Greater Quality Through Tactical Domain-Driven Design

John R. Connolly

john@articulatedomain.com

1. Abstract

Quality software, for many, means that software does the work it needs to do today without any defects, i.e. functional quality. This narrow definition gives us a sense of success in the present, but what about the software's ability to absorb change for the future? The reality is that for many organizations millions of dollars of maintenance budget are wasted on software that functions well for the initial intent but does not absorb new features easily. Newer measures in the DevOps arena such as the DevOps Research and Analysis (DORA) metrics are helpful when learning how adaptive we are. But how do we improve those numbers and potentially decrease total cost of ownership when change is impacting the business positively and the software change is not able to keep up? Today, in addition to all the attributes of quality software we have come to expect, we now must now consider at least one non-functional quality attribute. That attribute is *adaptive* or *adaptability*. This is important because useful software seems to always need new functionality. Many have experienced great pain of trying to update software not designed to absorb change. Domain-Driven Design (DDD) and related paradigms, tools, and mindsets can help us get to new levels of adaptability. Specifically, Tactical DDD enables teams to set and respect software boundaries based on the domain the software it serves. These clear boundaries make code much more navigable and change far less frustrating. Tactical DDD then can be augmented by other paradigms like advanced event driven design, promoting even more autonomy of function reducing the ripple effects of change. This can create new levels of non-functional quality that typically reduces wasted maintenance budget and improves the performance of the software development life cycle (SDLC) for all who contribute to the software product.

2. Biography

John R. Connolly started his career in database development in 1997 in the U.S. Air Force and has a B.S. in Organizational Management. He is currently the Principal Consultant of Articulate Domain, a company he started in 2019. He specializes in aligning solution designs with the problem domain through Domain-Driven Design, distributed systems design and uses tools like Eventstorming (Brandolini) to drive collaborative models with business, stakeholders and developers. John holds a patent for systems design of essential oil distillation automation. John is completing a master's degree in the clinical mental health field and resides in Salem with his wife Sandy and two cats.

1 Introduction

1.1 Current State of Software Design

Logical code can easily get disorganized fast as organizations ask developers to apply new feature after new feature quickly. In the industry, we have a term for code that is generally hard to navigate and reason about as a whole. We call this a Big Ball of Mud (BBoM). BBoMs are usually produced in what is often called a “Feature Factory” where backlogs are processed day in and day out. These feature factories generally do not have a *behavioral* design paradigm with which they ascribe or utilize.

The BBoM architectural model is so popular that it is now considered by many to be the most widely adopted architectural pattern in the world. It’s not intentional, but it is pervasive. Many believe it is inevitable that software eventually just must be difficult to advance with new capabilities. This perpetuates the adaptability quality problem. BBoMs are so difficult to work our way out of that many tend to acquiesce to the enhancement delays they cause. Once it gets too difficult to maintain, most opt for a rewrite of the whole system or choose to never upgrade the software again in any significant way, losing great competitive advantages. With the advent of new design paradigms like DDD there is not only a way to avoid these conditions, but there are paths to repair the BBoM as well.

1.2 Low Quality Costs

The total cost of ownership of maladaptive software can potentially ruin a company’s bottom line. For those who need empirical data to better understand this problem can read the robust book on the topic by Capers Jones and his team, “The Economics of Software Quality”, where he analyzed hundreds of companies and came to the conclusion that “Although formal design and code inspections originated more than 35 years ago (as of 2011), they still are the top-ranked methodologies in terms of defect removal efficiency.” (Jones et. al. 2011)

The waste in 2020 and 2021 was not insignificant. “According to ‘The Cost of Poor Software Quality in the U.S.’ by the Consortium for Information and Software Quality (CISQ), the collective bill in the U.S. for defective software in 2021 was an estimated $2.41 trillion, up almost 16% from the 2020s $2.08 trillion. That’s more than the GDP of all but a dozen countries. And it doesn’t even count an estimated $1.52 trillion in ‘technical debt’ (TD)—accumulated software vulnerabilities in applications, networks, and systems that have never been addressed but will have to be paid eventually.” (Poor Software Quality Can Cost Time and Money, Straightforward Solutions Are Available 2023)

1.3 Didn’t Agile Solve This?

Agile was introduce to the market in 2001 to course correct the hyper extended delays of the Waterfall SDLC model (“Manifesto for Agile Software Development.” 2001).

