staff adds a new Customer entry.
      3. If found (or after created), Support Staff generates a new support ticket for the Customer.
   2. Support Staff adds issue details to the new support ticket entry.
3. If resolvable, Support Staff assists the Customer and marks the support ticket as resolved.
4. Otherwise, Support Staff gives the Customer the support ticket reference number and pinky swears to resolve the issue and get back with them.

The Actor/System Interaction Diagram to illustrate this User Story is as follows:

The diagram shows the actors involved and their interactions with each other. Although this level of detail isn’t necessary for every User Story, it’s a useful technique for illustrating the business processes behind one or more User Stories. Furthermore, this technique need not be limited to the User Story level; alternatively, you could generate Actor/System Interaction Diagrams for major modules and/or include third party systems as additional “actors.” UML sequence diagrams are quite versatile for this kind of work.

### Outputs

* Project Vision
* User Stories or Use Cases
* Data Dictionary
* Actor/System Interaction Diagrams

## Day 2 – Define the Domain Conceptual Model

### Objective of the Day

Transform the requirements from Day 1 into an appropriate domain *conceptual* model reflecting objects, attributes, and associations.

### Inputs

* Project Vision
* User Stories or Use Cases
* Data Dictionary

### Activities

The domain *conceptual* model is a visual representation of conceptual or real-world objects in the domain of interest [Fowler, 1996]. (The domain *design* model, which will be discussed in Day 3, is a visual representation of the classes and behaviors which will be implemented in the software.)

The conceptual model is *not* a class diagram describing classes with methods. But in practice, the conceptual model will iteratively *become* the basis of the class diagram of the design model. Specifically, the conceptual model reflects the following information:

* Domain conceptual objects,
* Associations between conceptual objects, and
* Attributes of conceptual objects.

The important distinction between conceptual model and design model is that the conceptual model should represent a clear business-oriented abstraction of the domain; i.e., the client should be able to look at the conceptual model and understand it fully without difficulty. Compare this to the design model which will include method names, data types, polymorphic associations, and supporting classes, such as factories and services.

To illustrate, suppose your domain includes the concept of “Customer” and “Employee.” The conceptual model would reflect two separate objects, each having duplicate attributes such as first name and last name. The design model may instead reflect a parent “Person” class, inherited by both Customer and Employee, having duplicated fields moved up to the Person superclass. Applying UML and Patterns [Larman 2004] goes into great deal concerning this process and is highly recommended read.

We’ll get into more details of the design process when we see it in action, next.

##### Putting it Into Practice

For Acme Telecom’s CaTS, the business requirements have been established. It is now time to begin transforming the requirements into a domain model.

The first step along this path is to define the conceptual model. We’ll define the conceptual objects, define their relationships with other objects, and add attributes, describing the pertinent information associated with each. But how exactly do we go about naming the conceptual objects? [Larman 2004] suggests looking for objects which fall into a Conceptual Class Category, such as:

* Physical or tangible objects,
* Places,
* Transactions and transaction line items,
* Roles of people,
* Physical container of other things,
* Things in a container,
* Organizations,
* Events,
* Processes,
* Rules and policies, or
* Nouns.

For finding associations, Larman suggests looking for the following types of associations:

* A is a physical or logical part of B,
* A is physically or logically contained in/on B, or
* A is recorded in or is a line item of B.

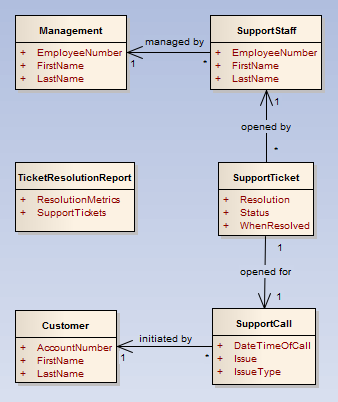
Now that we know what to look for, what artifacts do we look at as potential sources for objects? The primary sources include the vision, data dictionary, and user stories; i.e., the artifacts from Day 1.

As a caveat, if you find conceptual objects in the vision or data dictionary that are *not* included in the user stories, you should double check to make sure that the user stories are adequately representative of the requirements. While you may find extra concepts in the data dictionary, since it provides explanatory details of terms, extra concepts in the vision – but not found in the user stories – likely indicates an inadequacy, accordingly.

Looking at the inputs from the previous day, potential conceptual objects include:

* Customer
* Support Staff
* Management
* Support Call
* Support Ticket
* Ticket Resolution Report per Support Staff
* Ticket Resolution Report per Issue Type

Compiling the list of concept objects will likely incite useful discussion, much of which will help to ensure that the business has been adequately modeled, and how the concept model will be later transformed into the design model.

At right, review an example of how the conceptual model could be visually modeled, for the concepts discussed, with associations and attributes. As shown, all of the major elements of the conceptual model have been visualized. There are some interesting things to note concerning this model:

* There are many duplicated fields between Management and SupportStaff. When implemented, these will likely be combined into a single Employee class; but recall that this is the *conceptual* model and should include more than less detail to cover all of the unique concepts within the requirements.
* SupportTicket and SupportCall have a one-to-one relationship with each other. These may have instead been merged into a single concept (and will be done so in the design model), but splitting them into separate concepts at this stage emphasizes that there is a call from the customer which leads to a ticket being opened.
* TicketResolutionReport is a report concept which includes a listing of SupportTickets and a number of ResolutionMetrics. Note that there is not an explicit association between TicketResolutionReport and the SupportTicket object to emphasize that the report, rather than being a core element of the domain, is an auxiliary concept to the domain. This is a subtle and subjective point but can help to establish *bounded contexts*[[3]](#endnote-3) during the design of the implementation. While it would not have been wrong to have included an explicit association, doing so would not have added much useful meaning to the context of the requirements. Associations between objects in a conceptual model should indicate some degree of permanence.
* As an additional note, the ResolutionMetrics attribute of TicketResolutionReport may indeed be a separate concept object itself; but at this point, it’s added as a placeholder, to be expanded upon at a later time.

As may be inferred, there’s certainly a lot of subjectivity and brainstorming that goes into creating the domain conceptual model. As such, don’t try as much to get it “just right” as to ensure that it comprehensively captures all of the key concepts and associations within the requirements.

When creating the conceptual model and you are faced with the decision to split a concept (or merge separate concepts), err on the side of splitting the concept to provide more granularity than may be necessary for the actual implementation; i.e., it’s better for the conceptual model to be too expressive than too simplistic. Once the concepts are established, you can always combine and regroup as necessary for the design model, but you don’t want to miss anything along the way.

As a final note, defining conceptual objects is more important than identifying the associations among them, and more time should be spent on object identification, accordingly.

### Outputs

* Domain conceptual model

## Day 3 – Define the Domain Design Model

### Objective of the Day

Transform the domain model and actor/system interaction diagrams into a design model, which will be used as the basis for implementation. The design model will include two artifacts:

* **Class Diagram**: A class diagram extends the domain conceptual model to include data types, methods, and supporting classes necessary for implementation.
* **Sequence Diagrams**: Sequence diagrams express the interactions among classes and the messages invoking, and responding to, those interactions, accordingly.

### Inputs

* Actor/System interaction diagrams
* Domain conceptual model

### Activities

Designing the domain *conceptual* model, and evolving that into the domain *design* model represents the transition from business-oriented to implementation-oriented requirements. While there are two outputs from this process, class diagrams and sequence diagrams, they are defined in parallel with each other, each contributing to the design of the other.

Transforming the conceptual model into a design model is effectively done iteratively. This does not mean that each refinement need take a two-week iteration to complete. Indeed, each iterative refinement may only take 30 minutes or so to complete. Tackling this transformation iteratively allows you to learn from each pass, applying new knowledge to the class diagram and to the sequence diagrams in successive passes.

Furthermore, you need not tackle the entire domain model when defining the design model; only focus on the scope and level of detail that is immediately useful.

The steps involved with each elaboration iteration, for defining/refining the class diagram and sequence diagrams, are as follows:

1. Identify the bounded context which will be the focus of design model elaboration. (The bounded context need not be strictly adhered to, but should be identified to assist in focusing the elaboration effort.)
2. Select one or more high priority User Stories (or Use Cases) which are representative of the bounded context and, preferably, are of higher risk.
3. Refine (or initially define) the class diagram, focusing on classes, attributes, and associations within the context of the User Stories.
4. For each User Story, refine (or initially define) sequence diagrams illustrating the behavior and interactions of each class involved.
5. Update the class diagram to reflect any new or modified classes, behaviors, and/or attributes discovered during the design of sequence diagrams.

