CONTENTS
» About Domain-Driven Design
» Representing the Model
» Ubiquitous Language
» Strategic Design
» Modeling the Domain... and more!

GETTING STARTED WITH

Domain-Driven Design

BY ASLAM KHAN, UPDATED AND REVISED BY OBI OBEROI

ABOUT DOMAIN-DRIVEN DESIGN

This is a quick reference for the key concepts, techniques, and patterns described in detail in Eric Evans' book *Domain Driven Design: Tackling Complexity in the Heart of Software* and Jimmy Nilsson's book *Applying Domain-Driven Design and Patterns with Examples in C# .NET*. In some cases, it has made sense to use the wording from these books directly, and I thank Eric Evans and Jimmy Nilsson for giving permission for such usage.

While it is useful to present the patterns themselves, many subtleties of DDD are lost in just the description of the patterns. These patterns are your tools, and not the rules. They are a language for design and useful for communicating ideas and models amongst the team. More importantly, remember that DDD is about making pragmatic decisions. Try not to "force" a pattern into the model, and, if you do "break" a pattern, be sure to understand the reasons and communicate that reasoning too.

Often, it is said that DDD is object orientation done right, but DDD is a lot more than just object orientation. DDD also deals with the challenges of understanding a problem space and the even bigger challenge of communicating that understanding.

Importantly, DDD also encourages the inclusion of other areas such as Test-Driven Development (TDD), usage of patterns, and continuous refactoring.

REPRESENTING THE MODEL

Domain-Driven Design is all about design and creating highly expressive models. DDD also aims to create models that are understandable by everyone involved in the software development, not just software developers.

Since non-technical people also work with these models, it is convenient if the models can be represented in different ways. Typically, a model of a domain can be depicted as a UML sketch, as code, and in the language of the domain.

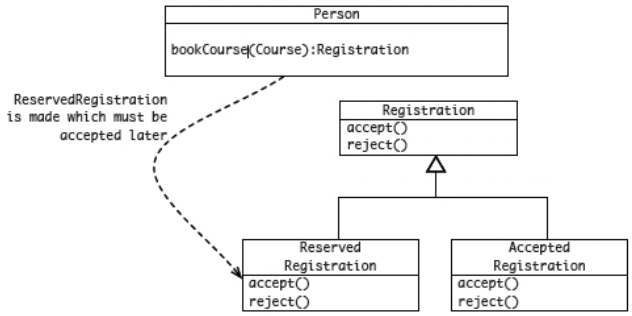
USING LANGUAGE

A person that is looking at attending a training course searches for courses based on topic, cost, and the course schedule. When a course is booked, a registration is issued, which the person can cancel or accept at a later date.

USING CODE

```
class Person {
   public Registration bookCourse(Course c) { ... }
}

abstract class Registration {
   public abstract void accept();
   public abstract void cancel();
}

class ReservedRegistration extends Registration { ... }
class AcceptedRegistration extends Registration { ... }
interface CourseRepository {
   public List<Course> find(...);
}
```

USING A UML SKETCH



UBIQUITOUS LANGUAGE

The consistent use of unambiguous language is essential in understanding and communicating insights discovered in the domain. In DDD, it is less about the nouns and verbs and more about the concepts. It is the intention of the concept, its significance and value that is important to understand and convey. How that intention is implemented is valuable, but for every intention, there are many implementations. Everyone must use the language everywhere and at every opportunity to understand and share these concepts and intentions. When you work with a ubiquitous language, the collaboration with domain experts is more creative and valuable for everyone.

Watch out for technical and business obstructions in the language that may obscure vital concepts hidden or assumed by domain experts. Often these terms deal with implementations, and not the domain concepts. DDD does not exclude the implementation, but it values the intention of the higher model.

Consider the following conversation:

When a person books a course, and the course is full, then the person has a status of "waiting". If there was space available, then the person's details must be sent via our message bus for processing by the payment gateway.

Here are some potential obstructions for the above conversation. These terms don't add value but they are excellent clues to dig deeper into the domain.

person has a status	Status seems to be a flag or field. Perhaps the domain expert is familiar with some other system, maybe a spreadsheet, and is suggesting this implementation.
sent via our message bus	This is a technical implementation. The fact that it is sent via a message bus is of no consequence in the domain.
processing	This is ambiguous and obscure. What happens during processing?
payment gateway	Another implementation. It is more important that there is some form of payment but the implementation of the payment is insignificant at this point.