There is a very good reason why Agile did not solve this adaptability issue. The word Agile was meant to convey the idea that *projects*, *teams* and *software* could be more agile. In practice, however, the industry focused primarily on agile projects and teams rather than systems themselves (Agile and the Long Crisis of Software n.d.). This lulled many leaders to implement Agile as a team management practice in a sort of assembly line type of mentality. While this team-oriented focus provided iterative *project management* alternatives to the long and drawn-out phases of waterfall, it generally did not address the need to design novel systems in any significant way. These project management activities became a commercial opportunity cementing these inefficiencies into contracts with companies providing Agile frameworks, often called Scrum, SAFe etc. Even with the advent of product-centric thinking, the industry did not address these design needs as there was more of a focus on task flow than design guidance in many of those implementations. Agile implementations mainly focused on project process and ceremony with little to no attention to curbing the tide on poor designs. As management pushed developers to produce new

features faster, measuring them by their sprint goals rather than their solution outcomes, many felt they needed to cut corners leaving no room to design adaptive systems.

1.4 Wait. Waterfall Again?

It may seem that introducing this idea of behavioral design could then take us a step back in time to the days of Waterfall. Waterfall had introduced a lot of rigid steps that prevented any change that would require rolling back to an already completed stage of the SDLC. This forced most projects to keep any stage of the process open for a long time to ensure no changes would be asked for before moving on to the next stage, creating a slow overall delivery model (Abraham 2022). Design had to be fully complete before development could start. Development had to be fully complete before testing could start and so forth. In addition, when a project was started and completed, the documentation had to be accurate and complete for all the capability captured. The waterfall process was so arduous that modifications were generally too expensive to accomplish. While Waterfall did have a lengthy design step, that level of design and detail was so extreme that software rarely made it to market in time to meet the demand. That is why Agile became so popular. Waterfall design is often referred to Big Up Front Design. Agile for the most part never implemented an intentional design step sponsored by the domain. This all or nothing approach to behavioral design is extreme on both ends and does not strike a good balance. Iterative behavioral design should be integrated into the SDLC.

1.5 Becoming Adaptive Intentionally

We will utilize the term *adaptability* rather than agility to reference our new key quality software design attribute to reduce confusion. In the long run, the quality of being adaptive, or software adaptability, can provide companies with something they often run out of - time. We already know that, while design adds energy to the SDLC, most CTOs know it generally saves time during the UX development cycle. In fact, good UX design often subtracts workload at the UI developer's keyboard. DDD translates that same benefit into the logical layer of software. Utilizing Tactical DDD principles, developers and testers alike have a much clearer target and a workable vision. When they know where methods and their related properties should coalesce and where they should not, this can reduce churn significantly. Therefore, it can take less time to develop software to a mature adaptive state. If teams want to serve their companies well for the duration of the need, which is often years, it will pay to adopt principles of DDD. This can be iteratively applied to the SDLC and will help teams steer clear of the BBoM outcomes.

DDD addresses two primary software design concerns that can impact the SDLC. They are Strategic and Tactical (Evans 2003). We won't spend much time on the Strategic concepts of DDD except to say that Domain-Driven Designers work on strategic designs when learning how to draw the line between one subsystem and another. Strategic DDD proposes investigating the notions of Subdomains and Bounded Contexts to support those decisions. Once those boundaries are formed, then Tactical DDD can be applied inside those boundaries. Tactical DDD domain models can struggle internally which is a good thing to be aware of. That condition often highlights a need to reassess Strategic DDD boundaries. When this happens in design time, it saves incredible amounts of wasted energy when those interoperability choices only need to be altered on paper.

So, there is an interplay that often happens between Strategic and Tactical DDD focal points as each informs the other of clues to improve design choices at a macro and micro level. Tactical DDD provides for the protection of logic, cohesion of domain concepts and reliability of functionality within those boundaries. Tactical DDD aims to improve feature capability absorption while reducing side effects. Adaptive design tactically is a critical outcome of quality DDD.