When defining sequence diagrams, do not feel constrained to only include classes which are already defined within the class diagram. The exercise of building out the sequence diagrams will quickly lead to the discovery of new classes and behaviors which are not yet a part of the class diagram; that’s one of the points of doing this. Furthermore, *do not* feel obliged to create a sequence diagram for every User Story or requirement. Sequence diagrams are time consuming to create and are quickly subject to change. Accordingly, I see sequence diagrams as disposable artifacts which need not be maintained after they have been implemented as code.

Sequence diagrams are particularly useful at the following times:

* To illustrate the interactions involved with a complex User Story,
* To illustrate a “boiler-plate activity,” such as the interactions involved with the CRUD (create/read/update/delete) of a single domain object, in order to establish a standard practice and to serve as a useful reference for new team members, and
* To tell a more junior developer *exactly* how you’d like something implemented.

In the final step, when updating the class diagram to reflect new or modified behaviors, use each message, passed between two classes within the sequence diagram, to indicate the need for a respective method on the class from which the message originates. In other words, if a communication is indicated between two classes in the sequence diagram, there should be a respective method (or return result from a message) in the class diagram.

Deciding to what granularity to create the sequence diagrams is a tricky decision. If too little time is spent, then the sequence diagram will provide very little guidance towards implementation. But too much time may result in granularity which is otherwise obvious or clearly assumed. As a rule, continue to break-down the sequence diagram to further levels of granularity until:

* The effort is no longer reducing uncertainty and risk,
* The increased granularity is no longer providing “new” information, or
* The increased granularity is no longer useful to the task at hand (such as providing an initial estimate).

From the perspective of including this practice within an overall project methodology, defining/refining the design model should be roughly performed at the beginning of a project to assist with estimating and to improve understanding of the project. It should also be performed in more detail at the beginning of each development iteration to define/refine the design model for that iteration.

#### GRASP Patterns

The definitive challenge at this point is figuring out exactly how to transform the conceptual domain model into an appropriate design model describing classes with attributes, associations, and behavior. While sequence diagrams are a tool to visualize interactions among classes and their behaviors, the diagramming technique itself does not provide guidance on *how* to determine how the classes will interact and what their respective behaviors will be. Doing this correctly is a critical element to ensuring the codebase is well-designed and maintainable.

GRASP patterns provide guidance for assigning responsibility and interactions to objects; GRASP is an acronym for General Responsibility Assignment Software Patterns and were defined to provide guidance for OOA&D. While there are nine GRASP patterns in all, five key GRASP patterns should be mastered and considered continuously in the creation of the design model. What follows are the five patterns, each including the problem that arises when adding behavior to a model, and the guiding principle/solution for meeting that challenge. (Reference Applying UML and Patterns by Craig Larman [Larman 2004], from which the following patterns are quoted, for a very well written discussion of these principles and other fundamentals of OOA&D.)

###### Information Expert

**Problem**: What is a general principle of assigning responsibilities to objects?

**Solution**: Assign a responsibility to the information expert – the class that has the *information* necessary to fulfill the responsibility.

###### Creator

**Problem**: Who should be responsible for creating a new instance of some class?

**Solution**: Assign class B the responsibility to create an instance of class A if one or more the following is true:

* B *aggregates* A objects.
* B *contains* A objects.
* B *records* instances of A objects.
* B *closely uses* A objects.
* B *has the initializing data* that will be passed to A when it is created (thus B is an Information Export with respect to creating A).

An exception to this is that if the logic required to create class A is of considerable complexity, consider moving the creation logic to a helper class, called a Factory [GoF 1995].

###### Low Coupling

**Problem**: How to support low dependency, low change impact, and increased reuse? Coupling is how strongly one class is connected to, or has knowledge of, other classes. Highly coupled classes suffer from the following problems:

* Changes in related classes force local changes.
* Harder to understand in isolation.
* Harder to reuse because its use requires the additional presence of the classes on which it is dependent.

**Solution**: Assign a responsibility so that coupling remains low.

This terse solution is not very helpful; indeed, it’s often simpler to instill low coupling by watching for signs of tight coupling and to refactor, accordingly. Martin Fowler’s Refactoring [Fowler 1999] is by far the best resource available for becoming familiar with these “smells” along with concrete guidance for taking corrective action.

###### High Cohesion

**Problem**: How to keep complexity manageable? Cohesion is how strongly related the responsibilities of a class are. Classes with low cohesion suffer from the following problems:

* Hard to comprehend.
* Hard to reuse.
* Hard to maintain.
* Delicate and constantly effected by change.

**Solution**: Assign a responsibility so that cohesion remains high.

Like the solution for Low Coupling, this isn’t immediately useful advice. The point is to aim towards encapsulating related logic together, and to introduce new classes as necessary to continue to do so. Again, [Fowler 1999] is a tremendous resource for this topic.

###### Controller

**Problem**: Who should be responsible for handling an input system event?

**Solution**: Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:

* Represents the overall system as a Façade [GoF 1995].
* Represents a User Story or Use Case scenario within which the system event occurs, often named <UserStoryName>Handler, <UserStoryName>Coordinator, or <UserStoryName>Controller.

As an aside, this GRASP pattern is nicely enabled via ASP.NET MVC with an explicit controller layer. But as the naming of a controller would imply, the scope of each controller should be limited to that implied by its name, accordingly. For example, a controller by the name of “ManageUsersController” would imply performing User-related CRUD operations; User-related report generation should likely belong to a separate controller. Incidentally, this would also better support High Cohesion – what is good for the support of one pattern is often good for another.

#### CRC Cards

It is sometimes difficult to figure out the very first step for adding behavior and collaboration activities to objects. A technique which may assist with getting the ball rolling or to help when brainstorming the behavior of a new bounded context within a domain is a technique known as CRC cards[[4]](#endnote-4) (Class-Responsibility-Collaborator cards).

CRC cards are very simple yet effective for defining brainstorming the responsibilities of each object. To utilize, grab some index cards, title each with a class name of interest, and write two things on each:

* **Responsibilities**: For each CRC card (i.e., each class), write one or a few bullet points describing the extent of responsibility for the class. Keep GRASP patterns in mind when defining the scope of responsibility.
* **Collaborator Objects**: List one or more collaborator classes which need to be interacted with in order to carry out the responsibilities of that class.

After documenting responsibilities and identifying collaborator objects, the task of transforming this information into the class diagram, and into sequence diagrams, is greatly simplified.

#### Refactoring & Refactoring to Patterns

GRASP patterns are a tremendous guide for answering the question of “what belongs where.” But as a system evolves, changes are inevitably introduced by client request, or more frequently, by further discovery of requirements. For example, what may at first appear to be simple may quickly become complex after coding begins. In these cases, *refactoring* techniques can greatly assist to answer the question of “how to adapt to change.” In a nutshell, refactoring is the practice of improving the design of existing code. While the details of refactoring are beyond the scope of this article, two tremendous resources (i.e., absolutely required reading for any developer) are Refactoring by Martin Fowler [Fowler 1999] and Refactoring to Patterns by Joshua Kerievsky [Kerievsky 2004].

##### Putting it Into Practice

This day’s “Putting it Into Practice” will consist of a number of example artifacts, including the class initial and revised class diagram, CRC cards, and sequence diagrams. Let’s review each in turn.

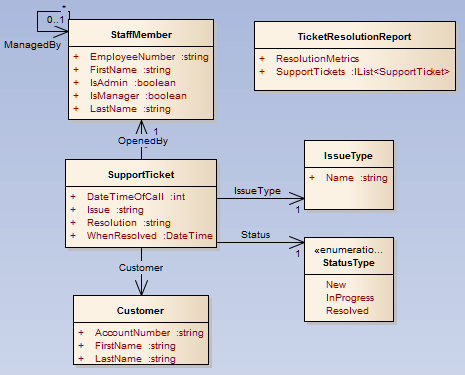
By far, transforming the concept model into the design model is the one of the trickiest tasks and the most important to get as close to “right” as possible. The rule to keep in mind is to design the simplest model which could possibly work to fit the needs of the requirements and elaborate iteratively and only when necessary. (Keep in mind the design elaboration steps discussed above.)

###### Initial Domain Class Diagram

At some point, the concept model must be transformed into an implementable design model. This is the time to put our techie hats on and start to think in terms of classes, properties and methods. To get the ball rolling, let’s take a stab at designing a minimal class diagram to support the concept model discussed earlier.