AIM FOR DEEP INSIGHTS

Keep a watch out for implementations and dig around for the real concepts and the intention of the concepts.

Let's review the same conversation, paying attention to clues that may be hidden in the conversation, behind some of the implementations.

*When a person books a course, and the course is full, then the **person has a status** of "waiting." If there was space available, then the person's details must be sent via our message bus for processing by the payment gateway.*

Digging deeper, we find that the person booking the course does not have a status. Instead, the outcome of a person registering for the course is a registration. If the course is full, then the person has a standby registration. All standby registrations are managed on a waiting list.

REFACTOR THE LANGUAGE

Remember that the language is used to build a representation of the model of the domain. So is the code. When the code is refactored, refactor your language to incorporate the new term. Ensure that the concept represented by the term is defined and that domain experts agree with its intention and usage.

Let's refactor the conversation to book a course.

When a person registers for a course, a reserved registration is issued. If there is a seat available, and payment has been received, then the reserved registration is accepted. If there are no seats available on the course, then the reserved registration is placed on a waiting list as a standby registration. The waiting list is managed on a first-come, first-served basis.

WORKING WITH CONCRETE EXAMPLES

It is often easier to collaborate with domain experts using concrete examples. Quite often, it is convenient to describe the domain examples using Behavior-Driven Development (BDD) story and scenario templates (see dannorth.net/whats-in-a-story).

Let's look at the same story from earlier using concrete examples, rephrased using the BDD templates.

Story: Register for a course
As a *person looking for training*
I want *to book a course*
So that *I can learn and improve my skills.*

In the story, a role is described (the "person looking for training") that wishes to achieve something ("to book a course") so that some benefit is gained ("learn and improve my skills").

Now that we have the story, there are many scenarios for that story. Let us consider the scenario of the course being full.

Scenario: Course is full
Given *that the Python 101 course accommodates 10 seats*
and *there are already 10 people with confirmed registrations for Python 101*
When *I register for "Python 101"*
Then *there should be a standby registration for me for Python 101*
and *my standby registration should be on the waiting list.*

The "**Given**" clause describes the circumstances for the scenario. The "**When**" clause is the event that occurs in the scenario, and the "**Then**" clause describes the outcome that should be expected after the event occurs.

STRATEGIC DESIGN

Strategic design is about design that is large and complex that focuses on the many parts that make up the large model, and how these parts relate to each other. This facilitates the design up front, enough to make progress without falling into the "my model is cast in stone" trap.

In DDD, these smaller models reside in bounded contexts. The manner in which these bounded contexts relate to each other is known as context mapping.

BOUNDED CONTEXTS

For each model, deliberately and explicitly define the context in which it exists. There are no rules to creating a context, but it is important that everyone understands the boundary conditions of the context.

Teams that don't have a good understanding of the different contexts within a system, and their relationships to one another, run the risk of compromising the models at play when integrating bounded contexts. Lines between models can become blurred resulting in a Big Ball of Mud if teams don't explicitly map and understand relationships between contexts.

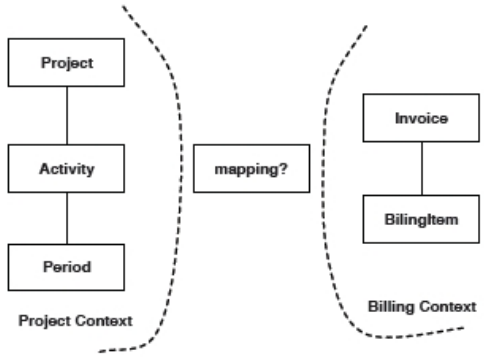
Contexts can be created from (but not limited to) the following:

- how teams are organized
- the structure and layout of the code base
- usage within a specific part of the domain

Aim for consistency and unity inside the context, and don't be distracted by how the model is used outside the context. Other contexts will have different models with different concepts. It is not uncommon for another context to use a different dialect of the domain's ubiquitous language.

CONTEXT MAPS

Context mapping is a design process where the contact points and translations between bounded contexts are explicitly mapped out. Focus on mapping the existing landscape, and deal with the actual transformations later.



HOT TIP

Use continuous integration within a single bounded context to smoothen splinters that arise from different understandings. Frequent code merges, automated tests, and applying the ubiquitous language will highlight fragmentation inside the bounded context quickly.