1.6 Design is Already Socially and Culturally Accepted

As mentioned previously, UX design is a known improvement to the SDLC and is often required before front-end developers can craft User Interfaces. As soon as a new project is announced, one of the very first things that is accomplished is the UX department's design of the UI. Wireframes are mulled over and

refined. Time is spent gathering customer feedback and design opinions from UX experts to craft a rough draft of exactly how users should interact with the system and how that system should provide feedback to the users. None of this is generally programmed at the beginning. This is smart. It is smart because it reduces waste when the system is in development mode. UI developers will have far fewer questions while crafting real-world applications at the presentation level because the context is made very clear by the design guides and the wireframes that show what is needed. As we will discuss, the same is true for behavioral design via Tactical DDD and is further economized by the tools behavioral designers are using to accomplish these designs.

To put a finer point on this correlation between UX and behavioral design, UX Design is desired because it makes development more profitable in several ways. Customers more readily like the first truly developed representation of the systems under development and meet the acceptance criteria more quickly. Developers do not have to spin code like a never-ending unsolved Rubik's Cube. A quality, well thought-out, UX design infuses a sense of direction and confidence into the process.

The great thing about the UX design model is that it can be used for future changes. This level of iterative design is also a benefit for the developers and the domain as whole as it saves time and keeps the design domain appropriate.

1.7 But We Make Data Models!

Data is not behavior, yet it is highly affected by behavior. Many IT leaders push for data models on the heels of having a good UX design or even in parallel. They have been trained through decades of *application development* reinforcement that teams need to connect the UX design to data models well to have a great system. If it were only so.

Udi Dahan, who masterfully teaches an Advanced Distributed Systems Design course, rightly points out that most development leaders and architects were trained in the days of application development. The world shifted to API and Event-Driven systems with decoupled modules or microservices, but the developer base was never fully trained on what that looks like when designing *systems* of logical software behavior. Decades ago, applications were singular systems with one database, the entire UI and all the behavior. Access 2, FoxPro, and many other desktop software development packages empowered software crafters to do it all in the 1980's and 90's. UI, Business Rules and Data coexisted in the same file. When servers came around, that development paradigm was applied to distributed systems. Distributed systems need an added layer of intentionality if they are going to be designed to adapt to new capabilities in the future.

2 The Ultimate Heart of the Matter

2.1 The Missing Piece

Intentional domain honoring design is generally not well understood and is not implemented to make behavior reduce friction for the business it serves. If this were a tiny fraction of the software developed, this paper would not be written. Often, clients of Articulate Domain, are asked, "Percentagewise, how much of the code in your more mature systems are the business rules or the behavior of the system vs the presentation and data definitions?" Invariably the answer so far has always been between 65-80% behavior. If true for most complex systems, then this means that most teams are leveraging the power of quality and time-saving design for only 20-35% of the mission critical software they are developing. Most of the companies interviewed have never heard of DDD or behavioral design in general. Hopefully by now we have surfaced a plausible reason why many quality issues, adaptive or otherwise, might easily creep into any system.

It is important to note that all software is designed and does what it is designed to do. If behavior is not intentionally designed, then it is unintentionally designed as the developer codes the software. This is painful for software developers who are guessing what a good domain honoring design would look like. If

companies won't make UI developers program without good wireframe design, why do they ask or even tell the business logic developer to churn through poor representations of behavior until one day they present a working system? And, once those systems are more mature, they are so maladaptive to change, developers will refuse to modify these rigid systems in any significant way. To make matters worse, it seems to be that many developers quit this type of job to get hired at another company where they will likely and unintentionally build another maladaptive system.

2.2 Testing Matters Too

It is also important to note that designing the behavior of systems is directly related yet not the same as designing tests. Tests should be designed and developed in the context of the behavior of a system. It is hard to test a system that is designed by developers on the fly, better known as emergent design, because the target design changes randomly as the developers realize their designs do not quite meet the need. If we are following a DDD paradigm then the design should point very easily to the need of the domain. These designs will feed into testing initiatives so that testing can be more confident as well. This includes Test-Driven Development (TDD) and Behavioral-Driven Development and other testing paradigms. Once domain needs are identified and are agreed upon, then tests at any level should reflect the needs at the level of the test. For instance, a unit test should ensure not only that the method works but works according to the domain. Any threshold, or limitation the domain puts on that method should be included and so forth.