Initially, we’ll focus on the bounded context of support ticket management, and the objects which participate with this activity. We’ll leave the reporting context until later on; this will also give Acme Telecom a little more time to make up their mind on what they want to see in the report.

What follows is the class diagram reflecting the first attempt at modeling the domain as an implementable design:



There are a number of differences from the conceptual model which were carefully considered:

* The concepts Management, SupportStaff, and Admin have been combined into a single class called StaffMember. The following considerations were taken into account to make this decision:
  + All of these concept objects are employees of Acme Telecom.
  + Each of the concepts share duplicate data.
  + While each may be allowed to do different tasks (e.g., manage staff vs. open support tickets), it does not yet appear that the classes themselves would behave differently. I.e., it’s not yet apparent that an Admin class would have different methods than a Manager class.
  + Having the flags “IsAdmin” and “IsManager” may indicate that we may need role-management capabilities; while this may be true, we’ll want to avoid delving into the weeds further until warranted. (With that said, we’ll want to fully think out security management before we begin coding…it’s much more difficult to incorporate security management later on in project development.)
* The concepts SupportTicket and SupportCall have been merged into a single class called SupportTicket. This could have been argued either way, but the two concepts are equivalent from a data workflow perspective. Consequently, I went for the simpler approach.
* SupportTicket.Status has been pulled out as an enum value as we rarely expect the status types to change. Additionally, since the status may also be considered within data workflows, having it as an enum value will simplify the workflow conditionals and provide support for the use of a finite state machine, if deemed necessary down the road. (See Refactoring to Patterns [Kerievsky 2004] for guidance on when/how to make this decision.)
* Preferably, SupportTicket.IssueType would also be an enum to keep things simple. But, the requirements state that the SupportStaff must be able to dynamically add new issue types, when needed, when opening a support ticket. Consequently, IssueType will need to be a persistable domain class.

Take special note that this model does not imply anything about infrastructural details such as persistence and security. These will be added to the class diagram as they are included in the sequence diagrams. Trying to anticipate all of the infrastructural classes at this time will likely lead to over-engineering (or *wrong*-engineering altogether).

With that said, the architect should have a good idea of how such infrastructural concerns will be addressed in the final solution. It is appropriate to have at least discussed how the overall project will be architected and what tools will be used to support infrastructural concerns. The overall architecture will have a strong impact on the sequencing of messages through the application tiers.

For the task at hand, we’ll take into account the fact that we’ll be using S#arp Lite as our underlying framework and architecture.

**Accordingly, before going further, read the article “S#arp Lite: The Basics” to get a better understanding of how a S#arp Lite project is architected:** [**http://devlicio.us/blogs/billy\_mccafferty/archive/2011/11/11/s-arp-lite-the-basicss.aspx**](http://devlicio.us/blogs/billy_mccafferty/archive/2011/11/11/s-arp-lite-the-basicss.aspx)**.** Pay particular attention to the package diagram, describing how the layers are organized.

###### Sequence Diagram for Opening a New Support Ticket

Did you read the introduction to S#arp Lite as referenced above? OK, great, we’re ready to continue.

We have designed the beginnings of the class diagram. We’re now ready to elaborating, refining, and adding behavior to the class diagram by analyzing the sequence of events involved with opening a new support ticket. This particular user story was selected because it covers much of the bounded context that we’re currently working on and is a critical element of the CaTS application.

As we’ll see, the design of the sequence diagram is where we have to seriously start thinking about:

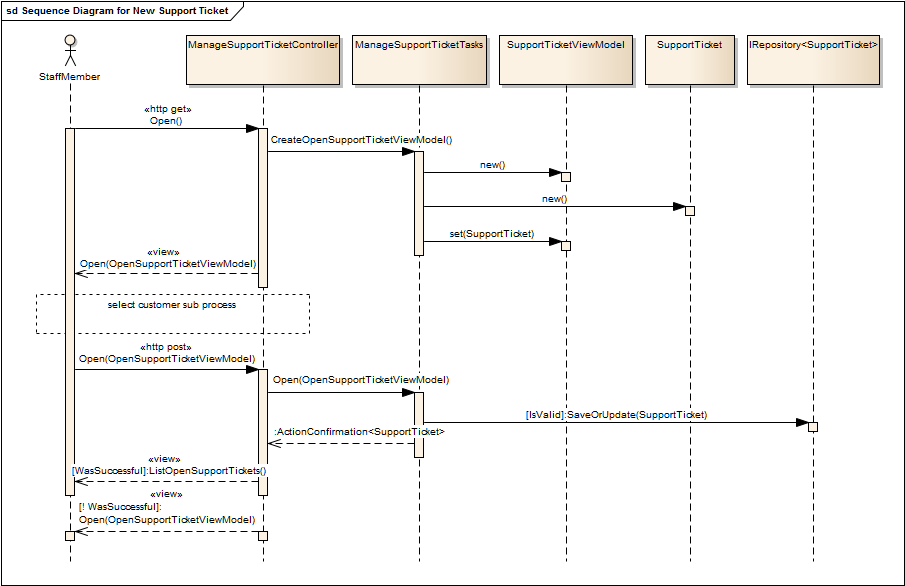
* How will data be loaded from the database?
* How will data be saved to the database?
* How will messages travel among the Web, Controllers, Tasks, Domain, and Data layers of the project?
* How will the Tasks layer expose the core API of the application to the Controllers layer?

While this is a daunting task, be patient and revise your first sequence diagram many times until you feel it’s clear and clean.

Let’s now consider a sequence diagram for the following user story:

*Support Staff may open a new support ticket and provide details including: Customer, Issue Details (such as description and “dynamically managed” type), Status, and Resolution Details (if resolved immediately). Status of the call may be New, In Progress, or Resolved. (“Dynamically managed” means that if the issue type isn’t available, the Support Staff may enter a new one when entering the support ticket.)*

To make things simpler, ignore any extraneous features in the first design of the sequence diagram to focus on the core of the requirements; e.g., we’ll ignore the dynamically managed issue types for now.



The sequence diagram above describes the collaboration of objects in support of allowing a staff member to open a new support ticket.

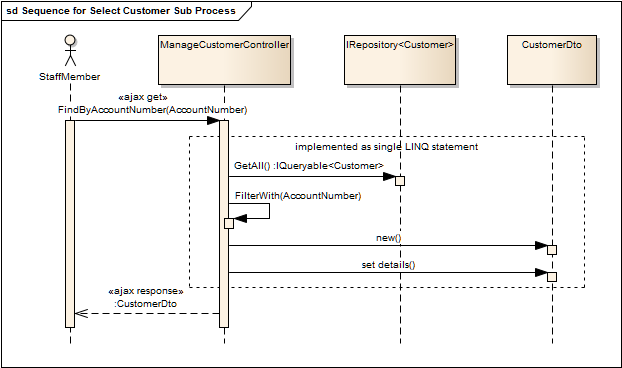
While designing the above sequence diagram, four new objects were identified, which may be added to the class diagram:

* **ManageSupportTicketController**: This MVC controller class will reside in the .Web layer of the project and will have the responsibility of handling requests for the management of a support ticket.
* **ManageSupportTicketTasks**: This tasks class (aka “application service” class [Fowler 2002]) will reside in the .Tasks layer of the project and will have the responsibility of carrying out the coordination activities for the management of a support ticket.
* **SupportTicketViewModel**: This view model DTO (data-transfer object) will carry information from the controller to the view and vice-versa for the purposes of allowing a StaffMember to add/update support ticket information.
* **IRepository<SupportTicket>**: This interface will provide a means to communicate with the underlying data access mechanism to persist the new support ticket to the database.

In a nutshell, the sequence is as follows:

1. The StaffMember requests to open a new support ticket.
2. The StaffMember is presented with a form.
3. A sub-process facilitates the StaffMember to select the respective customer (elaborated below).
4. The StaffMember provides the requested information and submits the form.
5. The StaffMember is either:
   * presented with the same form, with validation information, if the submitted information contained validation problems, or
   * presented with a listing of open support tickets with a message stating that the support ticket was successfully saved.

We’ll now turn our attention to look at the sub-process for selecting a customer...