PATTERNS FOR CONTEXT MAPPING

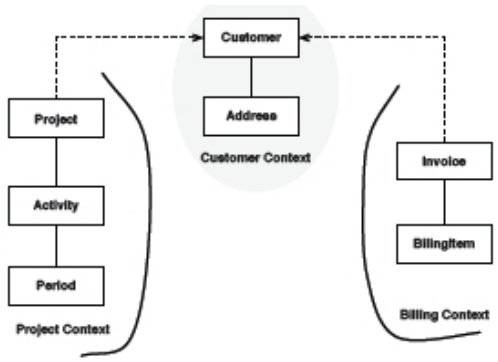
There are several patterns that can be applied during context mapping. Some of these context mapping patterns are explained below.

SHARED KERNEL

This is a bounded context that is a subset of the domain that different teams agree to share. It requires really good communication and collaboration between the teams. Remember that it does not conform to the lowest common denominator.

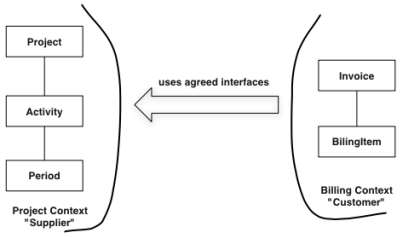
HOT TIP

Be careful with shared kernels! They are difficult to design and maintain and are most effective with highly mature teams!



CUSTOMER/SUPPLIER DEVELOPMENT TEAMS

When one bounded context serves or feeds another bounded context, then the downstream context has a dependency on the upstream context. Knowing which context is upstream and downstream makes the role of supplier (upstream) and customer (downstream) explicit.

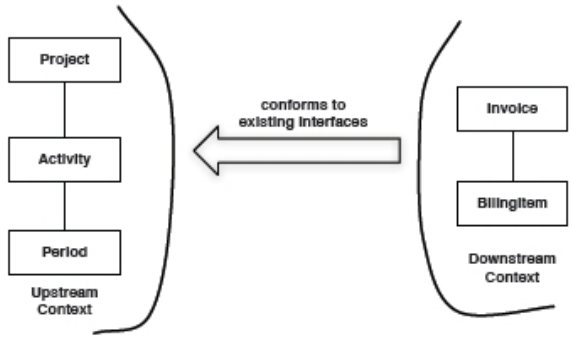


The two teams should jointly develop the acceptance tests for the interfaces and add these tests to the upstream bounded context's continuous integration. This will give the customer team confidence to continue development without fear of incompatibility.

CONFORMIST

When the team working with the downstream context has no influence or opportunity to collaborate with the team working on the upstream context, then there is little option but to conform to the upstream context.

There may be many reasons for the upstream context "dictating" interfaces to the downstream context, but switching to a conformist pattern negates much pain. By simply conforming to the upstream interfaces, the reduction in complexity often outweighs the complexity of trying to change an unchangeable interface.

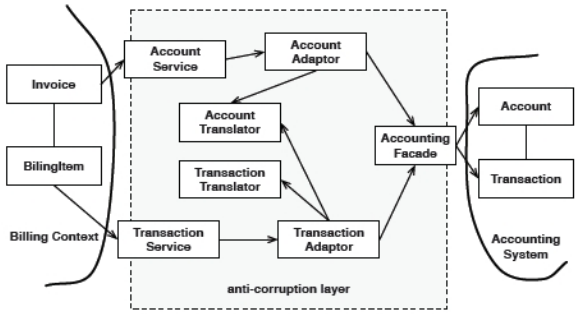


The quality of the downstream model, in general, follows that of the upstream model. If the upstream model is good, then the downstream model is good also. However, if the upstream model is poor, then the downstream will also be poor. Regardless, the upstream model will not be tailored to suit the downstream needs, so it won't be a perfect fit.

The conformist pattern calls for a lot of pragmatism! The quality of the upstream model, along with the fit of the upstream model may be "good enough." That suggests you would not want a context where you were working on the core domain in a conformist relationship.

ANTI-CORRUPTION LAYER

When contexts exist in different systems, and attempts to establish a relationship result in the "bleeding" of one model into the other model, then the intention of both will be lost in the mangled combination of the models from the two contexts. In this case, it is better to keep the two contexts well apart and introduce an isolating layer in between that is responsible for translating in both directions. This anti-corruption layer allows clients to work in terms of their own models.