In a distributed system, the domain need is generally supposed to be divided up into smaller services that perform specified capabilities. In this case, sub-system tests not only should be exclusive to the sub-system but should reflect any domain capabilities the system provides. When testing interactions between systems, generally governed by what we commonly call an *interface contract*, then those contracts should ensure domain concepts that impact those contracts are accounted for.

2.3 Little to No Intentional Design Can Work - Sometimes

For some software there is little need for Tactical DDD. Software that does not have a lot of behavior is a great candidate for such a decision. Just be sure there is not much behavior to plan for when making that design choice. It might be safe to design for a little bit to verify that it is unnecessary.

Another area that will not benefit much from a holistic design is the proof of concept. Unless the design itself is the focus of the investigation, the team should skip to just trying ideas that solve a specified problem. This type of project rarely has the adaptive rigor needed to ship products to production.

2.4 Some Software Benefits from Meticulous Behavioral Design

There is a time to do a lot of up-front intentional design for a longer period. This is a good thing for software that puts lives at risk or when profitability requires intricate clean design. Space modules and life-support systems are some examples of these devices that generally should not skip design steps and should be reviewed and tested for thoroughness and accuracy.

2.5 Most Software Needs Good Enough Design

Good enough design for complex software is hard to define but we do need a working definition. Good enough design is a process that affords a team the opportunity to model a solution that solves today's needs well and leaves that solution open to enhancement. This model should be accomplished quickly, answering critical questions well enough providing avenues for upgrades in the future. This is a skill that may take the proverbial 10,000 hours for a designer to reach an expert level of ability. Teams can often accomplish good enough design by taking time to decouple concerns at the transactional level quickly and making good domain honoring design choices.

3 Modern Behavioral Design

3.1 What is DDD Basically?

Eric Evans published his seminal book Domain-Driven Design: Tackling Complexity at the Heart of Software in 2003. A small segment of the industry knows about DDD, its uses, and benefits. Even though Microsoft consultants have written about it, European countries have generated a large following, Martin Fowler has described it, a plethora of videos are posted on YouTube, and Oxford is now teaching it as a software design course, it remains out of view of many.

Evans took programming practices up until the time of his book, and advanced abstract organizational thought to improve not only code structure but its relevance to the domain it serves. Object Orientation (OO) by that time was very much becoming the popular way to craft code, but it lacked a well-accepted paradigm that classified user requirements concepts as *domain* object types using *domain terminology* for its naming convention. OO also did not have an industry-wide method to craft *domain boundaries* based on the Language they use. Evans created a way to bridge this gap to make software much easier to work with by facing the complexity head on. DDD ended up being a language and technology agnostic paradigm that could be used by developers all over the world.

At a high level, DDD is a way to classify your domain for the sake of both the business experts and the IT experts alike such that there is no need to translate the terminology used in code into the terminology used in the business. There is one language per context and per context the business and IT speak the same language. In practice this is difficult to achieve, but the more it is accomplished the less friction there is in communication between business and IT. In fact, it is considered a desired outcome to make software look like something the business can understand no matter the software language used.

3.1.1 DDD: A Closer Look

DDD at its core is what the name indicates. A more descriptive way to state it would be to say that DDD is a set of principles where Domain-Driven Designers can work with Domain Experts (or Subject Matter Experts) to map Domain Boundaries and Domain Capabilities into Domain Models that programmers should be able to clearly understand and can more confidently express in code. This may be hard to absorb, so let's look at it from other angles before we describe the programmatic elements of DDD.

The *domain* is the area of a business, agency or organization that needs some solution or set of solutions to solve problems, accomplish something or somehow advance the product or service offering. Many will label the domain as the *problem space*. Domains can be further subdivided into subdomains. Subdomains can be further subdivided into smaller subdomains.