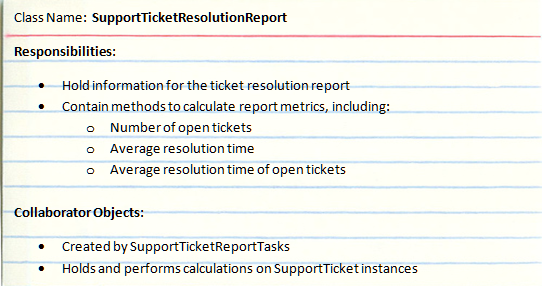
The sequence diagram above describes the sequence of events to allow a StaffMember to select a customer via an AJAX post-back on the page. The form will have a “Customer Account Number” input box on it; when it changes, an AJAX post-back will try to find the customer record. If the customer exists, the customer information will be dynamically shown; if it doesn’t, the StaffMember will need to provide a few customer details before submitting the form.

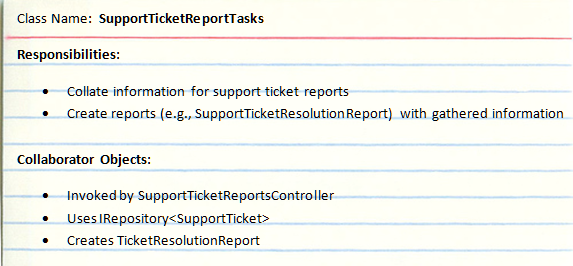
As implied, many details about a user story are discovered during the process of designing a sequence diagram. To ease organization of the requirements, the sequence diagrams and details may be simply printed out and stapled to an index card with the user story written on it, or uploaded as attachments to the user story in an online requirements management tool.

In both sequence diagrams, arguments may be made that all of the UML is not “by the book”; e.g., the use of stereotypes has been basterdized a bit to make the diagram more expressive. I bring this up to express the point that less time should be spent worrying about making a UML diagram perfect as should be spent conveying useful information. When everything’s said and done, these diagrams will likely be thrown away once they’ve been implemented as working code.

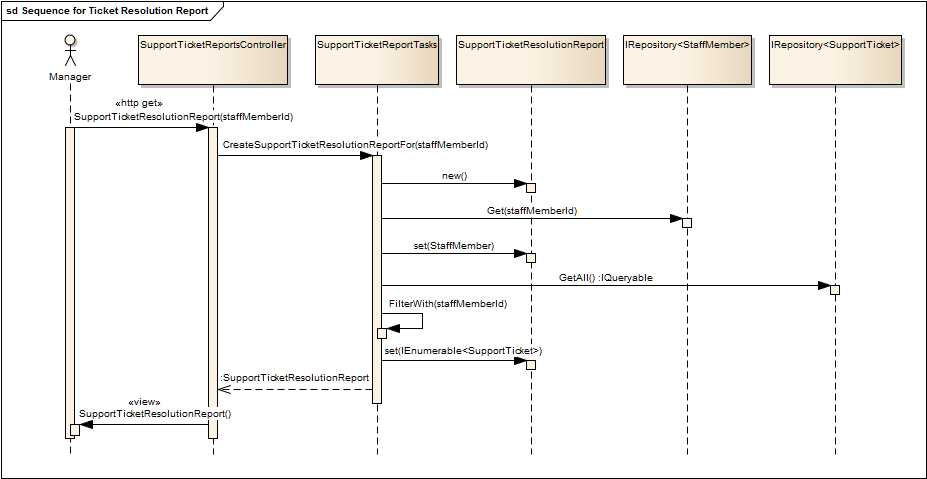
If the underlying project architecture is already established (e.g., we’re using S#arp Lite as the foundation), designing the sequence diagram is relatively straight-forward. If the intention is to grow the architecture “organically,” then elaborating the sequence diagram may take a few iterations.

###### CRC Cards for “Ticket Resolution Report by Support Staff”

In our discussions thus far, we haven’t gotten into the details of exactly how the support ticket resolution report will be generated. So before tackling the respective sequence diagram, we’ll use CRC cards to brainstorm a bit about the classes to be involved with the user story. The CRC cards don’t need to be exhaustively detailed; just enough in order to move on to the associated sequence diagram. CRC cards don’t/shouldn’t need to be maintained throughout the project life-cycle; once they’ve served their usefulness, toss them (or archive them away to some hidden place if throwing them away saddens you).

As demonstrated, the CRC cards were used to work out the basics of the classes involved with generating a support ticket resolution report, per the requirements. Now that the general responsibilities have been defined, we’re able to confidently move on to designing the respective sequence diagram.

###### Sequence Diagram for Creating a “Ticket Resolution Report by Support Staff”



The sequence diagram above describes the collaboration of objects in support of the creation of a support ticket resolution report for a specific staff member.

While designing the above sequence diagram, four new objects were identified, which may be added to the class diagram:

* **SupportTicketReportsController**: This MVC controller class will reside in the .Web layer of the project and will have the responsibility of handling requests for the creation of support ticket related reports.
* **SupportTicketReportTasks**: This tasks class will reside in the .Tasks layer of the project and will have the responsibility of carrying out the coordination activities for the creation of support ticket related reports.
* **IRepository<StaffMember>**: This interface will provide a means to communicate with the underlying data access mechanism to retrieve staff member information.
* **IRepository<SupportTicket>**: This interface will provide a means to communicate with the underlying data access mechanism to retrieve support ticket information.

We’ll now turn our attention to look at the revised domain class diagram, taking into account what we’ve learned with the creation of the CRC cards and sequence diagrams.

###### Revised Class Diagram.pngRevised Domain Class Diagram

Now it’s starting to look like a real application! The class diagram is now including enough classes to warrant noting which project layer each class belongs in. We also have enough information to decorate classes with methods (i.e., behavior) in addition to attributes. If you’ve been paying attention, you may notice that some of the class names are slightly different from the CRC cards and/or sequence diagrams; e.g., ManageCustomersController was called ManageCustomerController in the sequence diagram. This is perfectly fine and expected; your design should continue to evolve as conventions are established, details are discovered, and new changes are incorporated.

With the sequence diagrams established to describe how the objects collaborate, and the class diagram designed to show how the classes are organized, we’re ready to begin switching gears from design to implementation. If you’ve been designing user stories along the way, the actual implementation should be the easy part!

### Outputs

* Domain class diagram
* Sequence diagrams

# Part II – Implementing the Design

In Part II, we change our attention from design to implementation. Rightly so, it seems it has taken quite a while to write a single line of code. While the design activities in Part I may at first appear tedious and extraneous, even a little bit of design planning can pay dividends in an application which is scalable, maintainable, and which properly reflects the business needs of the client. Certainly, the time put into the design effort should be relative to the size and complexity of the project.

Each day of Part II will describe the objective of the day and the activities necessary to achieve the objective. Explanatory discussion will be interspersed throughout. It should be assumed that any all outputs produced from previous days should be referenced as needed during implementation. Each day has a corresponding folder found within the S#arp Lite download package, e.g., \SharpLite\Example\CaTS\Day4. Each day folder (Day4 – Day7) contains the finished product for the day, respectively. Some activities will simply copy code from these folders in order to simplify documentation; accordingly, you’ll want to download the S#arp Lite package from the GitHub repository at <https://github.com/codai/Sharp-Lite/downloads> in order to follow along.

## Day 4 – Setup the Project, Look & Feel, and Security Infrastructure

### Objective of the Day

Establish the foundational elements of the project including:

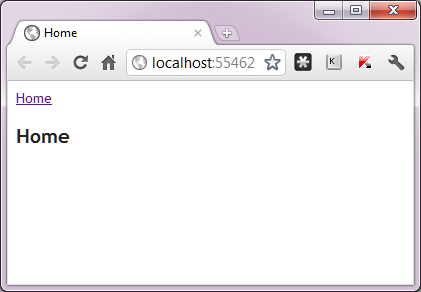
* Setup initial code base using S#arp Lite project template,
* Establish look & feel, and
* Implement basic security infrastructure.