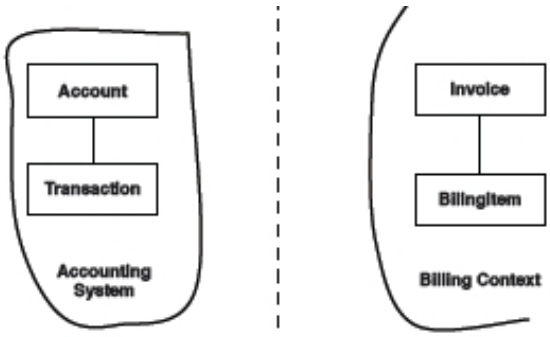


Anti-corruption Layer is a great pattern for dealing with legacy systems or with code bases that will be phased out.

SEPARATE WAYS

Critically analyze the mappings between bounded contexts. If there are no indispensable functional relationships, then keep the contexts separate. The rationale is that integration is costly and can yield very low returns.

This pattern eliminates significant complexity since it allows developers (and even the business managers) to find highly focused solutions in a very limited area of scope.



MODELING THE DOMAIN

Within the bounded contexts, effort is focused on building really expressive models; models that reveal the intention more than the implementation. When this is achieved, concepts in the domain surface naturally and the models are flexible and simpler to refactor.

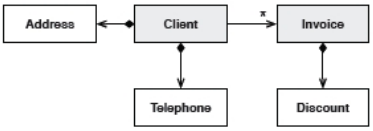
The DDD patterns are more of an application of patterns from GoF, Fowler, and others, specifically in the area of modeling subject domains.

The most common patterns are described here.

DEALING WITH STRUCTURE

ENTITIES

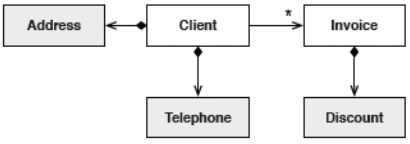
Entities are classes where the instances are globally identifiable and keep the same identity for life. There can be a change of state in other properties, but the identity never changes.



In this example, the Address can change many times but the identity of the Client never changes, no matter how many other properties change state.

VALUE OBJECTS

Value objects are lightweight, immutable objects that have no identity. While their values are more important, they are not simple data transfer objects. Value objects are a good place to put complex calculations, offloading heavy computational logic from entities. They are much easier and safer to compose, and by offloading heavy computational logic from the entities, they help entities focus on their role of life-cycle trackers.

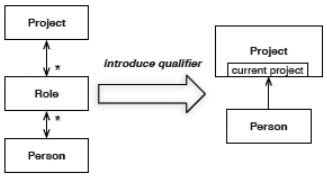


In this example, when the address of the Client changes, then a new Address value object is instantiated and assigned to the Client.

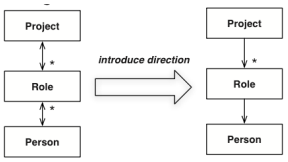
Value Objects have simple life cycles and can greatly simplify your model. They also are great for introducing type safety at compile time for statically typed languages, and since the methods on value objects should be side effect free, they add a bit of functional programming flavor too.

CARDINALITY OF ASSOCIATIONS

The greater the cardinality of associations between classes, the more complex the structure. Aim for lower cardinality by adding qualifiers.



Bi-directional associations also add complexity. Critically ask questions of the model to determine if it is absolutely essential to be able to navigate in both directions between two objects.



In this example, if we rarely need to ask a Person object for all its projects, but we always ask a Project object for all people in the roles of the project, then we can make the associations one directional. Direction is about honoring object associations in the model in memory. If we need to find all Project objects for a Person object, we can use a query in a Repository (see below) to find all Projects for the Person.

SERVICES

Sometimes it is impossible to allocate behavior to any single class, be it an entity or value object. These are cases of pure functionality that act on multiple classes without one single class taking responsibility for the behavior. In such cases, a stateless class, called a service class, is introduced to encapsulate this behavior.

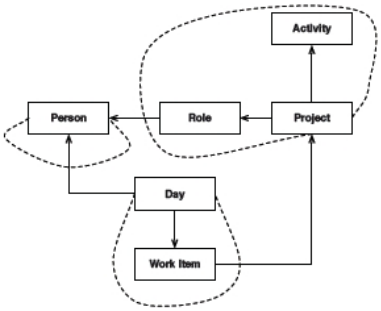
AGGREGATES

As we add more to a model, the object graph can become quite large and complex. Large object graphs make technical implementations such as transaction boundaries, distribution, and concurrency very difficult. Aggregates are consistency boundaries such that the classes inside the boundary are "disconnected" from the rest of the object graph. Each aggregate has one entity which acts as the "root" of the aggregate.