High level domain examples are insurance, ecommerce, banking, healthcare, air travel, hospitality, gaming, retail, education and the list goes on. Those domains are supported by other subdomains like accounting, shipping and operations and those can be further subdivided. If DDD were easy, we would just name the organizations' various departments and develop a software system for each department with interconnected data. Those who have tried to do this generally and quickly run into a myriad of complications. While it is not helpful to naïvely use these subdomains in this manner, these areas of the domain own the ground truths essential to making DDD adaptive systems. It is important to explore these domain areas and take steps to design models that are inspired by them and support their mission well. When those domains are driving the solution designs, domain to system solution alignment increases significantly. Honoring the domain this way is a necessary ingredient for most complex systems to be highly adaptive. One important point to note is that DDD is a paradigm of design, and not a silver bullet. No two DDD projects are the same, yet they use the same basic principles to craft quality designs.

3.1.2 Tactical Domain-Driven Design Benefits

DDD tactically affords developers a way to encapsulate domain capabilities more cleanly, reliably and less coupled than they would without this design paradigm. Creating solutions this way removes many of the negative side-effects that stem from common software changes. Also, once this skill is embedded in each team member working on a DDD designed system, this paradigm gives those team members a more defined language to discuss the elements in the system. Knowing Tactical DDD when working with a system that uses this paradigm makes it easier to navigate lines of code. When all teams operate this way in a large organization, it makes it easier for a developer to move from team to team and still be productive. Additionally, if developers veer from the normalcy of their DDD coding practice, they can steer the course back or at least make a case for the anomaly. Being able to come back months or years after working on a codebase and confidently understand it is a great benefit of Tactical DDD.

3.2 Key DDD Tactical Elements

3.2.1 Involve the Domain Experts and Discover Their Language

Before we get into these important Tactical DDD building blocks it is critical to understand that using Tactical DDD building blocks well does not make software adaptive by itself. Using Tactical DDD to craft a rich domain map that serves the domain well for the future is the holy grail of DDD. We cannot just build highly organized code and call it good. ***We must organize these building blocks in a way that truly expresses and serves the behavioral needs of the domain it serves.*** This is the essence of DDD. If we do not use these building blocks to express and serve the behavioral needs of the domain, we might as well stop pretending to serve the domain with DDD. This does not mean that we achieve perfection in our designs. We should strive to understand the domain need and map solutions that are likely to adapt as those needs change. Domain needs are best described by the Domain Experts in the industry that is being served.

Domain Experts often need encouragement, empathy and inspiration to detail domain needs. This can be a process of learning, doing, and learning some more. This makes iterative Tactical DDD a fit in an Agile environment. This helps fill the void that an Agile implementation focused on projects (or products) and teams primarily suffer from.

A word of caution should be expressed here. Domain Experts can be influenced by the systems that they work in. In fact, clear, complete and concise domain term definitions often get replaced by definitions in antiquated software or OTS packages. This can clutter the conversation and is something a good facilitator of DDD modeling sessions will catch and explore. Language is important in DDD and making it clear is a key part of the paradigm.

3.2.2 Domain Contextual Language (Ubiquitous Language)

Communication always rides on the common understanding of a language. Morse Code, Egyptian Hieroglyphics, French - it does not matter. Even though there is a common understanding in the dictionary of the terms we use, often there are several nuanced definitions. This can wreak havoc when trying to understand each other clearly. Fortunately, when both sides are clear on the terminology, they communicate so well it is as if they can finish each other's sentences, or code as it were. When there is ambiguity or differences of understanding, assumptions and miscommunication make agreements weak or not even possible. How many times do we encounter missteps that started with multiple definitions for a word or multiple words for a definition. A DDD focus will clear up these issues.

Clear language is at the heart of DDD. Strategic DDD efforts push for clarity on what Eric Evans calls the Ubiquitous Language. Though, ironically, that phrase is confusing to most when they first encounter it. To clear up this term, what Evans is driving at is that within each contextual sphere of influence, the same words always have the same definitions. This does not mean that the same word must have the same definition in separate contexts. For instance, the word "invoice" might have three different definitions in three subdomain contexts within a single business context, but it should never have two or more

definitions within one context. This led to the term “bounded context”, indicating that terms and their definitions are bound and never used differently unless the domain changes the term or its definition. The Ubiquitous Language is likely easier to understand as a ***contextual language***. So, we will use that term in the sections that follow. Bounded contexts or the contextual language of a domain or subdomain is a work in progress. As the work to define these terms matures, so will the language. This is no different than any language that is long lived and evolving.