### Activities

###### Prerequisites

* Install Visual Studio 2010 (VS)
* Install SQL Server
* Install ASP.NET MVC 3
* Install Templify (available from <https://github.com/endjin/Templify/downloads>)

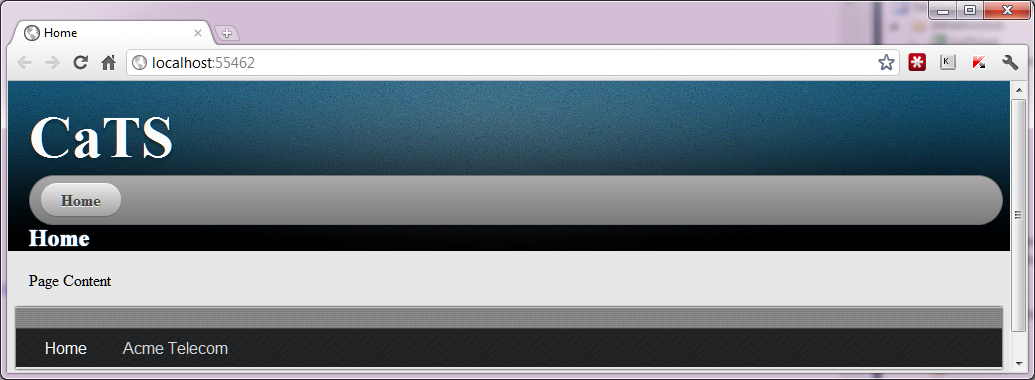
###### Setup the Project

1. Download and unzip the latest S#arp Lite package from <https://github.com/codai/Sharp-Lite/downloads>
2. Copy “\SharpLite\Template\s#arp-lite-project-v0.42.01.pkg” to “C:\Users\%USER%\AppData\Roaming\Endjin\Templify\repo\” (Note that the AppData folder is hidden by default.)
3. In Windows Explorer, create a new, empty folder for the project called “CaTS”
4. Right click the new folder and click "Templify Here"
5. In the Templify dialog:
   * Select the template "S#arp Lite Project - #"
   * Enter the name of your project, which is “CaTS”
   * Click "Deploy Template" (should take well less than a minute to run)
   * Exit Templify when it completes
6. Open the folder and then open the newly created solution
7. Create a database in SQL Server named “CaTS”
8. Update the connection string in the following two files to refer to the CaTS database:
   * YourProject.Web/Web.config
   * YourProject.Tests/App.config
9. Press F5 to be presented with a tremendously exciting “Home” screen.

At this point, the beginnings of the CaTS project have been established. If you haven’t already, refer to <http://devlicio.us/blogs/billy_mccafferty/archive/2011/11/11/s-arp-lite-the-basicss.aspx> to familiarize yourself with the folders generated on the file system along with the projects found within the VS solution.

###### Establish Look & Feel

As awesome as the homepage is already, we’re going to want to establish a look & feel for our site to reduce laughter when people see it. A little searching and I found a reasonably nice one at <http://freehtml5templates.com/kitesurf-html5-and-css3-template/>. As all templates will vary, I won’t go through the details of how the template was installed; you can look at the example code to see the finished product. From the example project, you’ll need to copy /Content/Site.css, /Content/Images/\*, and /Views/Shared/\_Layout.cshtml to your CaTS project. An example of the final look & feel is shown at right.

As a side note, each project that your team delivers should have a theme demo page in the project to show all team members how to use the theme on their forms and content. Taking this a bit further, the project team should establish UI guidelines for the project, to ensure a consistent look & feel.

###### Implement Basic Security Infrastructure

The codebase has been established, the look & feel has been applied; the only remaining infrastructural setup concern is that of security. It is important to implement the security infrastructure as soon as enough information has been gathered to determine what kind of security is necessary to support the requirements of the project. The longer you wait to implement security, the more retrofitting you’ll have to do down the road to put it in.

The requirements have only indicated three types of staff members so far: Admin, Manager and Support Staff. Accordingly, basic Forms Authentication with roles will support the requirements and should scale as needed. The ASP.NET Membership and Roles Providers may be used but I find the approach overcomplicated for simple models and not very scalable to more sophisticated scenarios. Accordingly, a simple alternative that should work nicely for our needs may be found at <http://stackoverflow.com/questions/4837103/asp-net-mvc-alternative-to-role-provider>. We’ll adapt the ideas in this article to the project at hand.

If the security requirements were anticipated to be quite complex, we might instead consider leveraging a more robust security solution, such as Rhino Security. (A guide for using Rhino Security on S#arp Architecture projects may be found at <http://devlicio.us/blogs/billy_mccafferty/archive/2009/04/30/adding-rhino-security-support-to-s-arp-projects.aspx> - the steps would only require a few small changes to be applicable to S#arp Lite projects.)

To implement the security infrastructure within the Acme CaTS application, we’ll walk-through three steps:

* Adding the staff member entity,
* Adding support for authentication (logging in and out), and
* Adding support for authorization (checking permissions).

Add the Staff Member Entity

Where to start? An appropriate place is with the StaffMember class. The security infrastructure exists to support authentication and authorization of StaffMember instances. Accordingly, this is a reasonable cornerstone to put in place.

1. Within CaTS.Domain, add the class StaffMember.cs to the root namespace as follows:  
     
   As you can see, the StaffMember class inherits from Entity. This tells the S#arp Lite framework that this object gets persisted to the database. Additionally, we’ve added validation and display attributes. One could argue that they should only be added when needed; they’re certainly not necessary for the security infrastructure. With that said, there’s no better time to make sure validation and attributes get applied properly than the moment you add a new domain object to the project.   
     
   A few other items to note:  
   • The HasUniqueDomainSignature is a custom validator, available in SharpLite.Domain.Validators which raises a validation issue if an instance of the object already exists in the database having the same “domain signature.” Note that the only property marked as being part of the domain signature is EmployeeNumber; i.e., the HasUniqueDomainSignature will only take the EmployeeNumber into consideration when looking for a conflicting match in the database. The DomainSignature attribute may be placed on multiple properties to use them in conjunction for validating uniqueness. For example, if you didn’t have EmployeeNumber, you could put HasUniqueDomainSignature over both FirstName and LastName to validate uniqueness for those two properties together.  
   • Each property includes the virtual keyword to facilitate NHibernate proxying.  
     
   What about a unit test you say? You could certainly write one for this, but I don’t find much value in writing unit tests for simple property accessors and validation attributes. I find those kinds of unit simply add extra maintenance without adding much value.

using System.ComponentModel.DataAnnotations;

using SharpLite.Domain;

using SharpLite.Domain.Validators;

namespace CaTS.Domain

{

[HasUniqueDomainSignature(

ErrorMessage="A staff member already exists with the same employee number")]

public class StaffMember : Entity

{

[DomainSignature]

[Required(ErrorMessage = "Employee number must be provided")]

[StringLength(6, MinimumLength = 6,

ErrorMessage = "Employee number must be 6 characters long")]

[Display(Name = "Employee Number")]

public virtual string EmployeeNumber { get; set; }

[Required(ErrorMessage = "First name must be provided")]

[StringLength(50, ErrorMessage = "First name must be 50 characters or fewer")]

[Display(Name = "First Name")]

public virtual string FirstName { get; set; }

[Required(ErrorMessage = "Last name must be provided")]

[StringLength(50, ErrorMessage = "Last name must be 50 characters or fewer")]

[Display(Name = "Last Name")]

public virtual string LastName { get; set; }

}

}

1. Compile and run all the unit tests, the CanConfirmDatabaseMatchesMappings unit test fails due to a failed query against the table StaffMembers. This makes sense since we haven’t created that table. We should do that now as breaking unit tests give me the jitters.
   1. Within NUnit, run the unit test CanGenerateDatabaseSchema, go to the “Text Output” tab and run the generated SQL against your CaTS database. (Alternatively, the unit test also dumps the SQL out to \app\CaTS.DB\schema\UnitTestGeneratedSchema.sql.)
   2. Run all of the unit tests and all should pass successfully.

Add Support for Authentication

The next logical piece is to support the ability for a staff member to login and logout of the application. This will require that the staff members have a password with which to login. As is best practice, if you’re storing passwords in the database, they should be hashed (irreversibly encrypted). Accordingly, we’ll start by adding a password to the staff member and a hasher to hash passwords.