When creating aggregates, ensure that the aggregate is still treated as a unit that is meaningful in the domain. Also, test the correctness of the aggregate boundary by applying the "delete" test. In the delete test, critically check which objects in the aggregate (and outside the aggregate) will also be deleted, if the root was deleted.

Follow these simple rules for aggregates:

- The root has global identity and the others have local identity
- The root checks that all invariants are satisfied
- Entities outside the aggregate only hold references to the root
- Deletes remove everything in the aggregate
- When an object changes, all invariants must be satisfied.



Remember that aggregates serve two purposes: domain simplification, and technical improvements. There can be inconsistencies between aggregates, but all aggregates are eventually consistent with each other.

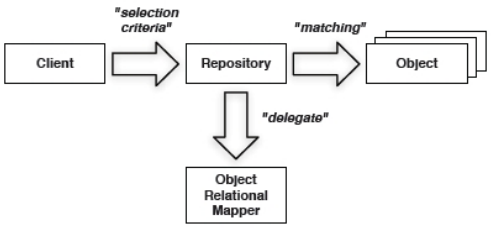
DEALING WITH LIFE CYCLES

FACTORIES

Factories manage the beginning of the life cycle of some aggregates. This is an application of the GoF factory or builder patterns. Care must be taken that the rules of the aggregate are honored, especially invariants within the aggregate. Use factories pragmatically. Remember that factories are sometimes very useful, but not essential.

REPOSITORIES

While factories manage the start of the life cycle, repositories manage the middle and end of the life cycle. Repositories might delegate persistence responsibilities to object-relational mappers for retrieval of objects. Remember that repositories work with aggregates too. So the objects retrieved should honor the aggregate rules.



DEALING WITH BEHAVIOR

SPECIFICATION PATTERN

Use the specification pattern when there is a need to model rules, validation and selection criteria. The specification implementations test whether an object satisfies all the rules of the specification. Consider the following class:

```
class Project {
  public boolean isOverdue() { ... }
  public boolean isUnderbudget() { ... }
}
```

The specification for overdue and underbudget projects can be decoupled from the project and made the responsibility of the other classes.

```
public interface ProjectSpecification {
  public boolean isSatisfiedBy(Project p);
}

public class ProjectIsOverdueSpecification implements
  ProjectSpecification {
  public boolean isSatisfiedBy(Project p) { ... }
}
```

This makes the client code more readable and flexible too.

```
If (projectIsOverdueSpecification.isSatisfiedBy(theCurrentProject) {
  ... }
```

STRATEGY PATTERN

The strategy pattern, also known as the Policy Pattern is used to make algorithms interchangeable. In this pattern, the varying "part" is factored out.

Consider the following example, which determines the success of a project, based on two calculations: (1) a project is successful if it finishes on time, or (2) a project is successful if it does not exceed its budget.

```
public class Project {
  boolean isSuccessfulByTime();
  boolean isSuccessfulByBudget();
}
```

By applying the strategy pattern we can encapsulate the specific calculations in policy implementation classes that contain the algorithm for the two different calculations.

```
interface ProjectSuccessPolicy {
  Boolean isSuccessful(Project p);
}

class SuccessByTime implements ProjectSuccessPolicy { ... }
class SuccessByBudget implements ProjectSuccessPolicy { ... }
```

Refactoring the original Project class to use the policy, we encapsulate the criteria for success in the policy implementations and not the Project class itself.

```
class Project {
  boolean isSuccessful(ProjectSuccessPolicy policy) {
    return policy.isSuccessful(this);
  }
}
```

COMPOSITE PATTERN

This is a direct application of the GoF pattern within the domain being modeled. The important point to remember is that the client code should only deal with the abstract type representing the composite element. Consider the following class:

```
public class Project {
  private List<Milestone> milestones; private List<Task> tasks;
  private List<Subproject> subprojects;
}
```

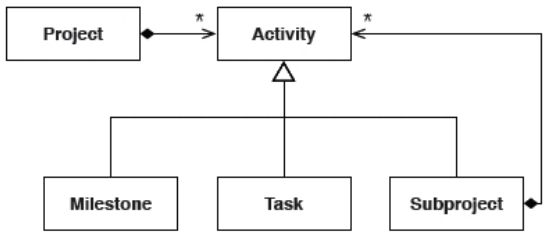
A Subproject is a project with Milestones and Tasks. A Milestone is a Task with a due date but no duration. Applying a composite pattern, we can introduce a new type Activity with different implementations.

```
interface Activity {
  public Date due();
}

public class Subproject implements Activity {
  private List<Activity> activities;
  public Date due() { ... }
}

public class Milestone implements Activity {
  public Date due() { ... }
}

public class Task implements Activity {
  public Date due() { ... }
  public int duration() { ... }
}
```

Now the model for the Project is much simpler.