Once we understand that a bounded context is a boundary around contextual terms and their definitions, i.e. the contextual language, then many will ask, “where do we use those terms?” These terms are used throughout the culture of the domain. The business, agency, organization, their print material, their code base and any other format that uses these ideas should use the same language. In code, the nouns, verbs, adjectives etc., from the domain get used in the namespaces, properties, packages, modules, functions, methods, subroutines, interfaces, tests other constructs.

Gone should be the days of naming class objects or functions using the animal kingdom, favorite automobiles/motorcycles or sci-fi movie characters. ***Professional software should use the terms of the profession it serves.*** This is critical to adaptivity because any translation between the domain and the software is one added point of potential communication failure. The more we remove translation need, the more likely we understand how to read the code against the backdrop of the domain.

3.2.3 Aggregates

When designing software from a Tactical DDD perspective, behavioral code is encapsulated in Aggregates. The systems outside of the aggregate communicate with the aggregate through an Aggregate Root. Inside the aggregate there will usually be domain behavior and information. These behaviors and information are organized into interconnected Entities, Value Objects, Domain Services, and Domain Events. New aggregates are often created by a Factory. Once the first successful execution of an aggregate is completed, the state is saved through a Repository into some datastore. Some will store the latest full state only, while others will save a new event on top of all historical events of that aggregate so that the state changes can be replayed later. Saving Domain Events in a series is known as EventSourcing – a topic we will not be expounding upon in this paper. Some of these terms may not be clear since these terms can be used in an overloaded fashion.

3.2.4 An aggregate is a conceptual whole of a capability with all the behavior and data needed to achieve that capability. Aggregates take a command to process domain behavior and if needed store the resulting data. After the Aggregate attempts to fulfill that command, whether it succeeds or fails, it reports the appropriate outcome back to that calling party. The critical thing about this is that no caller can directly call an element inside the aggregate. They can call only the root of the aggregate. All implementation details are hidden from view from all other systems.

The aggregate root is generally an entity, but not always. The key point is, there is one point of contact on the aggregate called the aggregate root and it exposes the interface(s) that afford other systems the opportunity to request that the aggregate attempt to do what that aggregate was designed to do.

Aggregates enable adaptivity in that if the behavior can be updated and the name of the commands are still relevant, then we do not have to make changes to the rest of the connecting systems. Also, if the aggregate can be extended without harm to past transactions, then we are backwards compatible as well. If the aggregate is not capable of being extended, then we can make a new aggregate with new functionality like versioning an API. This can become cluttered with too many aggregates, but the design concept forces us to think deliberately about this choice.

A real-world example of an Aggregate and its Root could be a MemberSubscriptions aggregate. The information needed to make a successful payment would be supplied to a command on the Aggregate Root, ApplyMemberSubscriptionPayment. Inside the aggregate, there may be many steps of the process like gaining access to the credit card processing system, and more. The world outside only knows that it

can call ApplyMemberSubscriptionPayment and it will either return successful or as a failed attempt. It could have information about the condition of the failure that allows the calling code to determine if trying again is a business-friendly option. It may even have an event that posts to let the rest of the systems that care to know that the payment was accomplished so that notifications and more can be automated.

3.2.5 Entities Give Identity

Entities are classes that have identities generally in the real world but could be a manufactured reality as well. The main idea here is that a John Smith in New York and John Smith in San Francisco are not the same people, and each have their own identities and need their own identifier. This allows us to work with the correct data for the correct John Smith. Entities help us remain safe as we remain adaptive since we know that an entity is not just some random data object. Entities deserve careful attention when we make modifications, especially in this day of regulation and information safety. Adaptive safety is a key to making sure we make quality changes, not just easy changes.

3.2.6 Value Objects Reduce Cost

When we can, we should favor Value Objects over Entities. Value Objects provide us with a disposable nature to our software. An example is the object that holds a value for the hex color #E3242B *(Red Rose)*. That object is the same as another object in memory for the hex color #E3242B. The more we rely on reusable value objects, the less we care about having the same number of those value objects as elements that need them, the more we can just use a suitable and available value object with the data we want.