[Required(ErrorMessage = "PasswordHash must be provided")]

public virtual string PasswordHash { get; set; }

1. To the bottom of the class StaffMember.cs, add the following property to support the password:
2. Within NUnit, run the unit test CanGenerateDatabaseSchema, go to the “Text Output” tab and run the generated SQL against your CaTS database to have the PasswordHash column added to the StaffMembers table. Run the tests again to see them pass.
3. In test-driven fashion, add a unit test to demonstrate and confirm how password hashing will be performed:
   1. Within CaTS.Tests, add references to the project CaTS.Domain and to \lib\SharpLite.Domain.dll

using CaTS.Domain.Utilities;

using NUnit.Framework;

namespace CaTS.Tests.Domain.Utilities

{

[TestFixture]

public class HasherTests

{

[Test]

public void CanCompareHashedStrings() {

var hash1 = Hasher.Hash("dog");

var hash2 = Hasher.Hash("cat");

var hash3 = Hasher.Hash("dog");

Assert.That(! hash1.Equals(hash2));

Assert.That(hash1.Equals(hash3));

}

}

}

* 1. Add the test class CaTS.Tests/Domain/Utilities/HasherTests.cs:  
       
     This unit test states that there should be a Hasher class within the CaTS.Domain.Utilities namespace and that it exposes a single method called Hash. Can’t compile can ya? That’s OK, that’s what test-driven development is all about; now let’s add the code to make this test pass.

1. Within CaTS.Domain, add a folder called Utilities. Some argue that a utilities folder ends up looking like a junk drawer. But there’s a reason everyone has a junk drawer…it’s a handy spot to put those classes which are useful in various contexts but don’t have a clear business-specific context; e.g., StringUtils, EmailSender, etc. But like your junk drawer at home, try to keep it clean. If a class can be put into a business-specific namespace, favor that over the utilities folder.
2. Within CaTS.Domain, add the class Utilities/Hasher.cs:  
     
   Note that the hash salt helps to avoid hacks using rainbow tables; accordingly, the salt should be unique, private, and non-trivial.

using System.Security.Cryptography;

using System.Text;

namespace CaTS.Domain.Utilities

{

public class Hasher

{

public static string Hash(string value) {

var addSalt = string.Concat(HASH\_SALT, value);

var sha1Hashser = new SHA1CryptoServiceProvider();

var hashedBytes = sha1Hashser.ComputeHash(Encoding.Unicode.GetBytes(addSalt));

return new UnicodeEncoding().GetString(hashedBytes);

}

private const string HASH\_SALT = "I^>cI'}7hgIdKlCLY2%:";

}

}

1. Ultimately, the staff member is going to need to login. To encapsulate the requirements of the login information, we’ll introduce a view model which will be passed to the controller, containing the data from the login form. Within CaTS.Tasks, add the class /ViewModels/LoginViewModel .cs:

using System.ComponentModel.DataAnnotations;

using CaTS.Domain.Utilities;

namespace CaTS.Tasks.ViewModels

{

public class LoginViewModel

{

[Required(ErrorMessage = "Employee number must be provided")]

[Display(Name = "Employee Number")]

public string EmployeeNumber { get; set; }

[Required(ErrorMessage = "Password must be provided")]

public string Password { get; set; }

public string ReturnUrl { get; set; }

public string GetPasswordHash() {

return Hasher.Hash(Password);

}

}

}

1. We now need a means to handle logging-in and logging-out, including both the controller and the view:
   1. Copy \SharpLite\Example\CaTS\Day4\app\CaTS.Web\Controllers\AuthenticationController.cs to the respective location in your CaTS solution (and include in the VS project). This controller handles the initial request to the login page, handling of the post back from the login form, and logging out. Arguably, this controller does more logic than I’d typically like to see in a controller. But since logging-in is tightly coupled to the HTTP context, the controller serves as an efficient means to encapsulate all of this logic. Alternatively, you could introduce an AuthenticationTasks class within the tasks layer to encapsulate the logic; but you’d end up having to introduce a reference to System.Web to the tasks layer as well…something I prefer to avoid if at all possible.
   2. Copy \SharpLite\Example\CaTS\Day4\app\CaTS.Web\Views\Authentication\Login.cshtml to your CaTS solution (and include in the VS project). This is the login form to enter the employee number and password.

<authentication mode="Forms">

<forms loginUrl="Authentication/Login" name=".CaTSAuthentication"

protection="All" defaultUrl="/" timeout="60"></forms>

</authentication>

1. Within CaTS.Web, modify the authentication tag in the web.config to the following:
2. We’re now ready to test the login page functionality. In order to do so, an initial user needs to be added to the database. The following script will add a staff member with a password of “p@ssw0rd” already hashed:

INSERT INTO StaffMembers VALUES ('8493YUI', 'John', 'Doe', N'汹누톫꺒恩投홀ᵬ鉷')

1. Within VS, run the application with F5 and login with 8493YUI / p@ssw0rd. After logging in, you should see a “Logout” button appear in the menu which you can click to…well…logout.

Add Support for Authorization

Authorization will include two components. The first will be a means to assign a staff member to one or more roles. The second will be a means to restrict access to controller actions.

1. We’ll begin by adding support for staff member role assignments. Add a unit test to demonstrate (and confirm) how authorization will work within CaTS:
   1. In test-driven fashion, add the class AuthorizationTests.cs to the CaTS.Tests/Domain folder:  
        
      The class contains two tests: the first for verifying the authorization of a staff member having a single role, and the second for verifying the authorization of a staff member having two roles.

using System;

namespace CaTS.Domain

{

/// <remarks>All powers of 2 so we can OR them to combine role permissions.</remarks>

[Flags]

public enum RoleType

{

Administrator = 1,

Manager = 2,

SupportStaff = 4

}

}

using NUnit.Framework;

using CaTS.Domain;

namespace CaTS.Tests.Domain

{

[TestFixture]

public class AuthorizationTests

{

[Test]

public void CanAuthorizeStaffMemberWithSingleRole() {

var staffMember = new StaffMember {

Roles = RoleType.Administrator

};

Assert.That(staffMember.IsAuthorizedAs(RoleType.Administrator));

Assert.That(! staffMember.IsAuthorizedAs(RoleType.SupportStaff));

}

[Test]

public void CanAuthorizeStaffMemberWithMultipleRoles() {

var staffMember = new StaffMember {

Roles = RoleType.Manager | RoleType.SupportStaff

};

Assert.That(staffMember.IsAuthorizedAs(RoleType.Manager));

Assert.That(staffMember.IsAuthorizedAs(RoleType.SupportStaff));

Assert.That(! staffMember.IsAuthorizedAs(RoleType.Administrator));

}

}

}

* 1. Within CaTS.Domain, add the class RoleType.cs to the root namespace as follows:  
       
     This roles enum allows us to perform a bitwise OR to set the authorized roles and allows us to perform a bitwise AND to check for the roles, as described next.
  2. To the bottom of the class StaffMember.cs, add the following property and method to fulfill the remainder of the test requirements:  
       
     One could argue that separate authorization classes should be introduced, as the StackOverflow article suggested, in order to cleanly separate the authorization check from the staff member object. I would argue that we should keep things as simple as possible until introducing further complexity is warranted.

/// <summary>

/// Represents one or more roles that the staff member has been assigned.

/// E.g., set property to RoleType.Manager for one role and

/// (RoleType.Manager | RoleType.SupportStaff) for two roles.

/// </summary>

public virtual RoleType Roles { get; set; }

public virtual bool IsAuthorizedAs(RoleType requiredRoles) {

// Check if the roles enum has the specific role bit set

return (requiredRoles & Roles) == requiredRoles;

}

* 1. Within SQL, add the column “Roles” to the StaffMembers table as a nullable int.
  2. Compile and run all the unit tests …go green!

1. Now that the domain model for supporting authorization is complete, we need a controller attribute to restrict access to particular actions.
   1. Copy \SharpLite\Example\CaTS\Day4\app\CaTS.Web\Controllers\Attributes\RequireRoleAttribute.cs to your CaTS solution (and include in the VS project).

[RequireRole(RoleType.Administrator)]

public ActionResult AdminPage() {

return View();

}

* 1. Add a controller action to HomeController.cs to see the attribute in action:  
       
     As implied, this action will only be viewable to staff members in the admin role. All others will get redirected to the login page as they will not be authorized to view it.
  2. Add the web page CaTS.Web/Views/Home/AdminPage.cshtml which will be shown to staff members in the admin role:

@{

ViewBag.Title = "Home";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<p>

You are an admin!

</p>

* 1. Now, run the project, login as the staff member, and browse to /Home/AdminPage …no go. Within SQL, update the Roles column of the staff member to be 1 to assign the staff member to the admin role, or 3 to assign the staff member to both the admin and manager roles. You should now have no problem browsing to /Home/AdminPage.

With this, we now have support for both forms authentication and authorization; i.e., the security infrastructure is now in place.

## Day 5 – Develop CRUD and Data Management Capabilities

With the initial project setup, we’re now ready to move on to inclusion of our CRUD and data management capabilities.