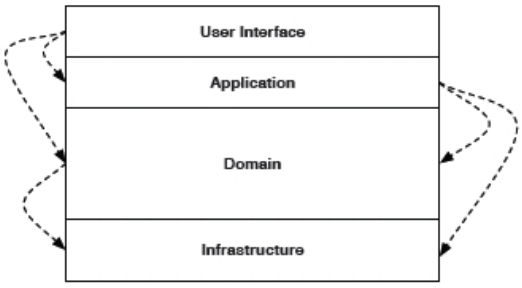
```
public class Project {
  private List<Activity> activities;
}
```

A UML representation of this model is shown below.



APPLICATION ARCHITECTURE

When the focus of design is on creating domain models that are rich in behavior, then the architecture in which the domain participates must contribute to keeping the model free of infrastructure too. Typically, a layered architecture can be used to isolate the domain from other parts of the system.



Each layer is aware of only those layers below it. As such, a layer at a lower level cannot make a call (i.e. send a message) to a layer above it. Also, each layer is very cohesive and classes that are located in a particular layer pay strict attention to honoring the purpose and responsibility of the layer.

User Interface	Responsible for constructing the user interface and managing the interaction with the domain model. Typical implementation pattern is model-view- controller.
Application	Thin layer that allows the view to collaborate with the domain. Warning: it is an easy "dumping ground" for displaced domain behavior and can be a magnet for "transaction script" style code.
Domain	An extremely behavior-rich and expressive model of the domain. Note that repositories and factories are part of the domain. However, the object- relational mapper to which the repositories might delegate are part of the infrastructure, below this layer.
Infrastructure	Deals with technology specific decisions and focuses more on implementations and less on intentions. Note that domain instances can be created in this layer, but, typically, it is the repository that interacts with this layer, to obtain references to these objects.

Aim to design your layers with interfaces and try to use these interfaces for "communication" between layers. Also, let the code using the domain layer control the transaction boundaries.

RECENTLY ADDED PATTERNS

Big Ball of Mud	This is a strategic design pattern to deal with existing systems consisting of multiple conceptual models mixed together, and held together with haphazard, or accidental, dependent logic. In such cases, draw a boundary around the mess and do not attempt to try sophisticated modeling within this context. Be wary of this context sprawling into other contexts. The original pattern was written by Brian Foote and Joseph Yoder and is available at www.laputan.org/mud/mud.html
Domain Events	Sometimes domain experts want to track the actual events that cause changes in the domain. Domain events are not to be confused with system events that are part of the software itself. It may be the case that domain events have corresponding system events that are used to carry information about the event into the system, but a domain event is a fully- fledged part of the domain model. Model these events as domain objects such that the state of entities can be deduced from sets of domain events. Event objects are normally immutable since they model something in the past. In general, these objects contain a timestamp, description of the event and, if needed, some identity for the domain event itself. In distributed systems, domain events are particularly useful since they can occur asynchronously at any node. The state of entities can also be inferred from the events currently known to a node, without having to rely on the complete set of information from the entire system.

ABOUT THE AUTHOR

OBI OBEROI is a Principal Developer Evangelist and a Software Architect at Paree Labs Inc., a company that specializes in designing, developing and maintaining enterprise level distributed applications including mobile apps using mainly Microsoft centric and hybrid technologies. Obi has been working with .NET since the first betas and has been engaged in the agile space since 2006. Obi has a Master's degree in Computer Science and runs a local Dot Net User Group in the greater Toronto area and is actively involved in organizing Developer Workshops, Code Camps et. al., and speaks at User Groups and Conferences besides being a passionate member of the developer community. Oberoi has worked on many projects from medium to large scale engagements including architecture, design, and development. Over the course of his career, he has designed and helped to create systems for the Financial, Retail, Health and Logistics sectors.

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795

VERSION 1.0