There is no reason to have 50,000 objects with the same hex value for red when we need to consume it 50,000 times, unless there is a performance reason, we can use one. Value objects are easier to work with and will reduce complexity by the fact that we don't have unnecessary identifiers on these objects. Additionally, it is commonly understood that maintaining Value Objects costs less than maintaining Entity types given the disposable nature of a Value Object.

3.2.7 Domain Services

The term *service* is used a lot in a lot of IT departments, and we do not have a common industry understanding of what that term means. In DDD terms, a Domain Service is a functional capability that does not require storage of transactional entity data but does perform a calculation or some other evaluation on input data. These are often standalone methods or functions. This building block should be used only when the method or function has no true entity, or value object home.

These services can be quite alluring to use, but it is important to realize that they in themselves can start the slippery slope of a big ball of mud. When we think we have a domain service, we should ask where the data is coming from that is being calculated or evaluated. If it is all from the same entity, maybe that function should be attached to that entity.

For example, instead of having a BillPay Domain Service as an aggregate all its own, maybe there should be a command "PayInvoice" on the aggregate entity "CustomerInvoice" with a command at the Aggregate Root that makes paying the invoice possible from the invoice. This process is likely not a Domain Service either because it tracks and saves some domain data.

An example of a domain service could be a calculation that is needed by many aggregates such as determining the roadway distance between two points on a map. That distance is needed to accomplish other things, but the distance calculation is utilized by a lot of other processes. It takes in two points, goes through the process of learning the shortest distance and then returns it to the caller. We never need to save that data when it is calculated.

3.2.8 Events Record History

Events are an amazing and growing concept in the custom software industry. Events, from a DDD or distributed systems perspective, are a record of an occurrence in the past. Once that event happened it is part of history. It happened and that record is not something that can change. Bank accounts are a good example. If a deposit was partial, then another deposit would produce a second event that would bring the balance to the correct value. We cannot change the first deposit, it happened. We can only react to it. Events are powerful in that they draw a line in the sand of reality. We know an event happened because it is emitted and stored. Events can trigger new commands for automation, or they can help a user know what happened.

Events help bring visibility to the computational flows of systems allowing us to know more about what we need to change when we are adapting to the future. EventSourcing strategies store events as they occur on an aggregate so that we can rewind and fast-forward through historical records of that aggregate to learn the steps that the data took to get to some state. Event Sourcing is a practice that the DDD community has been learning and growing with for many years. This concept of Event Sourcing is made easier by understanding DDD and Command Query Responsibility Separation (CQRS). CQRS at its heart is a pattern that separates the paths of the workflow that changes data from the paths that report it.

Domain Events are powerful, but the management of an event driven system is not trivial. Care and attention should be paid to the process of understanding how Events are triggered and how they are used to communicate to the rest of the system.

3.3 Eventstorming A Domain

The key to adaptive systems through DDD is that the domain is pulled into focus so that your software is so aligned that when the business changes, the software can change right along with it. To accomplish this, you will want to know what the Domain is doing and what it needs. Eventstorming and related paradigms have become popular in some circles to assist with this process.

If you have not heard about the orange sticky revolution started by Alberto Brandolini, let’s introduce you to Eventstorming – a tool to make DDD even easier. Eventstorming by its very nature is best facilitated by someone with training. Workshops are best started with a group of Domain Experts helping everyone understand domain flows.

We can say a lot about EventStorming, however, Alberto Brandolini and Paul Rayner each wrote great books on the topic. Here, we will just talk about the high-level understanding of the tool. In general, EventStorming Facilitators will take teams through three levels of Eventstorming: Big Picture; Process Modeling; Software Design. Let’s look at each at a high level.

3.4 Big Picture Eventstorming

During a Big Picture Eventstorming workshop, Domain Events are the focus. The Domain Experts write on the orange sticky notes all the Domain Events they know about for the process that we are designing. No order to start with is needed. Just get them all up on the board. Back to billing for example, some events might be, Bill Generated; Bill Sent; Bill Paid and so forth. Once the energy for Domain Events starts to subside, the team will be directed to put them on a timeline to get a sense of the flow.