### Objective of the Day

Establish all CRUD related capabilities of the system for simple and complex data types.

### Activities

The day’s activities represent the meat of the application, providing all of the capabilities necessary to manage the system’s data. We’ll examine two data management scenarios in detail. Firstly, we’ll detail the steps necessary to add support to manage customer details, a relatively simple data type in the CaTS application. We’ll also detail the steps necessary to add capabilities for creating support tickets. This more complex and interesting scenario will require non-trivial CRUD capabilities, including AJAX communications.

All other data management capabilities may be examined by looking at the resulting code found in \SharpLite\Example\CaTS\Day5.

##### Manage Customers

This task will focus on implementing the code necessary to fulfill the following user story:

* Support Staff may manage Customer details, adding and updating Customer information when necessary.

###### Add the Domain Model

1. One item not previously mentioned is the inclusion of an email address for the Customer. We already had AccountNumber, FirstName, and LastName, but we’ve discovered during implementation that we neglected a means to contact the Customer. Consequently, we’ll want to include an email address. (Just like in a real project, it should be expected to discover items such as this along the way. No need to panic and spend days updating all the documentation; simply accommodate the change and move on.) S#arp Lite uses .NET provided data annotations to provide validation of fields; unfortunately, the .NET data annotations are a bit simplistic out of the box. So to validate an email address you can write a custom RegEx validator…or…install the data annotation extensions for MVC 3. (You can read more about these extensions at <http://dataannotationsextensions.org/>.)
   1. Within CaTS.Domain, right click references and click “Manage NuGet Packages”
   2. Within the package manager, click the “Online” panel and then search for “DataAnnotationsExtensions.” You should get two results. The one without the “.MVC3” suffix provides server-side support. The one with the “.MVC3” suffix provides integrated server-side/client-side validation suing unobtrusive jQuery validation.
   3. Select DataAnnotationsExtensions.MVC3, click Install, and then Close when complete.

using System.ComponentModel.DataAnnotations;

using DataAnnotationsExtensions;

using SharpLite.Domain;

using SharpLite.Domain.Validators;

namespace CaTS.Domain

{

[HasUniqueDomainSignature(

ErrorMessage = "A customer already exists with the provided account number")]

public class Customer : Entity

{

[DomainSignature]

[Required(ErrorMessage = "Account number must be provided")]

[StringLength(8, MinimumLength = 8,

ErrorMessage = "Account number must be exactly 8 characters")]

[Display(Name = "Account Number")]

public virtual string AccountNumber { get; set; }

[Required(ErrorMessage = "First name must be provided")]

[StringLength(200,

ErrorMessage = "First name must be 200 characters or fewer")]

[Display(Name = "First Name")]

public virtual string FirstName { get; set; }

[Required(ErrorMessage = "Last name must be provided")]

[StringLength(200,

ErrorMessage = "Last name must be 200 characters or fewer")]

[Display(Name = "Last Name")]

public virtual string LastName { get; set; }

[Email]

[Required(ErrorMessage = "Email address must be provided")]

[StringLength(200,

ErrorMessage = "Email must be 200 characters or fewer")]

[Display(Name = "Email")]

public virtual string EmailAddress { get; set; }

}

}

1. Within CaTS.Domain, add the class Customer.cs to the root namespace as follows:
2. Compile and run the CanGenerateDatabaseSchema and run the corresponding “create table Customers” SQL against your CaTS database. (As a reminder, in NUnit, the SQL will show up under the “Text Output” tab.)
3. All unit tests should now be passing.

###### Add the CRUD Components

1. Within CaTS.Tasks, add the view model class /ViewModels/EditCustomerViewModel.cs as follows:   
     
   This DTO will provide the means to send data to the edit form. Because a customer instance is quite simplistic – having only primitive data type properties to be populated via text boxes – the view model will simply carry the customer to the form. But suppose a dynamically populated dropdown list was to be offered in addition to the text boxes, then the view model could also carry information that would be used to populate that dropdown. We’ll see this in action when we get to adding support for creating a support ticket.

<div id="nav">

<li class="current">

@Html.ActionLink("Home", "Index", "Home", new { area = "" }, null)

</li>

@if (HttpContext.Current.User.Identity.IsAuthenticated) {

<li class="current">

@Html.ActionLink("Customers", "Index", "Customers", new { area = "" }, null)

</li>

<li class="current">

@Html.ActionLink("Logout", "Logout", "Authentication", new { area = "" }, null)

</li>

}

</div>

1. Within CaTS.Tasks, add the class CustomerCudTasks to the root namespace as follows:  
     
   This tasks class represents the meat of everything necessary to support basic CUD capabilities (CRUD without the Read) for instances of the Customer object. Granted, the Customer class is a very simple class; accordingly, we can take advantage of using the BaseEntityCudTasks to hide away the duplicated code involved with CUDing a simple object. The only real requirement is that your view model includes a property name that’s the same as the entity type that you’re managing; e.g., a Customer property called Customer.  
     
   Admittedly, BaseEntityCudTasks is not a trivial base object, but it’s simple to leverage for basic CUD needs. Additionally, if your CUD logic differs for that provided by BaseEntityCudTasks, simply change it to meet your needs. We’ll see shortly that BaseEntityCudTasks isn’t very useful for more complex CUD scenarios, but it keeps things simple when appropriate.

using CaTS.Domain;

using CaTS.Tasks.ViewModels;

using SharpLite.Domain.DataInterfaces;

namespace CaTS.Tasks

{

public class CustomerCudTasks : BaseEntityCudTasks<Customer, EditCustomerViewModel>

{

public CustomerCudTasks(IRepository<Customer> customerRepository)

: base(customerRepository) {

}

protected override void TransferFormValuesTo(Customer toUpdate, Customer fromForm){

toUpdate.AccountNumber = (fromForm.AccountNumber ?? "").Trim();

toUpdate.EmailAddress = (fromForm.EmailAddress ?? "").Trim();

toUpdate.AccountNumber = (fromForm.AccountNumber ?? "").Trim();

toUpdate.AccountNumber = (fromForm.AccountNumber ?? "").Trim();

}

}

}

1. We now need to add the controller and web pages to present the customer listing page along with the add/update customer page. To save space within this document, let’s copy files from the example code to your solution:
   1. Copy \SharpLite\Example\CaTS\Day5\app\CaTS.Web\Controllers\ManageCustomersController.cs to the respective location in your CaTS solution (and include in the VS project). This controller handles all requests raised by the web pages, handing off the heavy lifting to the CustomerCudTasks. A key item to note is within the Index action method. You’ll see that the controller invokes the repository directly, instead of handing off this simple query to the tasks layer. While the query could certainly be moved to the tasks layer to introduce a spic and span, clean separation of concerns, it would only add a layer of indirection for an otherwise trivial query. Later, we’ll see how more complex queries can be added to the domain layer for better encapsulation and reuse.
   2. Copy \SharpLite\Example\CaTS\Day5\app\CaTS.Web\Views\ManageCustomers\\*.cshtml to your CaTS solution (and include in the VS project). These are the UI pages for management of the Customers, including:
      * Index.cshtml: List the customers in the database. Note that the delete button is wrapped in a form with an AntiForgeryToken to add better deletion protection.
      * Details.cshtml: Show the details of a specific customer.
      * Edit.cshtml: Provide a form to add a new customer or update an existing one.
2. Within CaTS.Web, modify the navigation section in /Views/Shared\_Layout.cshtml to reflect the following to include a menu link to the customer listing page:

<div id="nav">

<li class="current">

@Html.ActionLink("Home", "Index", "Home", new { area = "" }, null)

</li>

@if (HttpContext.Current.User.Identity.IsAuthenticated) {

<li class="current">

@Html.ActionLink("Customers", "Index", "ManageCustomers",   
 new { area = "" }, null)

</li>

<li class="current">

@Html.ActionLink("Logout", "Logout", "Authentication", new { area = "" }, null)

</li>

}

</div>

1. With this in place, simply F5 to run the application and you’ll be on your way with managing customers.

While you’re at it, try creating two customers with the same account number to activate the HasUniqueDomainSignature validation attribute. A subtle note to make is that validation of HasUniqueDomainSignature occurs on the server-side since it has to make a call to the database, while the data annotations validators were enforced on the client-side.