3.5 Process Modeling Eventstorming

More types of process stickies are introduced. *Personas, Commands, Processes, Views, Business Policies* and *Questions* get added to the flow. These processes consider manual steps as well as automated steps. The facilitator shows a legend and soon the Domain Experts, if they are interested, will start to learn how to describe their processes in a way developers can more easily understand than typical requirements documents or user stories.

3.6 Software Design Eventstorming

Developers will take the process model and work to make a software design. This will coalesce the data and the behaviors where needed and decouple them where needed. The aggregates we discussed earlier will emanate from this process. The language can be OO or Functional. Java, Ruby, or .Net. A solution template that is DDD honoring will make applying these designs to code much easier. This is how we can save time and money the way UI developers do with UX designs.

4 Applying DDD To the SDLC

Adding a design step to a process that never made room for behavioral design could prove challenging. Let's explore that idea.

4.1 Project Management Education

Project Managers have struggled with managing software projects, especially if they were trained to manage projects that fit the manufacturing world. Managing software projects with a design step can help since design is also a manufacturing concept. However, it will still be important to learn how to manage *thought work* efforts for the unique challenges it brings as compared to manufacturing project management. If the Project Manager learns what DDD is, this can help them move toward a common language with the designer/developers putting these principles to work.

4.2 Some Executives Need to Champion Quality Behavioral Design

Many developers need permission to do good work with new steps. They can tend to feel like quality is not important more than quantity in many corporate cultures so they will not gravitate to new concepts like DDD that provide quality and eventually quantity. Executives need to find a way to message that they prefer a good quality adaptive product development mentality, because it provides for more capability in the future for less.

4.3 SDLC Upgrades

The main SDLC upgrade is the ability to accomplish a good design. When there are questions, the developer needs to have permission to ask for a clarification of a design. The Domain Expert needs to be part of the team, and the developer needs to be able to ask questions. Architectural decisions will come from these conversations so it will be important to make sure impacted teams are aware of those.

4.4 DDD Team Formation Considerations

Teams come in all shapes and sizes. It is generally best to keep team sizes small: 3-8 people. DDD will be more adaptive when the ones who create aggregates or at least maintain them have a bit of deep knowledge of the domain. They will make better decisions with the design because they understand that aspect of the domain. If they only have only a few team members, then collaboration is easier. If 95% of their changes are inside the aggregates they work on, their daily flow will be faster.

5 Conclusion

The way many craft software today with an emergent design model often can create stress on the SDLC. If we intertwine Tactical Domain-Driven Design, we can reduce the amount of churn needed to create the solutions we need to be successful. We can encapsulate behaviors into Aggregates that make change more predictable. Taking time to plan a solution architecture by aligning it to the domain makes change a little more manageable and can reduce waste. Eventstorming can make that process more achievable in an enterprise setting. Designs can flow into software templates that are structured for DDD solutions.

References

Abraham, Max. 2022. “Waterfall Methodology – Ultimate Guide.” *Management.org* (blog). March 21, 2022. https://management.org/waterfall-methodology.

“Agile and the Long Crisis of Software.” n.d. Logic(s) Magazine. Accessed August 19, 2024. https://logicmag.io/clouds/agile-and-the-long-crisis-of-software/.

Brandolini, Alberto. 2015. “Introducing EventStorming.” Leanpub. September 9, 2015. https://leanpub.com/introducing_eventstorming.

Evans, Eric. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA: Addison Wesley.

Forsgren, Nicole, Jez Humble, and Gene Kim. 2018. Accelerate: The Science of Lean Software and Devops: Building and Scaling High Performing Technology Organizations. It Revolution Press.

Jones, Capers, Olivier Bonsignour, and Jitendra Subramanyam. 2011. *The Economics of Software Quality*. Boston, MA: Addison-Wesley Educational.

“Manifesto for Agile Software Development.” 2001. Agilemanifesto.org. Accessed August 14, 2024. https://agilemanifesto.org.

“Poor Software Quality Can Cost Time and Money, Straightforward Solutions Are Available.” 2023. CSO Online. March 13, 2023. https://www.csoonline.com/article/574723/poor-software-quality-can-cost-time-and-money-straightforward-solutions-are-available.html.

Rayner, Paul. 2019. “The EventStorming Handbook.” Leanpub. February 15, 2019. https://leanpub.com/eventstorming_handbook.