As you can see, adding CRUD support for trivial objects is just that, trivial. At a high level, the CRUD process works as follows for each step of the process, with caveats included for when this simplistic approach begins to break down:

* For the listing page, simply pass a listing of the entities as the model. This approach breaks down if each row of the listing needs to include summary information; e.g., the number of open support tickets per customer. This could run into the select n+1 problem.[[5]](#endnote-5) We’ll take a look shortly at a nifty solution for this using LINQ.
* For the details page, simply pass the entity as the model. Even for complex objects, this is usually the simplest approach. If you have other data to pass to the model, such as security related information, simply wrap the entity within a view model containing properties for the other data as well; similar in nature to the EditCustomerViewModel seen earlier.
* For the edit page, pass a view model containing a reference to the entity.
* For collecting data posted from the form, allow the model binder to bind directly to the entity class, treating an instance of the entity class as a DTO for what will be transferred to the persistent instance. This approach breaks down in more complex scenarios, such as when the form contains inputs associated with different objects. If you start feeling like binding to an entity (e.g., Customer) is becoming unwieldy or “weird,” then favor binding instead to a view model to act as a DTO for the data from the form.

We’ll review a number of techniques in the next section, and the coming day, which may be leveraged to tackle more complex scenarios.

##### Create New Support Ticket

This task will focus on implementing the code necessary to fulfill the following user story:

* Support Staff may open a new support ticket and provide details including: Customer, Issue Details (such as description and “dynamically managed” type), Status, and Resolution Details (if resolved immediately). Status of the call may be New, In Progress, or Resolved. (“Dynamically managed” means that if the issue type isn’t available, the Support Staff may enter a new one when entering the support ticket.)

For this user story, we’ve already done quite a bit of preparation work. On day 3, we created the “Sequence Diagram for New Support Ticket” and detailed all of the anticipated classes that would be needed to support this user story. Additionally, we detailed the customer selection process with the “Sequence for Select Customer Sub Process.” Now, it’s “simply” a matter of implementing the design.

There are quite a few pieces with this one. To expedite the process, we’ll copy files from the example code into your own solution, discussing the more interesting points of each along the way.

###### Add Domain Elements of Support Ticket Management

1. Within CaTS.Domain, create a new namespace called “Support.” This namespace will hold everything related to the management of support tickets.
2. Copy \SharpLite\Example\CaTS\Day5\app\CaTS.Domain\Support\IssueType.cs to the respective location in your project (and include in the VS solution). This is the persistable object to represent the various issue types, as described in the class diagram. Recall that the user story states that a staff member may dynamically add new issue types when creating a support ticket; accordingly, this is a persistable class, deriving from Entity, instead of being an enum.
3. With NUnit, run CanGenerateDatabaseSchema and run the script for “create table IssueTypes” against your DB to get all tests passing.
4. Copy \SharpLite\Example\CaTS\Day5\app\CaTS.Domain\Support\StatusType.cs to the respective location in your project (and include in the VS solution). This is an enum representing the various status states of a support ticket, as described in the class diagram.  
   The SupportTicket class, which we’ll be adding shortly, will reference this enum as a property. If we want the int value of the enum stored in the DB (which I personally prefer), then we need to add an NHibernate custom type to inform NHibernate how to convert an int to the StatusType enum and vice-versa. Accordingly…
5. Copy \SharpLite\Example\CaTS\Day5\app\CaTS.NHibernateProvider\Overrides\StatusCustomType.cs to the respective location in your project (and include in the VS solution). This is the custom type that we’ll use to inform NHibernate how to perform the int/StatusType-enum conversion. We’ll use it shortly.
6. Copy \SharpLite\Example\CaTS\Day5\app\CaTS.Domain\Support\SupportTicket.cs to your project. This self-explanatory class has one interesting thing to note: it does not include a HasUniqueDomainSignature. This means that you can have multiple support tickets in the database with identical information. While this will not be likely, there doesn’t appear to be a business reason to prevent this. (Your client may disagree…I would advise that you side with your client if that’s the case.)
7. Copy \SharpLite\Example\CaTS\Day5\app\CaTS.NHibernateProvider\Overrides\SupportTicketOverride.cs to your project. This override is necessary to inform NHibernate that SupportTicket’s Status property should use the custom type, previously included when storing and retrieving that particular property. Furthermore, the override indicates the exact name of the column to store the value in. Notice that we didn’t have to override every property, just the one(s) needing to be overridden.
8. With NUnit, run CanGenerateDatabaseSchema and run the DB scripts related to SupportTickets, including the creation of the table and the three foreign keys. When transposing the SQL over to the SQL Server 2008, I changed “IssueDescription NVARCHAR(255) null” to “IssueDescription text null” to allow a much larger text block to be stored. Accordingly, feel free to change the auto-generated SQL as needed to meet your standards and preferences. (In the near future, I hope to modify S#arp Lite to support “migration scripts” similar to what Ruby on Rails supports.)

At this point, we now have all of the domain layer elements in place for management of support tickets.

###### Add Task Elements for Support Ticket Management

Now that we have the domain elements in place for support tickets, we need to head up to the CaTS.Tasks layer which will coordinate the logic for support ticket creation.

Thus far, we’ve been relatively light on unit tests. On the other hand, we haven’t really done anything that would warrant much unit testing. In real-world projects, I take three stances to unit testing which you are welcome to take with a grain of salt:

* *No testing of trivial code.* For some code, I don’t write any unit tests at all. For example, I don’t see any point in writing unit tests to verify that property accessors are working as expected.
* *Little testing of HTTP-contextual code.* I rarely write unit tests for controllers. My arguments for this are as follows:
  + The controllers should be doing very little to begin with, so if you find yourself needing to write unit tests for them, perhaps you’re making your controllers take on responsibilities which should be moved to the .Tasks layer, and
  + I find that the complexity of writing and maintaining HTTP-contextual test code is rarely justified by the “paltry” testing results.
* *Strong testing of complex logic or logic with many steps.* Whenever you find yourself with a non-trivial method or sequence of events, verify it with a unit test. QED.
* *Strong testing of hard-to-learn modules.* Unit tests are frequently helpful to simply explain how a module works. Indeed, there’s no more accurate documentation than a working unit tests. If you’d like to get a developer up to speed on a complicated module, have them study the unit tests.
* *Test-driven development of areas which are difficult to design.* Sometimes, sequence diagrams just aren’t enough. Sometimes you’re faced with a coding challenge that is difficult to figure out with pen and paper. Accordingly, I’ll frequently turn to writing a unit test first in these cases to help me figure out what the API should look like. I find that this approach keeps the resulting API much cleaner and simple than if I had instead taken a “get ‘er done!” attitude. (*You* know what I’m talking about here! ;)

## Day 6 – Provide Reporting Capabilities

### Objective of the Day

Lorem ipsum

### Inputs

Lorem ipsum

### Activities

Lorem ipsum

### Outputs

Lorem ipsum

# Part III – Integrating with External Services

## Day 7 – Integrate with External Services

### Objective of the Day

Lorem ipsum

### Inputs

Lorem ipsum

### Activities

Lorem ipsum

### Outputs

Lorem ipsum

# References

[Evans 2003] Domain-Driven Design: Tackling Complexity in the Heart of Software.

[Fowler 1996] Analysis Patterns: Reusable Object Models.

[Fowler 2002] Patterns of Enterprise Application Architecture.

[Galloway 2011] Professional ASP.NET MVC 3.

[GoF 1995] Design Patterns: Elements of Reusable Object-Oriented Software.

[Larman 2004] Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.

[Pilone 2006] UML 2.0 Pocket Reference.

[Troelsen 2010] Pro C# 2010 and the .NET Platform.

1. See [Evans 2003] for the definitive Domain-Driven Design reference or InfoQ’s summary at <http://www.infoq.com/minibooks/domain-driven-design-quickly> [↑](#endnote-ref-1)
2. See Beck or xUnit Patterns [↑](#endnote-ref-2)
3. Bounded Contexts are nicely explained at <http://devlicio.us/blogs/casey/archive/2009/02/11/ddd-bounded-contexts.aspx> and discussed in more detail in [Evans 2003] [↑](#endnote-ref-3)
4. CRC cards, invented by Kent Beck and Ward Cunningham, are briefly described at <http://en.wikipedia.org/wiki/Class-responsibility-collaboration_card> and in more detail at <http://c2.com/doc/oopsla89/paper.html> [↑](#endnote-ref-4)
5. See <http://ayende.com/blog/3732/solving-the-select-n-1-problem> for a good description of the “select n+1” problem. (An alternative solution to the problem is detailed within this document using LINQ queries.) [↑](#endnote-ref-5)